Refactoreo de Diagramas de Clases UML empleando Slicing de Modelos

Diego A. Cheda [*]
DSIC, Universidad Politécnica de Valencia
Valencia, 46022, España
dcheda@dsic.upv.es

y

Salvador V. Cavadini
Project EVEREST, Institut National de Recherche en Informatique et Automatique (INRIA)
Sophia-Antipolis, 06902, Francia
Salvador.Cavadini@sophia.inria.fr

Resumen

El refactoreo es un proceso que permite mejorar la estructura interna de un sistema sin modificar su comportamiento. Su aplicación sobre diagramas UML es actualmente campo de investigación. Por otra parte, el slicing constituye una técnica ampliamente usada en diversas áreas de la ingeniería de software (como ser *debugging*, *testing*, reuso, mantenimiento, etc.) y más recientemente ha encontrado aplicación en el campo de los modelos UML. Sin embargo, la combinación de ambos procedimientos en el dominio de lenguajes como UML no ha sido objeto de indagación hasta el momento. En este trabajo se presenta una aproximación para su uso combinado.

Palabras claves: Refactoreo, refactoreo de modelos, transformación de modelos, *slicing* de modelos.

1. INTRODUCCIÓN

En el mundo real, la necesidad de evolucionar constituye una propiedad intrínseca del software. Éste, con cierta frecuencia, se modifica y adapta a nuevos requerimientos del entorno. Así, se torna fundamental diseñar y construir sistemas cuyo mantenimiento sea simple: con código sencillo de comprender, con una estructura capaz de ser alterada sin complicaciones, entre otras características.

Además, los métodos de desarrollo modernos, como *eXtreme Programming*[1] [3], se basan en un proceso iterativo e incremental y buscan que cada paso no sólo permita aumentar el número de funcionalidades del producto sino también mejorar la calidad del mismo (por ejemplo, incrementar su eficiencia, facilitar su reusabilidad y comprensión, etc.).

Al mismo tiempo, el coste y esfuerzo implicados en el desarrollo de software conduce inevitablemente al reuso y a la evolución de los sistemas existentes. Por esa razón, el diseño debe enfocarse

[*]Este trabajo está parcialmente financiado por la Unión Europea (FEDER - Fondo Europeo de Desarrollo Regional) y el Ministerio de Educación y Ciencia de España (MEC) bajo la subvención TIN2005-09207-C03-02.

[1]Metodología *lightweight* para el desarrollo rápido de software.

hacia la obtención de componentes factibles de ser reutilizados. La programación orientada a objetos se distingue por promover la reutilización. Sin embargo, a menudo, es indispensable realizar cambios en el software a fin de lograr que sus partes sean más reusables. En ese sentido, Arnold [1] define la *reestructuración* (*restructuring*) como:

> "[...] *the modification of software to make the software easier to understand and to change or less susceptible to error when future changes are made.*"[2]

La reestructuración tiene como motivación principal entender y mejorar programas desestructurados o pobremente estructurados.[3] Esto implica reducir saltos incondicionales, eliminar código muerto, dividir el sistema en módulos, mejorar el formato del código (identación), entre otras cuestiones.

Opdyke [18] sugiere el concepto de *refactorización* (*refactoring*) como una variante de la reestructuración aplicada a programas orientados a objetos. Fowler [7] propone la siguiente definición de refactoreo:

> "*The process of changing a* [object-oriented] *software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure.*"[4]

De esta forma, y por nombrar algunos casos, un programador podría desear cambiar el nombre de un método por uno más representativo, reemplazar código duplicado por llamadas a funciones que encapsulen esa tarea, cambiar la interfaz de una operación para facilitar su reuso, etc.; estas acciones dotarían al sistema de mayor calidad. Entonces, el refactoreo consiste en aplicar al sistema una secuencia de pequeñas transformaciones tendientes a conservar el comportamiento de éste, pero mejorando su aspecto interno.

En general, el refactoreo se ha empleado principalmente sobre código fuente. No obstante, su utilización es posible a nivel de modelo, en este caso en diagramas de *Unified Modeling Language* (*UML*).

