Refactorings en el Contexto de MDA

Claudia Pereira
Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina
cpereira@exa.unicen.edu.ar

y

Liliana Favre [1]
Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina
lfavre@arnet.com.ar

Abstract

The Model Driven Architecture (MDA) is an initiative proposed by the Object Management Group (OMG) to model centric software development. It is based on the concepts of models, metamodels and automatic transformation from abstract models to executable components or applications. The outstanding ideas behind MDA are the different kinds of models, the separation of the specification of the system functionality from its implementation on specific platforms and the control of the model evolution. Refactorings play a fundamental role in the perfective evolution of the models. In this paper, we propose refactoring specification based on metamodeling techniques. The refactoring rules are specified in the Object Constraint Language (OCL) as contracts between metamodels. We propose a uniform treatment of MDA-refactorings at different abstraction levels. We describe foundations for MDA-based refactoring that enable extending the functionality of the existing CASE tools in order to improve the MDA-based process quality.

Keywords: refactoring, MDA, metamodeling, model transformation

Resumen

La arquitectura model-driven (MDA) es una iniciativa propuesta por el Object Management Group (OMG) para la especificación de sistemas basada en el uso de modelos. Los elementos básicos en esta arquitectura son los modelos, metamodelos y transformaciones automáticas desde modelos abstractos a componentes ejecutables o aplicaciones. Las ideas centrales son la clasificación de modelos en distintos niveles de abstracción, la independencia entre la especificación de la funcionalidad del sistema y su implementación sobre una plataforma en una tecnología específica y el control de la evolución de los modelos. Los refactorings tienen un rol fundamental en la evolución perfectiva de los modelos. En este artículo se propone una especificación de refactorings basados en MDA utilizando una técnica de metamodelado, especificándolos en el Object Constraint Language (OCL) como contratos entre metamodelos. Se presenta un tratamiento uniforme de los refactorings para los distintos modelos propuestos por la arquitectura MDA. La incorporación de esta propuesta a la funcionalidad de las herramientas CASE existentes permitiría mejorar la calidad de los procesos basados en MDA.

Palabras claves: refactoring, MDA, metamodelado, transformaciones de modelos

[1] Comisión de Investigaciones Científicas de la Provincia de Buenos Aires

1 INTRODUCCIÓN

Refactoring es una técnica sistemática para mejorar diseños de software orientado a objetos a partir de la aplicación de transformaciones que preservan el comportamiento. Actualmente, se lo considera una práctica relevante en los procesos de desarrollo basados en la arquitectura *Model Driven* (Model Driven Arquitecture - MDA).

MDA es una iniciativa propuesta por Object Management Group (OMG) para mejorar los procesos de desarrollo de software centrados en modelos. Las ideas subyacentes a MDA son separar la funcionalidad del sistema de su implementación sobre plataformas específicas y administrar la evolución del software desde modelos abstractos a implementaciones con un alto grado de automatización y de interoperabilidad con múltiples plataformas y lenguajes de programación. Los conceptos de modelos, metamodelos y transformaciones son cruciales para plasmar estas ideas [10].

MDA distingue diferentes tipos de modelos según el grado de abstracción: modelos independientes de la computación (Computer Independent Model - CIM), modelos independientes de la plataforma (Platform Independent Model - PIM), modelos específicos a una plataforma (Platform Specific Model - PSM) y modelos específicos a la implementación (Implementation Specific Model - ISM). Los diferentes modelos de software pueden ser descriptos con el lenguaje estándar UML [20] enriquecidos con expresiones OCL [13].

Para lograr interoperabilidad, los desarrollos centrados en modelos (Model Driven Development - MDD) representan a todos los artefactos generados en un lenguaje de metamodelado común. En MDA, los metamodelos se describen usando Meta Object Facility (MOF) [12] que captura la diversidad de estándares de modelado para integrar diferentes tipos de modelos y metadatos e intercambiarlos entre diferentes herramientas.

El desarrollo *model-driven* con MDA se lleva a cabo a través de una secuencia de transformaciones basadas en refinamientos y refactorings para soportar la evolución de los modelos. Un refinamiento es el proceso de construir una especificación más detallada a partir de otra más abstracta. Un refactoring es el proceso de transformar un modelo en otro en el mismo nivel de abstracción, sin cambiar su comportamiento observable con el fin de mejorar aspectos no funcionales del mismo tales como simplicidad, flexibilidad, adaptabilidad, facilidad en su uso, reusabilidad y portabilidad. La aplicación de refactorings es importante para mejorar la calidad de los procesos que requieren transformaciones de modelos que pueden ser usados tanto en procesos de ingeniería directa (forward engineering) como inversa (reverse engineering).

