# REFACTORING DE DIAGRAMAS DE CLASES UML

*Claudia Pereira* *Liliana Favre**

*INTIA - Facultad de Ciencias Exactas*
*Universidad Nacional del Centro de la Provincia de Buenos Aires*
*Tandil - Argentina*
*{ cpereira, lfavre @exa.unicen.edu.ar }*
**CIC (Comisión de Investigaciones Científicas de la Provincia de Buenos Aires)*

## 1. INTRODUCCIÓN

Se denomina *refactoring* al proceso de reestructurar software orientado a objetos aplicando una secuencia de transformaciones que preservan la funcionalidad del mismo a fin de mejorar alguna métrica. Es una actividad esencial para controlar la evolución del software facilitando futuras adaptaciones y extensiones. Cobró importancia en procesos de desarrollo como XP (eXtreme Programming) que requiere de la reestructuración de modelos y código existentes a partir de pasos pequeños y sistemáticos (Beck, 2000). También es fundamental en el contexto de técnicas de ingeniería reversa de sistemas *legacy*.

Algunas herramientas CASE UML proveen facilidades, si bien limitadas, para el *refactoring* sobre código, es decir ligado a la sintaxis de un lenguaje de programación en particular. Actualmente OMG (Object Management Group) promueve desarrollos de software basados en UML (OMG, 2004) a partir de una arquitectura *Model Driven* (MDA, 2004). MDA define un *framework* para modelar que separa la especificación de la funcionalidad del sistema de su implementación sobre una plataforma en una tecnología específica. La idea clave es la automatización de transformaciones de modelo-a-modelo. En este contexto se vuelven esenciales las técnicas de *refactoring* para mejorar los modelos de los distintos niveles de abstracción de un diseño.

En esta investigación se propone el *refactoring* de diagramas de clases UML enriquecidos con expresiones OCL (Warmer y Kepple, 2003) a partir de un sistema transformacional basado en reglas y estrategias. El objetivo de este sistema transformacional es proveer asistencia para los *refactorings* de modelo-a-modelo mediante la aplicación de reglas de transformación que preservan la funcionalidad del modelo original. Las transiciones entre versiones se realizan de acuerdo a reglas precisas basadas en la redistribución de clases, atributos, operaciones y asociaciones del diagrama. Durante el proceso de transformación, se necesitan estrategias para guiar la aplicación de las reglas de transformación que permiten construir un nuevo diagrama UML.

Se describen en la sección 2 los trabajos relacionados. La sección 3 presenta el sistema de transformación y un conjunto de reglas y estrategias para reestructurar clases y asociaciones. Finalmente la sección 4 concluye y discute futuros trabajos.



## 2. TRABAJOS RELACIONADOS

La primera publicación relevante sobre *refactoring* fue realizada por Opdyke (1992), quien establece precondiciones que aseguran la preservación del comportamiento para un conjunto de *refactorings* definidos en términos de C++. Roberts (1999) describe técnicas basadas en contratos de *refactoring*. Fowler (1999) aplica informalmente técnicas de *refactoring* sobre código fuente en el lenguaje de programación Java, explicando los cambios estructurales a través de ejemplos con diagramas de clases. Fanta y Rajlich (1998) y Fanta y Rajlich (1999) estudian el *refactoring* de código C++.

Varias propuestas proveen soporte para reestructurar modelos UML. Gogolla y Ritchers (1998) proponen transformar características avanzadas de diagramas de clases UML en construcciones más básicas con restricciones OCL. En Evans (1998) se establecen reglas que permiten transformar un modelo estático en otro diagrama con propiedades deducidas basado en el uso de transformaciones diagramáticas. En Sunyé y otros (2001) se presenta un conjunto de reestructuraciones y se explica cómo pueden ser diseñadas para preservar el comportamiento de modelos UML. Philipps y Rumpe (2001) reconsideran propuestas de refinamiento existentes para tratar formalmente con nociones de comportamiento, equivalencia de comportamiento y preservación de comportamiento. Whittle (2002) investiga el rol de las tranformaciones sobre diagramas de clases UML con restricciones OCL. Demeyer y otros (2002) describen el estado del arte sobre el área de *refactoring*. Porres (2003) define e implementa *refactorings* de modelos en base a transformaciones basadas en reglas. Van Gorp y otros (2003) proponen un conjunto de extensiones al metamodelo UML que permiten razonar sobre *refactoring* para todos los lenguajes orientados a objetos.
Varias herramientas automatizan distintos aspectos de *refactoring*. Por ejemplo, Guru (Moore, 1995) es una herramienta que soporta transformaciones automáticas para reestructurar jerarquías de herencia de objetos SELF preservando el comportamiento de los mismos. Smalltalk Refactoring Browser (Roberts y otros, 1997) es un *browser* avanzado para VisualWork que realiza automáticamente transformaciones que preservan el comportamiento. Together ControlCenter (TogetherSoft, 2004) aplica *refactoring* de código sobre requerimientos del usuario.