Por otra parte, el *slicing de programas* es una técnica que permite extraer aquellas sentencias que (potencialmente) afectan el valor de cierto conjunto de variables en un programa [25]. Entre los múltiples usos del slicing se encuentran algunas aplicaciones en relación con la reestructuración que pretenden agrupar código en funciones o procedimientos con el propósito de hacerlo reusable [12, 13]. La técnica también es aplicada en software orientado a objetos [14] y actualmente se ha investigado su uso en diagramas de clases de UML [10]. En el presente trabajo se intenta bosquejar una primera visión de cómo podría emplearse el slicing como parte del proceso de refactoreo de diagramas de clase UML.

A continuación se muestra un breve estado del arte sobre refactoreo en general y en particular sobre su aplicación en modelos. Luego, se describe la técnica de slicing y cómo aplicarla junto con refactoreo en diagramas de clases. Finalmente, se presentan las conclusiones y futuras actividades de investigación.

---

[2] "[...] *la modificación de software para hacerlo más fácil de entender y de modificar o menos susceptible de contener errores cuando se realicen cambios en el futuro*". Traducción de los autores.

[3] Este tipo de programas también se conoce como "*spaghetti code*", término peyorativo empleado para indicar que el programa fuente es complejo y díficil de entender, generalmente, debido a las estructuras de control enrevesadas que entorpecen su lectura.

[4] "*El proceso de cambiar un sistema de software* [orientado a objetos] *de tal modo que no se altere el comportamiento externo del código, mejorando su estructura interna*.". Traducción de los autores.

2. REFACTOREO

El concepto de refactoreo fue formalizado por Opdyke [18], Brant [20] y Roberts [21] y popularizado por Beck [3] con su libro sobre *eXtreme Programming*.

En líneas generales, puede entenderse el refactoreo como el conjunto de técnicas o heurísticas para mejorar la estructura interna del software preservando su comportamiento. El propósito de estos cambios es hacer que un programa sea reutilizable, fácil de entender y modificar.

Si bien su aplicación se limitó principalemente al nivel de código fuente (programas), recientemente, surgieron desarrollos que permiten distinguir tres tipos de técnicas:

- Refactoreo de programas: es un tipo específico de transformación de programas en el que se aplican modificaciones al código fuente con el objetivo de mejorar la calidad de éste, pero conservando siempre su comportamiento.
- Refactoreo de modelos es una clase especial de transformación aplicada a diseños. Este tipo de refactoreo se aborda con más profundidad en el parágrafo 3.
- Refactoreo de requerimientos, propuesto por Russo *et al* [22], se aplica a especificaciones de sistemas realizadas en lenguaje natural con el fin de descomponer tales requisitos en *puntos de vista* (*view point*). Cada punto de vista encapsula descripciones parciales de algún componente del sistema. Las relaciones entre éstos últimos se identifican para detectar interdependencias problemáticas. El proceso de reestructuración permite luego aumentar la comprensión, mantenimiento y evolución mediante la detección de inconsistencias en las relaciones existentes entre los puntos de vista.

2.1. Actividades relacionadas con el proceso de refactoreo

El proceso de refactoreo consiste en una serie de actividades que puede estructurarse de la siguiente manera:

1. Identificar dónde debe realizarse el refactoreo. Esto comúnmente se efectúa empleando ciertos “patrones” para reconocer a qué parte del artefacto de software (programa, modelo o requerimiento) es posible aplicarle una refactorización. Tal acción se denomina usualmente “detección del *mal olor*” (*bad smell detection*).
2. Determinar qué reglas de refactoreo deben aplicarse a los puntos identificados.
3. Garantizar que el refactoreo preservará la semántica del artefacto del software. La definición de refactoreo establece que debe preservarse el *comportamiento* del código, pero no proporciona una definición suficientemente precisa de *comportamiento* lo que dificulta su verificación.

   Opdyke [18] sugiere que, para un mismo conjunto de valores de entrada, el conjunto de valores de salida resultante de la ejecución del software debe ser el mismo antes y después de efectuar el refactoreo. Sin embargo, esto suele ser insuficiente ya que existen otros aspectos que deben considerarse como ser restricciones de tiempo, consumo de recursos, propiedades de seguridad, etc.

   Una forma práctica de enfocar el problema es llevar a cabo una rigurosa disciplina de *testing*. Con un conjunto amplio de pruebas sería posible efectuar las mismas sobre el software refactoreado para comprobar si éste las supera. Otro enfoque posible es definir formalmente las propiedades del software y luego del refactoreo verificar (por ejemplo, utilizando técnicas como el chequeo de modelos [4]) que estas propiedades han sido preservadas.