En este trabajo se propone una técnica de metamodelado alineada con MDA para la definición de refactorings de modelos, expresados como contratos OCL entre metamodelos. En [4] se muestra la especificación de refactorings sobre modelos independientes a una plataforma de implementación; en este artículo se presenta una clasificación de refactorings en los distintos niveles de abstracción de MDA y se describen algunos de los resultados presentados en [15].

Este artículo está organizado de la siguiente manera. En la sección 2 se presenta el framework arquitectural para refactorings basados en MDA. En la sección 3 se describe la técnica de especificación de refactorings basada en metamodelos. La sección 4 muestra una clasificación de refactorings aplicables a los distintos modelos propuestos por MDA. La sección 5 detalla trabajos relacionados en el área y en la sección 6 se presentan las conclusiones y futuros trabajos.

2 REFACTORING EN MDA

Se propone para el refactoring de modelos un framework arquitectural basado en MDA que distingue tres niveles diferentes de abstracción vinculados a modelos, metamodelos y especificaciones formales.

En el nivel de modelos, se presenta un conjunto de refactorings de diagramas de clases UML/OCL, para los modelos de software propuestos por MDA. Estos refactorings se basan en un conjunto de reglas que permiten transformar un modelo mejorando ciertos factores de calidad sin cambiar su comportamiento observable.

En el nivel de metamodelos, los refactorings aplicables a nivel de modelos son descriptos utilizando metamodelos MOF. Para esto, se especifican metamodelos origen y destino para cada refactoring. El metamodelo origen define una familia de modelos a los cuales puede aplicarse el refactoring y el metamodelo destino caracteriza a los modelos generados. Del mismo modo, se especifica cada refactoring como contrato OCL estableciendo precondiciones y postcondiciones para la aplicación de la transformación.

En el nivel de formalización, se describe una integración de técnicas formales con los metamodelos MOF del nivel anterior para asegurar la consistencia y validez de los refactorings basados en MDA. Se propone formalizar los metamodelos y los refactorings usando el lenguaje de metamodelado algebraico NEREUS el cual permite la integración con otros lenguajes formales [2][3].

Los refactorings aplicados a los distintos modelos de MDA mejoran la calidad de los mismos e incrementan su portabilidad entre distintas plataformas. De aquí la importancia de reestructurar los modelos ya que son el punto de partida en la secuencia de transformaciones. Además, fomenta la productividad del diseñador al automatizar las transformaciones de modelos. La especificación de los refactorings de manera uniforme para los distintos niveles de abstracción propuestos por MDA favorece la interoperabilidad entre las distintas plataformas y la integración con especificaciones formales permiten definirlos de manera más precisa y consistente permitiendo razonar sobre los refactorings.

La figura 1 muestra el framework arquitectural propuesto para refactorings basados en MDA. En este artículo se pone el énfasis en la técnica de especificación de refactorings basada en metamodelos.

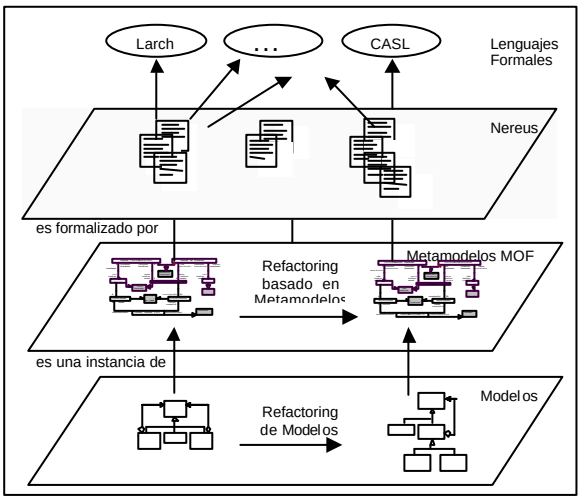


Figura 1: Refactorings basados en MDA

3 ESPECIFICACIÓN DE REFACTORINGS COMO CONTRATOS OCL

El refactoring de modelos transforma un modelo en otro funcionalmente equivalente al primero. Esta transformación se especifica utilizando la técnica de metamodelado. Un metamodelo, modelo que describe los modelos, consta de: metaclases, que describen los elementos que pueden existir en un modelo y relaciones entre metaclases que especifican las relaciones que pueden existir en un modelo.