## 3. REFACTORING A PARTIR DE "REGLAS + ESTRATEGIAS"

Se propone en esta investigación aplicar reglas y estrategias para el *refactoring* de modelos estáticos UML anotados en OCL. Las reglas de transformación preservan el comportamiento observable del modelo original y pueden crear nuevos elementos en el modelo, modificar o remover existentes.
La Figura 1 muestra el proceso de transformación a partir de "reglas + estrategias". Cada *refactoring* se realiza sobre un subconjunto del modelo seleccionado por el diseñador manteniendo el historial de las transformaciones aplicadas para soportar *traceability*. Los *refactorings* resultan una técnica poderosa cuando son aplicados repetidamente en un modelo.

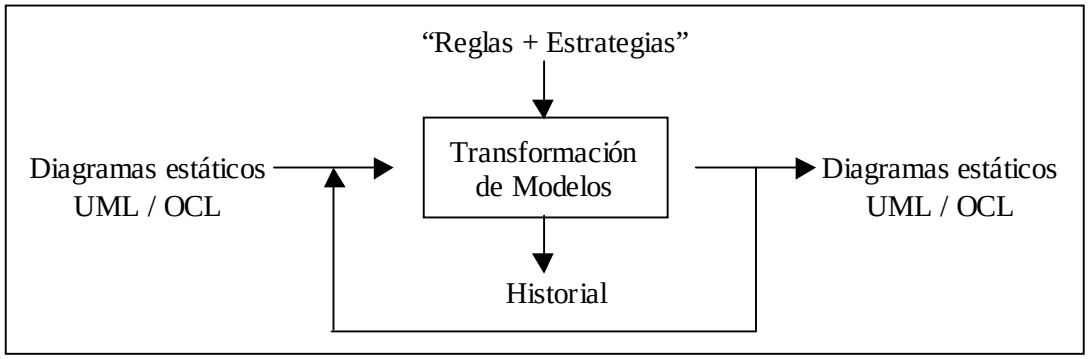


**Figura 1- Refactoring a partir de "reglas + estrategias"**

Las reglas de transformación se denotan mediante un esquema de entrada, un esquema de salida y condiciones de aplicabilidad que deben ser chequeadas para poder efectuar las transformaciones.
Se definen un conjunto de reglas para la reestructuración de jerarquías de herencia y un conjunto de reglas más poderosas para la reestructuración de diagramas de clases que permiten una transformación gradual y automática garantizando consistencia y equivalencia funcional.
Se describen en 3.1 un conjunto de reglas para el *refactoring* de clases y estrategias para aplicarlas y en 3.2. reglas de transformación vinculadas a asociaciones.

### 3.1. Refactoring de jerarquías de herencia

**Reglas de transformación**

Se describen a continuación un conjunto de reglas para transformar jerarquías de clases.

*Folding*
Une dos clases que tienen una relación de herencia directa obteniendo una nueva clase que reúne el comportamiento de ambas. Esta regla reduce el nivel de una jerarquía de clases en aquellos casos donde no hay un interés particular en el comportamiento de la clase base, ya sea porque es una clase abstracta o porque la cantidad de operaciones de la clase no justifica tener otro nivel en la jerarquía.

*Abstracción*
Divide el comportamiento de una clase generando dos clases que mantienen una relación de herencia directa. Mediante la aplicación de esta regla se puede abstraer en una nueva clase base el comportamiento más general reconocido dentro de otra clase.

*Composición*
Dadas dos clases sin relación de herencia entre sí, se forma una nueva clase con la unión de ambas. Esta regla es útil para agrupar comportamiento y reducir la herencia múltiple fundamentalmente cuando las clases comparten un descendiente.