4. Aplicar el refactoreo.

5. Evaluar el efecto producido por el refactoreo en términos de incremento de la calidad del software. Una forma de valorar la calidad alcanzada luego de realizar refactoreo es medir o estimar el impacto sobre ciertas características internas tales como complejidad, acoplamiento o cohesión y otras relativas a la *performance* del software.

6. Mantener la consistencia entre los componentes refactoreados y los demás artefactos del software.

3. REFACTOREO DE MODELOS

En esta sección describiremos cómo el concepto de refactoreo fue ampliado para ser empleado en modelos a través de un conciso resumen de las principales actividades de investigación en el área de refactoreo de UML.

3.1. Breve estado del arte

En [23] se describe un conjunto inicial de reglas de refactoreo, en lenguaje natural para el caso de diagramas de clases y con pre y postcondiciones expresadas con *Object Constraint Language* (*OCL*) para los diagramas de estados. El referido trabajo también aborda el problema que presenta la aplicación de transformaciones que impactan en diferentes vistas del modelo UML.

Astels [2] propone una aproximación para realizar refactoreo en UML aplicando transformaciones de mayor granularidad (por ejemplo, extraer una jerarquía de clases) y cómo utilizar los diagramas para identificar posibles aplicaciones de refactoreo.

Tanto en [8] como en [9] se expone una extensión del metamodelo UML que permite verificar pre y postcondiciones, componer refactoreos y detectar "*bad smells*".

Una consideración importante a la hora de determinar el éxito del refactoreo es evaluar qué efecto provoca en el modelo. Tahvildari *et al* [24] definen un proceso de refactorización que analiza el impacto que produce la aplicación de ciertas transformaciones en un sistema. Para tal fin, compara mediciones realizadas con métricas indicativas de la calidad antes y después de efectuar el refactoreo. Además, se propone un catálogo de métricas que permiten detectar situaciones en las que una transformación particular puede ser aplicada para mejorar la calidad del sistema. En [5], los autores muestran una colección de reglas de refactoreo y discuten acerca de cómo pueden ser automatizadas. Markovic [15] investiga cómo chequear, reusar y componer reglas de refactoreo y presenta un algoritmo que verifica la aplicabilidad de dichas transformaciones sobre un modelo dado. Recientemente, Markovic y Baar [16] definen un conjunto de reglas para refactoreo de diagramas de clases de UML basadas en el estándar *Query/View/Transformation* de OMG.

3.2. Refactoreo de diagramas de clases

En general, para realizar *refactoring* se definen reglas que permiten transformar el modelo a una forma que se considere o estime más adecuada de acuerdo a cierto criterio de calidad y sin modificar el comportamiento del sistema.

Tomando como punto de partida las operaciones definidas por Opdyke [18] y por Roberts [21], es posible redefinir las mismas a fin de adaptarlas a los diagramas de clases UML. Estas son las principales operaciones:[5]

[5] Para más detalle se puede consultar [23].

- Renombrar elementos
- Agregar, remover y mover atributos, métodos y clases
- Especializar y generalizar clases

Un modo interesante de representar éstas reglas puede encontrarse en [16] donde las transformaciones son descriptas mediante una gramática de grafos permitiendo que el refactoreo se realice a través de reescritura de grafos [19].

4. SLICING

En esta sección se describen algunos conceptos básicos sobre slicing y posteriormente se muestra la aplicación de la técnica sobre modelos.

4.1. Slicing de programas

Weiser [25] plantea que los programadores, durante la depuración de un programa, dividen a éste en piezas de código coherentes y no necesariamente contiguas. Estas porciones de código, denominadas *slices*, afectan potencialmente los valores calculados para algún punto de interés del programa, al que se denomina *criterio de slicing*. En otros términos, el slice de un programa *P* con respecto al criterio de slicing (*p, v*), donde *p* es una posición o sentencia determinada del programa y *v* la variable de interés, es un subprograma *P'* tal que contiene aquellas sentencias que afectan al valor de *v* calculado en *p*.