Los metamodelos en MDA se especifican con MOF que captura la diversidad de estándares de modelado para integrar diferentes tipos de modelos y metadatos e intercambiarlos entre diferentes herramientas. Los metamodelos MOF se expresan como una combinación de diagramas de clases UML y especificaciones OCL. Para cada refactoring particular se especifican metamodelos MOF origen y destino como especializaciones del metamodelo UML (versión 2.1.1). El metamodelo origen define una familia de modelos a los cuales puede aplicarse la regla de transformación y el metamodelo destino caracteriza a los modelos generados.

El estándar propuesto por OMG para definir transformaciones es el Query, View, Transformation (QVT) [17] que permite realizar consultas sobre modelos, crear vistas sobre un modelo y escribir definiciones de transformaciones. En la presente propuesta, las reglas de transformación se expresan como contrato OCL con una notación sencilla. Las reglas establecen precondiciones y postcondiciones para la aplicación de la transformación, especificando la relación entre los elementos del metamodelo origen y los elementos del metamodelo destino. La notación propuesta utiliza el Essential OCL [13] que es un subconjunto mínimo de OCL y está alineada con el estándar QVT, en particular con el package CORE de QVT que depende de Essential MOF [12] y Essential OCL.

4 CLASIFICACIÓN DE REFACTORINGS EN EL CONTEXTO MDA

A continuación se presenta una clasificación de refactorings de diagramas estáticos UML para los distintos modelos de diseño propuestos por MDA. Se proponen refactorings aplicables a modelos independientes de una plataforma de implementación, a modelos dependientes de una plataforma y a modelos dependientes de la implementación.

La clasificación de los refactorings en los distintos niveles se basa en el análisis de los elementos del modelo que involucra cada transformación. Un refactoring puede agregar, modificar o eliminar elementos en un modelo, si dichos elementos pueden estar presentes en un nivel de abstracción dado, el refactoring se clasificará en dicho nivel.

Para la definición de reglas de transformación en los distintos niveles de MDA se especifican metamodelos correspondientes a cada nivel, ya que estas reglas imponen relaciones entre un metamodelo origen y un metamodelo destino. Los metamodelos son modelos que especifican la estructura, semántica y restricciones de una familia de modelos. Los metamodelos a nivel PIM describen una familia de modelos independientes de la plataforma. Los metamodelos a nivel PSM definen una familia de modelos dependientes de una plataforma particular. Los metamodelos a nivel ISM definen una familia de modelos dependientes de una plataforma y lenguajes de programación orientados a objetos particulares. En particular, la propuesta se ejemplificará sobre modelos específicos a la plataforma Java.

4.1 Refactorings a Nivel PIM

A continuación se presenta el refactoring *Extract Composite* tomado de [9], que se aplica cuando en una jerarquía de herencia existen subclases que almacenan colecciones de hijos los cuales son clases en la misma jerarquía y además poseen métodos para el tratamiento de dicha colección (figura 2). La aplicación de esta regla permite crear una superclase *Composite* y mover campos y métodos duplicados desde las subclases a la superclase, eliminando la duplicación de métodos al factorizar en una superclase el comportamiento común.

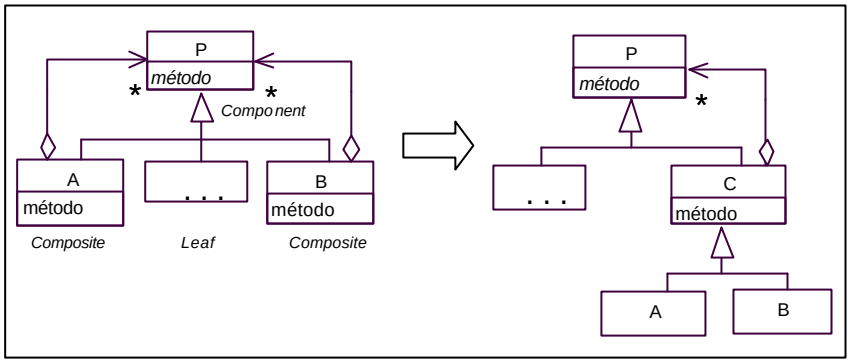


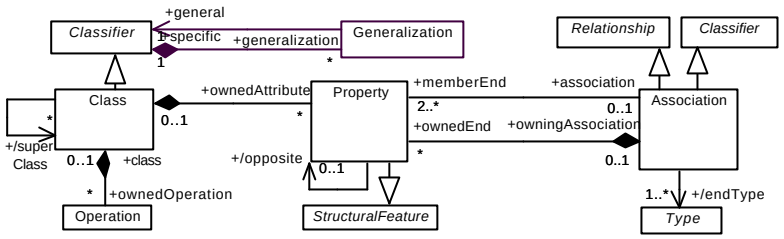
Figura 2: Refactoring *Extract Composite*