*Factorización*
Factoriza las operaciones equivalentes en una nueva clase base a partir de dos clases sin relación de herencia entre sí. Esta regla elimina operaciones y atributos duplicados.

*Unfolding por atributos*
Divide el comportamiento de una clase, generando dos clases que mantienen una relación de herencia directa. Tales clases surgen a partir de realizar una partición de los atributos en dos subconjuntos disjuntos. Esta regla es útil cuando las operaciones no se refieren simultáneamente a todos los atributos, sino que sólo hacen referencia a algunos de ellos.

**Estrategias para el refactoring de jerarquías de herencia**

Las reglas de reestructuración son unidades básicas de transformación, es decir a partir de ellas se pueden construir secuencias particulares de aplicación de reglas denominadas estrategias de reestructuración. A continuación se describen algunas de ellas.

*Composición sin duplicación de métodos*
Integra el comportamiento de dos clases no genéricas eliminando la repetición de métodos, ya que se obtiene una sola clase que reúne todo el comportamiento sin repetición de aquellos métodos que fueran equivalentes. La secuencia de reglas es: Factorización → Composición → Folding.

*Factorización y unión de clases derivadas*
Factoriza métodos equivalentes en una clase base existente e integra el comportamiento de dos clases no genéricas que derivan de ella. La secuencia de reglas que define esta estrategia es la siguiente: Factorización → Composición → Folding.

*Factorización de métodos equivalentes a una clase base*
Es útil en aquellos casos donde se identifique la existencia de métodos equivalentes en clases derivadas y se desee abstraer tal comportamiento a la clase padre existente. Reduce la repetición de métodos en las clases derivadas e integra el comportamiento común a la clase padre. La secuencia de reglas es: Factorización → Folding.

*Abstracción de métodos a una clase base*
Promueve comportamiento a la clase padre a partir de un subconjunto de métodos sin crear nuevas clases y sin aumentar la cantidad de niveles de la jerarquía. La secuencia de reglas es: Abstracción → Folding.

*Unfolding por atributos a una clase base*
Promueve comportamiento a la clase padre a partir de un subconjunto de atributos A, sin crear nuevas clases y sin aumentar la cantidad de niveles de la jerarquía. La secuencia de reglas es: Unfolding (A)→ Folding

### 3.2. Reglas para el refactoring de asociaciones

Se describen a continuación algunas reglas aplicables a asociaciones en diagramas de clases.

*Agregado de una asociación transitiva*
Dada una asociación entre las clases A y B y una asociación entre las clases B y C, permite derivar una asociación entre A y C, determinando apropiadamente el tipo de asociación, las multiplicidades y la navegabilidad de cada final de asociación. (Whittle, 2002)

*Borrado de una asociación transitiva*
Dada una asociación entre las clases A y B, una asociación entre las clases B y C, y una asociación entre A y C permite borrar la asociación transitiva entre A y C. (Whittle, 2002)

*Sustitución de una asociación*
Dada una asociación R, permite sustituirla por una asociación menos restringida. Es decir, en cualquier asociación R, un final de asociación E con multiplicidad *mult1* puede ser sustituida por un final de asociación con multiplicidad *mult2*, donde $mult1 \subseteq mult2$. (Evans, 1998)

*Promoción de una asociación*
Dada una asociación R con multiplicidad *mult1* (conectada a la clase A) y multiplicidad *mult2* (conectada a la clase B). Si B es una subclase, entonces R puede ser ascendida a la superclase de B con la condición que su multiplicidad con A después de la transformación sea opcional, es decir, $0 \in mult1$. (Evans, 1998)

*Unión de asociaciones unidireccionales*
Dadas dos clases que se referencian una a la otra, con navegabilidad en sentidos contrarios, permite unirlas en una única asociación bidireccional. (Kollmann y Gogolla, 2001)