La aplicación natural del slicing fue la de auxiliar a los programadores en las tareas de depuración. Sin embargo, una vez reconocida su utilidad, se percibió que su uso también podía resultar beneficiosa en otras áreas tales como *testing*, integración, comprensión y mantenimiento de programas, paralelismo, obtención de métricas, reuso e ingeniería reversa entre otras. Dado que cada una de estas áreas requiere slices con ciertas características o propiedades especiales, han surgido, acompañadas con sus correspondientes métodos de cálculo, varias nociones y definiciones de slice que difieren, en mayor o menor grado, de la original.

4.2. Slicing como un problema de alcanzabilidad en grafos

La noción esencial en el slicing es la de dependencia. Los valores calculados por la sentencia del criterio sólo pueden verse afectados por aquellos obtenidos por sentencias de las cuales la primera depende. Las dependencias de datos y de control suelen ser los tipos de dependencia que se utilizan para hacer slicing de programas. Es posible representar de manera explícita estas dependencias mediante el *grafo de dependencia del programa* (*PDG* por *program dependence graph*), Ferrante *et al* [6], aprovechan esta propiedad del PDG y describen un algoritmo para extraer slices desde él. Más tarde, en [11], Lakhotia generaliza el problema del slicing de programas como un problema de alcanzabilidad en grafos proveyendo así un marco común en dónde modelar distintos algoritmos de slicing y estudiar sus características.

5. SLICING DE DIAGRAMAS DE CLASES UML

Debido a la complejidad que representa manejar sistemas con gran cantidad de clases y relaciones entre ellas, es necesario contar con herramientas que faciliten la extracción de partes relevantes para determinado interés de un diagrama de clases.

Recientemente, Kagdi *et al* [10] introdujeron el concepto de slicing en diagramas de clases UML. Las definiciones que se detallan a continuación difieren de las dadas en [10] ya que sólo se estudiarán los diagramas de clases.

Considerando que un diagrama de clases puede ser visto como un multigrafo dirigido $M = (C, R, \Gamma)$ donde:

- $C = \{c_1, c_2, ..., c_n\}$ es un conjunto finito de clases del modelo,
- $R = \{r_1, r_2, ..., r_m\}$ es un conjunto finito de relaciones entre las clases y
- $\Gamma : R \mapsto C \times C$ es una función que vincula las clases y relaciones.

El conjunto finito de relaciones R incluye las relaciones como asociación, generalización, dependencia, etc. Las clases y relaciones se vinculan mediante la función Γ.

El slice de un modelo M se define como una función S sobre el modelo para un determinado criterio de slicing SC:

$$S(M, SC) = M' = (E', R', \Gamma') \subseteq M$$

El criterio de slicing SC es un par $SC = (I, A)$ donde

- $I \subseteq C$ es un conjunto de clases de interés,
- $A \subseteq R$ un conjunto con los tipos de relaciones que se recorrerán en el cálculo del slice.

El slice se calcula recorriendo el multigrafo M desde cada punto de interés $i_j \in I$ y teniendo en cuenta las relaciones $a_k \in A$. Así se obtienen todas aquellas clases que se relacionan con cada i_j empleando los tipos de relación de A.

6. USO DEL SLICING EN EL PROCESO DE REFACTOREO DE DIAGRAMAS DE CLASES

Como ya mencionamos en 2.1, el proceso de refactoreo incluye, entre otras actividades, la identificación de los componentes sobre los que se aplicará el refactoreo, la definición de las reglas de refactoreo a utilizar, la aplicación de las reglas y la verificación de que el proceso preservó la semántica del artefacto de software.

El slicing resulta de suma utilidad en estas actividades ya que permite una identificación precisa y posterior aislamiento de los componentes de interés y aquellos que serán potencialmente afectados por el facoreo. El slice ofrece una vista más clara del modelo y brinda la oportunidad de realizar las modificaciones de forma independiente del resto del modelo y sin afectar al mismo. Trabajar sobre el slice simplifica la tarea de la elección de las reglas de factoreo ya que el usuario debe concentrarse sólo en una parte del modelo y no en su totalidad. Las ventajas de trabajar sobre un slice son evidentes en la tarea de verificar que el refactoreo preservó la semántica del modelo. Como las operaciones sólo se aplican a una parte aislada del modelo, es posible determinar con más facilidad la preservación del entorno. Por ejemplo, si esta verificación se realiza a través del *testing*, el universo de casos de prueba necesarios para el slice será generalmente más acotado que el necesario para el modelo completo.