Para especificar este refactoring a nivel PIM se describen los metamodelos origen y destino como especializaciones del metamodelo UML complementados con restricciones OCL. El metamodelo UML de Diagramas de Clases (figura 3.1) describe los conceptos de modelado básicos de los diagramas de clases UML y gira alrededor de la metaclase *Class* que describe las características que debe poseer una clase en este diagrama. En particular se centra en sus propiedades y operaciones, como así también en las diferentes relaciones que se pueden establecer entre las clases: generalizaciones y asociaciones.

El metamodelo origen (figura 3.2) sugiere que una instancia de *Component* tiene dos o más instancias de *ComponentCompositeGeneralization* (*compositeSpecialization*), es decir, dos o más relaciones de generalización donde cada relación de generalización tendrá como clase hija una instancia de *Composite.* Por otro lado, una instancia de *Component* tendrá dos o más extremos de asociación (*associationEnd*) donde cada uno está vinculado a una asociación del tipo *CompositeComponentAssoc* donde el otro extremo de asociación tiene como clase participante a una instancia de *Composite.* Una instancia de *Component* posee una o más instancias de *ComponentLeafGeneralization,* es decir, una o más relaciones de generalización donde cada relación tendrá como clase hija una instancia de *Leaf.* Las metaclases sombreadas en gris corresponden al metamodelo UML.

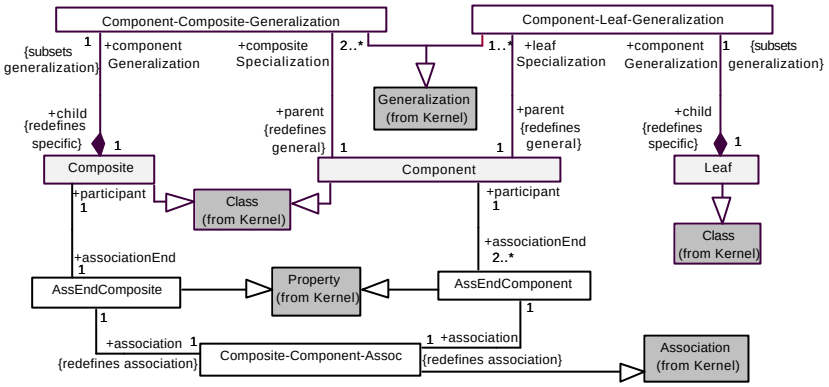
El metamodelo destino (figura 3.3) sugiere que en un modelo resultante de aplicar este refactoring, una instancia de *Component* tiene exactamente una instancia de *ComponentCompositeGeneralization* y un único extremo de asociación de tipo *AssEndComponent*.

La definición de la regla de refactoring se expresa como una transformación basada en los metamodelos origen y destino, la cual describe un mecanismo para convertir los elementos de un modelo origen, instancia del metamodelo origen, en elementos de otro modelo, instancia del metamodelo destino correspondiente. La aplicación de la regla *Extract Composite* crea en el modelo destino una superclase (*Composite*), detecta operaciones y atributos equivalentes y los mueve de las subclases a la superclase. La figura 4 muestra la especificación de la regla de transformación como contrato OCL.



Operation::isEquivalentTo (op:Operation): Boolean
isEquivalentTo (op)=
-- chequeo de las signaturas y de la especificación del comportamiento de las funciones
...
Property::isEquivalentTo (p:Property):Boolean
isEquivalentTo(p) = ...