## 4. CONCLUSIONES

Se presentó una propuesta para el *refactoring* de modelos estáticos UML con expresiones OCL. El énfasis está dado en transformaciones de modelo-a-modelo que preservan el comportamiento. Las transiciones entre las versiones se realizan de acuerdo a reglas precisas basadas en la redistribución de clases, atributos, operaciones, herencia y asociaciones en el diagrama para facilitar futuras extensiones y adaptaciones. Las transformaciones pueden deshacerse, lo cual provee *traceability* en el *refactoring* de modelos. La noción de corrección en este contexto está basada en una especificación del metamodelo UML en el lenguaje algebraico NEREUS (Favre, 2003).
Si bien el conjunto de reglas definido permite realizar interesantes *refactorings*, es limitado ya que no contempla transformaciones que involucren las diferentes vistas UML. Actualmente se está investigando reglas asociadas a patrones de diseño.
En Enriques y otros (2002) se presenta un prototipo para la reestructuración de jerarquías de clases en C++. Para demostrar la factibilidad de esta propuesta, se implementará un prototipo que asista

en el *refactoring* a nivel de diseño, implementando un conjunto de reglas de transformación básica (folding, unfolding, abstracción, composición, factorización) y reglas vinculadas a asociaciones.

## REFERENCIAS


Beck, K. (2000). *Extreme Programming explained.* Addison-Wesley.

Demeyer, S., Du Bois, B., Stenten, H. y Van Gorp, P. (2002). Refactoring: Current Reasearch and Future Trends Language Description, *Tools and Applications*. (LDTA 2002).

Enriques, S., Mariezcurrena, C. y Ortega, M. (2002). *Reestructuración de jerarquías orientadas a objetos.* Tesis de grado, TR 287. UNCPBA, Buenos Aires, Argentina.

Evans, A. (1998). Reasoning with UML Class Diagrams. *Workshop on Industrial Strength Formal Method*, IEEE Press.

Fanta, R. y Rajlich, V. (1998). Reengineering an Object Oriented Code. *IEEE International Conference on Software Maintenance*, 238-246.

Fanta, R. y Rajlich, V. (1999). Reestructuring Legacy C Code into C++. *IEEE International Conference on Software Maintenance*, 77-85.

Favre, L. (2003). El Lenguaje NEREUS. *Reporte interno. Grupo de Tecnologías de Software, INTIA*. UNCPBA, Buenos Aires, Argentina.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Programs.* Addison-Wesley.

Gogolla, M.y Richters, M. (1998). Transformation Rules for UML Class Diagrams. *UML' 98 Workshop*, Springer-Verlag, Berlin, 92-106.

Kollmann, R. y Gogolla, M. (2001). Application of UML Associations and Their Adornments in Design Recovery. *8th Working Conference on Reverse Engineering*, IEEE, Los Alamitos.

MDA (2004). *The Model Driven Architecture*. Disponible en www.omg.org/mda

Moore, I. (1995). Guru – A tool for Automatic Restructuring of Self Inheritance Hierarchies. *Tools USA 95 (Technology of Object-Oriented languages and Systems*, Tools 17), 267-275.

OMG (2004). *Unified Modeling Language Specification*, v. 1.5. Object Management Group. Disponible en www.omg.org

Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks*. Tesis doctoral, Universidad de Illinois, Urbana-Champaign.

Philipps, J. y Rumpe, B. (2001). Roots of refactoring. *10th OOPSLA Workshop on Behavioral Semantics*, Tampa Bay, Florida, USA.

Porres, I. (2003). Model Refactorings as Rule-Based Update Transformations. *UML 2003*, Springer Verlag. Disponible en www.tucs.fi/Publications

Roberts, D. (1999). *Practical Analysis for Refactoring*, Tesis doctoral, Universidad de Illinois.

Roberts, D., Brant, J. y Johnson, R. (1997). A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, Vol 3, N° 4.

Sunyé, G., Pollet, D., LeTraon y Jézéquel, J. (2001). Refactoring UML Models. *UML 2001*, Lecture Notes in Computer Science 2185, 134-138.

TogetherSoft, ControlCenter (2004). Disponible en www.togethersoft.com

Van Gorp, P., Stenten, H., Mens, T. y Demeyer, S. (2003). Towards automating source-consistent UML Refactorings. *UML 2003*, Springer Verlag. Disponible en win-www.uia.ac.be/u/lore

Warmer, J y Kepple, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition.* Addison-Wesley.

Whittle, J. (2002). Transformations and Software Modeling Languages: Automating Transformations in UML. *UML 2002*. Lecture Notes in Computer Science 2460 (eds. J. Jezequel; H. Hussman), Springer-Verlag, 227-241.