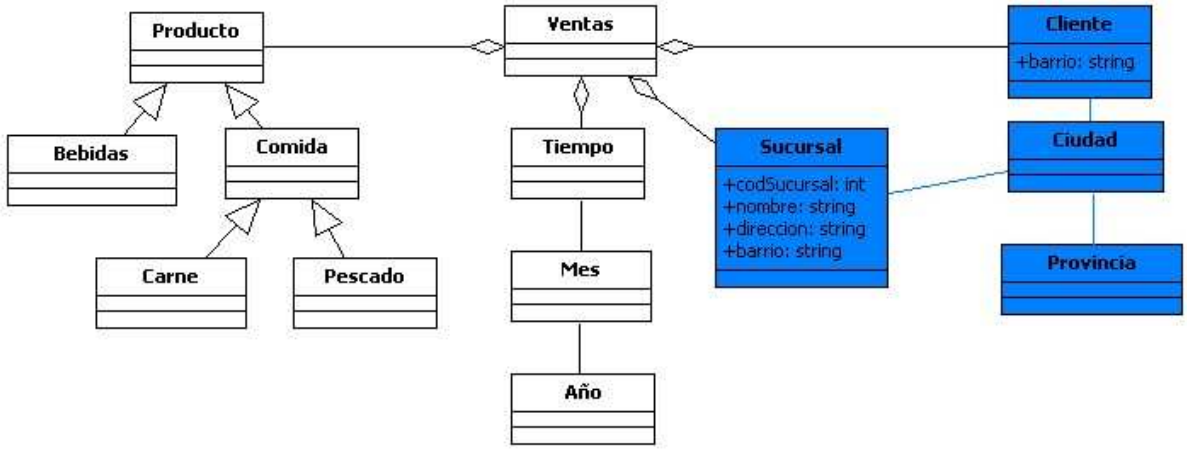

Figura 1: Diagrama de clases sin refactoreo.

6.1. Ejemplos de combinación de slicing y refactoreo

A continuación se presentan dos ejemplos de uso combinado de las operaciones básicas de refactoreo y el slicing de diagramas UML.

Ejemplo 6.1.1 *En el diagrama de clases de la figura 1 puede observarse que la clase* Cliente *tiene un atributo* barrio*. El diseño no es óptimo debido a que* barrio *seguramente repetirá en distintas instancias de* Cliente*. Por otra parte,* Sucursal *también tiene un atributo* barrio*. Esta duplicidad innecesaria dificulta el mantenimiento de los datos referidos a* barrio*. Para solucionar este problema es oportuno aplicar refactoreo.*

Si se utiliza slicing de modelos para aislar los componentes sobre los que efectuaremos la refactorización deberemos definir el criterio de slicing $SC = (I, A)$ *donde:*

- $I = \{Cliente\}$
- $A = \{Asociacion\}$

Tal como se observa señalado con color en la figura 1, con este criterio, se obtiene el slice $S(M, SC) = M' = (E', R', \Gamma')$ *donde:*

- $E' = \{Cliente, Ciudad, Provincia, Sucursal\}$
- $R' = \{Asociacion\}$
- $\Gamma' : R' \mapsto E' \times E' = \{(Cliente, Ciudad), (Ciudad, Provincia), (Ciudad, Sucursal)\}$

Luego, es posible aplicar un refactoreo para extraer una nueva clase desde otra. En este caso, se extrae la clase Barrio *a partir de* Cliente*. Esa nueva clase* Barrio *se vincula tanto con* Cliente *como con* Ciudad *a través de una asociación. Puede observarse en la figura 2 el resultado del refactoreo.*

Sin embargo, luego del refactoreo anterior, en la clase Sucursal *debe eliminarse el atributo* barrio *y reemplazarlo por una asociación. Así, se obtiene el modelo de la figura 3.*

De este modo, luego de refactorear el modelo a través de su slice, se obtiene uno equivalente pero de mayor calidad.