1 – Metamodelo UML simplificado

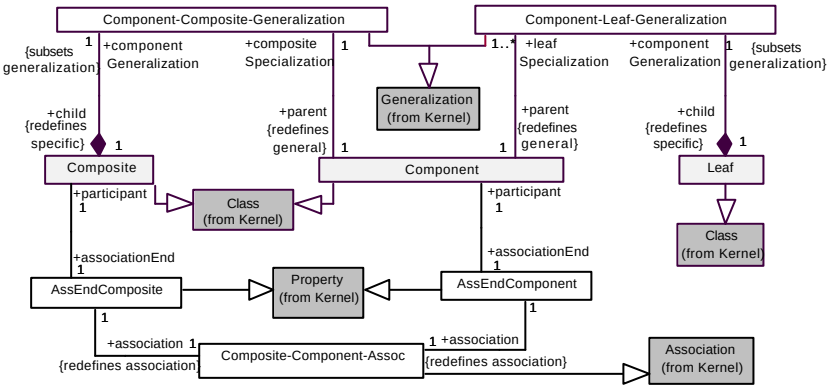


context AssEndComposite **inv**:
self.aggregation = #shared or
self.aggregation = #composite

context Component **inv**:
-- Los extremos de asociación de tipo *AssEndComponent*
-- son equivalentes.
self.associationEnd →
forAll (a1, a2 | a1 = a2 or a1.isEquivalentTo(a2))

context Component **inv**:
-- Para cada clase de tipo Composite existe una operación que
self.compositeSpecialization.child → forAll (class |
class.ownedOperation → exists (op |
-- es equivalente a operaciones de las otras clases Composite
self. compositeSpecialization.child → excluding (class) →
forAll (c | c.ownedOperation → exists (o |
op.isEquivalentTo(o))))) ...

2 – Metamodelo origen del refactoring *Extract Composite*



context AssEndComposite **inv**:
self.aggregation = #shared or self.aggregation = #composite
...

3 – Metamodelo destino del refactoring *Extract Composite*

Figura 3**:** Metamodelos MOF para el refactoring *Extract Composite*

```
Transformation Extract Composite {
parameters
    source: Metamodelo Origen Extraer Composite:: Package
    target:  Metamodelo Destino Extraer Composite:: Package

postconditions
    post:
    -- Para toda clase sourceClass, instancia de Component, del paquete source,
    source.ownedMember → select(oclIsTypeOf(Component)) → forAll ( sourceClass |
        -- existe una clase targetClass, instancia de Component, del paquete target tal que,
        target.ownedMember → select(oclIsTypeOf(Component)) → exists ( targetClass |

            -- targetClass tiene una sola relación del tipo Component-Composite-Generalization,
            targetClass.oclAsType(Component).compositeSpecialization → size() =1  and

            -- targetClass tiene un sólo extremo de asociación que la asociacia con una clase Composite,
            targetClass.oclAsType(Component).associationEnd → size() =1  and

            -- targetClass y sourceClass tienen las mismas clases Leaf,
                targetClass.oclAsType(Component).leafSpecialization.child =
            sourceClass.oclAsType(Component).leafSpecialization.child and  ....

    post:
    -- Para toda clase sourceClass, instancia de Composite, del paquete source,
    source.ownedMember → select(oclIsTypeOf(Composite)) → forAll ( sourceClass |
        -- existe una clase targetClass, instancia de Class, del paquete target tal que,
        target.ownedMember → select(oclIsTypeOf(Class)) → exists ( targetClass |

            -- una clase de tipo Composite es superclase de targetClass,
            targetClass.oclAsType(Class).superClass.oclIsTypeOf(Composite)  and

            -- targetClass y sourceClass tienen el mismo nombre,
            targetClass.name = sourceClass.name  and

            -- Por cada operación equivalente de las clases  Composite del paquete source
            sourceClass.oclAsType(Composite).ownedOperation → forAll( op  |
                (source.ownedMember → select(oclIsTypeOf(Composite))
                →excluding(sourceClass)) → collect (oclAsType(Composite).ownedOperation)
                    → forAll ( o |
                    if o.isEquivlalentTo(op)  then
                    -- en el paquete target habrá una operación equivalente en la  superclase de targetClass,
                        targetClass.oclAsType(Class).superClass.ownedOperation → exists
                            (  targetOp |        op.isEquivalentTo (targetOp)   )  and
                        targetClass.oclAsType(Class).ownedOperation → excludes(op)
                    else -- de otro modo, la operación será de targetClass.
                        targetClass.oclAsType(Class).ownedOperation → includes(op)
                    endif ))
                            ……..
    post:
    -- Por cada clase cliente de la clase Composite en el paquete source,
    -- existe una clase cliente de la nueva clase Composite en el paquete target,
    ..........................
}
```