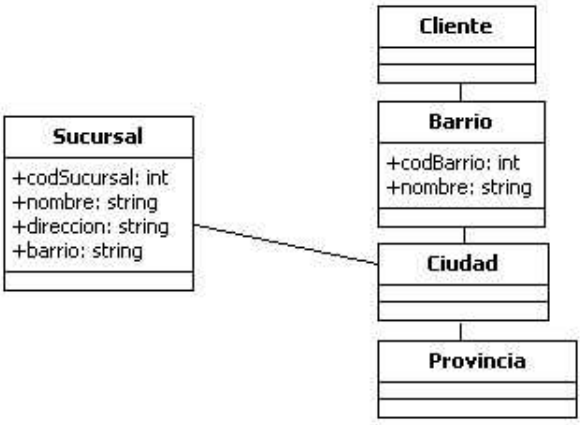

Figura 2: Extracción de una nueva clase.

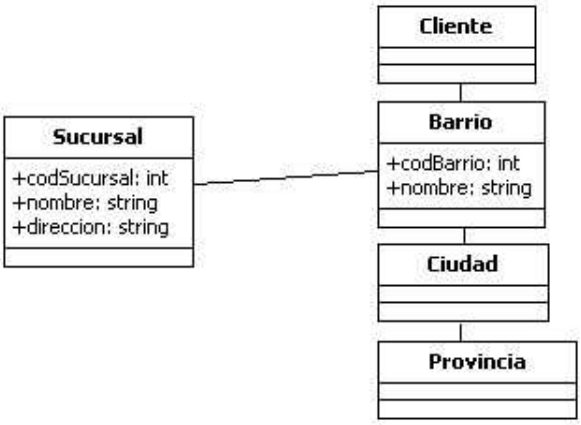

Figura 3: Eliminación de una asociación redundante.

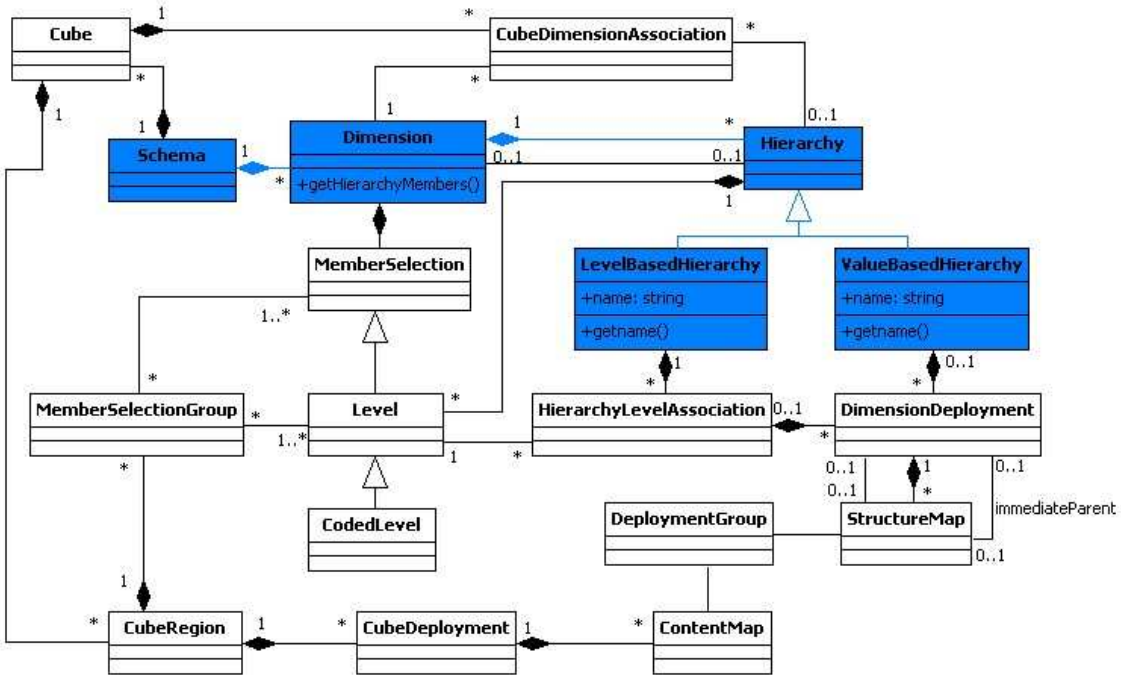

Figura 4: Metamodelo OLAP de OMG.

Nótese que si se elige como criterio de slicing $SC = (\{Sucursal\}, \{Asociacion\})$*, el slice obtenido sería idéntico al mostrado en la figura 1. Además, si en lugar de iniciar el refactoreo a partir de* Cliente *se iniciara desde* Sucursal *podría conseguirse el resultado mostrado en la figura 2, pero realizando un mayor número de pasos de refactoreo.*

Si bien el ejemplo presentado es simple, sirve para mostrar el potencial de la combinación del refactoreo con el slicing y brinda una primera aproximación sobre cómo esta idea podría ser aplicada con buenos resultados a diagramas mucho más complejos como el que se utiliza en el ejemplo siguiente.

Ejemplo 6.1.2 *La figura 4 muestra una parte del metamodelo OLAP (On-Line Analytical Processing) del estándar Common Warehouse Metamodel (CWM) de OMG (Object Management Group) [17] al que se le ha añadido algunos métodos y propiedades.*

En el diagrama de clases puede observarse que la clase Hierarchy *se especializa en dos subclases* LevelBasedHierarchy *y* ValueBasedHierarchy*. El diseño no es óptimo por dos motivos:*

- *Ambas comparten un mismo atributo (*name*) y*
- *Las subclases tienen dos métodos que proporcionan idénticos resultados (*getname()*).*

En este caso, si fuera necesario un mantenimiento de los métodos, se deberían modificar dos clases. Es posible mejorar esta situación mediante una serie de refactoreos.

Se identifica la sección del diagrama que es de interés empleando el criterio de slicing $SC = (I, A)$ *donde:*

- $I = \{LevelBasedHierarchy, ValueBasedHierarchy\}$

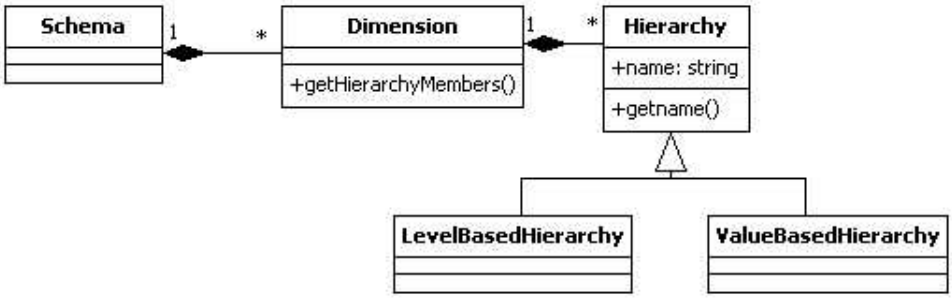

Figura 5: Refactoreo aplicado las clases *LevelBasedHierarchy* y *ValueBasedHierarchy*.

- $A = \{Generalizacion, Composicion\}$

Con este criterio, se obtiene el slice $S(M, SC) = M' = (E', R', \Gamma')$*, como se muestra resaltado con color en la figura 4, donde:*

- $E' = \{Schema, Dimension, Hierarchy, LevelBasedHierarchy, ValueBasedHierarchy\}$
- $R' = \{Generalizacion, Composicion\}$
- $\Gamma' : R' \mapsto E' \times E' = \{(Schema, Dimension), (Dimension, Hierarchy), (Hierarchy, LevelBasedHierarchy), (Hierarchy, ValueBasedHierarchy)\}$

Luego, es posible aplicar dos refactorizaciones sobre el slice:

- *Mover el atributo* name *a la superclase y*
- *Mover el método* getname() *a la superclase.*

De esta forma se eliminan los atributos y un métodos repetidos.
A pesar de que con el refactoreo aplicado el modelo es más adecuado, puede observarse que la clase Dimension *contiene un método que, por motivos de diseño, debería encontrase en* Hierarchy. *Por esa razón y como se ve en la figura 6, es conveniente transformar el diagrama de tal forma que dicho método pertenezca a la clase* Hierarchy.

7. CONCLUSIONES Y DIRECCIONES FUTURAS

El artículo presenta un pantallazo del refactoreo aplicado a modelos y del slicing para diagramas de clase UML como preámbulo de la introducción de la idea de usar estas técnicas de manera combinada. Como se indica, el slicing permite la identificación precisa de los componentes que serán objeto de refactoreo y sus relaciones. Es mediante esta delimitación que se facilitan las tareas de seleccionar el tipo de transformación a aplicar y de identificar los componentes potencialmente afectados por la misma.

Como puede observarse en los ejemplos aportados, la definición del criterio de slicing requiere que el usuario indique el conjunto de clases de interés y el tipo de relaciones entre componentes que deben tenerse en cuenta. La identificación de las clases de interés se realiza a partir del *bad smell*; sin embargo,sin embargo, no queda claro todavía en base a qué parámetros efectuar la selección de

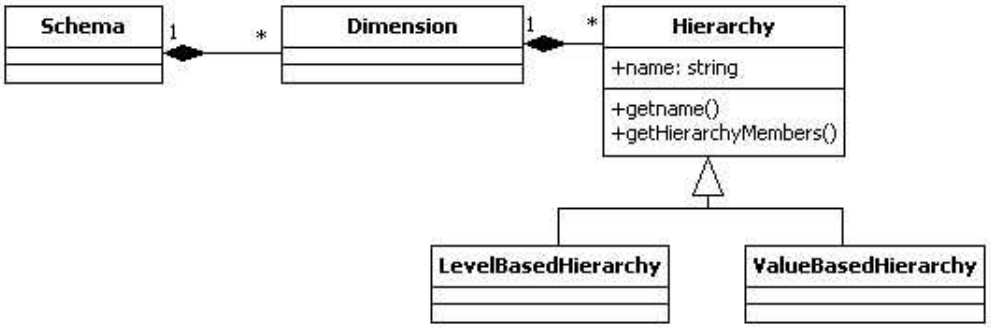

Figura 6: Refactoreo aplicado a la clase *Dimension*.

las relaciones del criterio de slicing. Es por esto que se presenta interesante la identificación de los factores que deben considerarse al definir el conjunto de relaciones del criterio de slicing.

También puede resultar provechoso el estudio de la utilidad en el refactoreo de la combinación de slices (unión, intersección, *chopping*).

REFERENCIAS

[1] R. S. Arnold. Software restructuring. *Proceedings of the IEEE*, 1989.

[2] Dave Astels. Refactoring with uml, 2002.

[3] Kent Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[5] Alexandre L. Correa and Cláudia Maria Lima Werner. Applying refactoring techniques to uml/ocl models. In *UML*, pages 173–187, 2004.

[6] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[7] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[8] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Enabling and using the uml for model-driven refactoring. In Serge Demeyer, Stéphane Ducasse, and Kim Mens, editors, *Proceedings WOOR'03 (ECOOP'03 Workshop on Object-Oriented Re-engineering)*, pages 37–40. Universiteit Antwerpen, July 2003.

[9] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent uml refactorings. In *UML*, pages 144–158, 2003.

[10] Huzefa Kagdi, Jonathan I. Maletic, and Andrew Sutton. Context-free slicing of uml class models. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 635–638, Washington, DC, USA, 2005. IEEE Computer Society.

[11] A. Lakhotia. Graph theoretic foundations of program slicing and integration, 1991.

[12] Arun Lakhotia and Jean-Christophe Deprez. Restructuring programs by tucking statements into functions. *Information and Software Technology*, 40(11-12):677–690, 1998.

[13] Filippo Lanubile and Giuseppe Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Trans. Softw. Eng.*, 23(4):246–259, 1997.

[14] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.

[15] Slavisa Markovic. Composition of UML described refactoring rules. In Octavian Patrascoiu, editor, *OCL and Model Driven Engineering, UML 2004 Conference Workshop, October 12, 2004, Lisbon, Portugal*, pages 45–59. University of Kent, 2004.

[16] Slavisa Markovic and Thomas Baar. Refactoring ocl annotated uml class diagrams. In *MoDELS*, pages 280–294, 2005.

[17] Object Management Group (OMG). Common Warehouse Metamodel (CWM) Specification. http://www.omg.org/docs/formal/03-03-02.zip, 2003.

[18] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[19] Detlef Plump. Essentials of term graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 51, 2001.

[20] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, 1997.

[21] Donald Bradley Roberts. *Practical analysis for refactoring*. PhD thesis, 1999. Adviser-Ralph Johnson.

[22] A. Russo, B.Ñuseibeh, and J. Kramer. Restructuring requirements specifications, 1999.

[23] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring uml models. In *UML; '01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 134–148, London, UK, 2001. Springer-Verlag.

[24] Ladan Tahvildari and Kostas Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 183, Washington, DC, USA, 2003. IEEE Computer Society.

[25] M.D. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.