Figura 4: Refactoring *Extract Composite* en OCL

4.2 Refactorings a Nivel PSM

El refactoring que se detalla a continuación sobre modelos específicos a la plataforma Java está basado en el refactoring propuesto en [7]. Cuando en dos clases de un modelo existen operaciones comunes pueden realizarse dos tipos de refactorings: extraer el comportamiento común a una superclase o a una interfaz. En los lenguajes orientados a objetos que soportan herencia múltiple, se podría pensar en extraer una superclase con las interfaces y código comunes. Java tiene herencia simple, limitando a las clases a tener una única superclase, sin embargo, se puede implementar este refactoring usando interfaces, con la diferencia que sólo puede extraerse las operaciones comunes y no el código común. Este refactoring refleja una práctica ventajosa y fomentada en la comunidad de diseñadores bajo la plataforma Java. La figura 5 ejemplifica este refactoring.

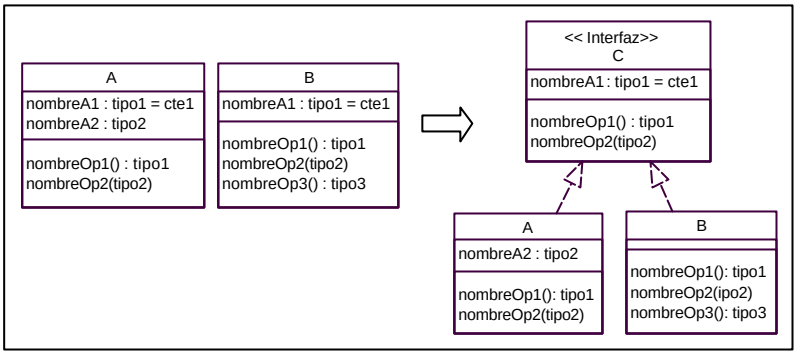


Figura 5: Refactoring *Extract Interface*

Los metamodelos origen y destino correspondientes a este refactoring se especifican como especializaciones del metamodelo JAVA. Este último, especificado como especialización del metamodelo UML, refleja la presencia de clases e interfaces Java en los modelos de esta plataforma (figura 6.1). Una clase contiene atributos, extremos de asociación y operaciones, puede heredar de una única clase base pero puede implementar varias interfaces. Una interfaz puede contener atributos estáticos y finales, métodos sin implementación y extremos de asociación y además puede heredar de varias interfaces.

El metamodelo origen (figura 6.2) sugiere que las instancias de *A* y de *B* pueden tener operaciones, instancias de *A_Operation* y *B_Operation* respectivamente, entre las cuales existe una relación de equivalencia sintáctica (instancia de *Operation_Relationship*). Las instancias de *A* y *B* también pueden tener atributos equivalentes. Las metaclases sombreadas en gris claro corresponden al metamodelo PSM-Java, y en gris oscuro corresponden al metamodelo UML. El metamodelo destino (figura 6.3) muestra que las clases *A* y *B* implementan una interfaz (instancia de *AnInterface*) la cual tendrá las operaciones y atributos equivalentes.

La regla de transformación se aplica sobre un modelo con dos clases que tienen un subconjunto de operaciones comunes a través de los siguientes pasos: crear una interfaz Java, declarar en la interfaz las operaciones comunes y los atributos equivalentes, declarar que las clases originales implementan dicha interfaz y adaptar las declaraciones de tipo de los clientes para usar la interfaz.

4.3 Refactorings a Nivel ISM

Para la aplicación de refactorings a nivel de modelos específicos a la implementación Java, se necesita identificar elementos que están presentes en el nivel de código. En el ejemplo que se detalla a continuación se necesita analizar si los métodos de una clase poseen código nulo, es decir, si la implementación del método carece de sentencias. Por lo general, una clase concreta en la plataforma Java tiene métodos con código nulo cuando necesita satisfacer un contrato implementando una interfaz pero realmente sólo necesita código para alguno de los métodos de la interfaz. Por lo general, una clase concreta en la plataforma Java tiene métodos con código nulo cuando necesita satisfacer un contrato implementando una interfaz pero realmente sólo necesita código para alguno de los métodos de la interfaz. Los métodos con código nulo están en la clase para satisfacer una regla del compilador Java: las declaraciones de todos los métodos miembros de cada superinterfaz directa de una clase deben ser implementados (a menos que la clase sea declarada abstracta). Los métodos con código nulo forman parte de la interfaz de la clase (sus métodos públicos) pero es una falsa descripción de su comportamiento.

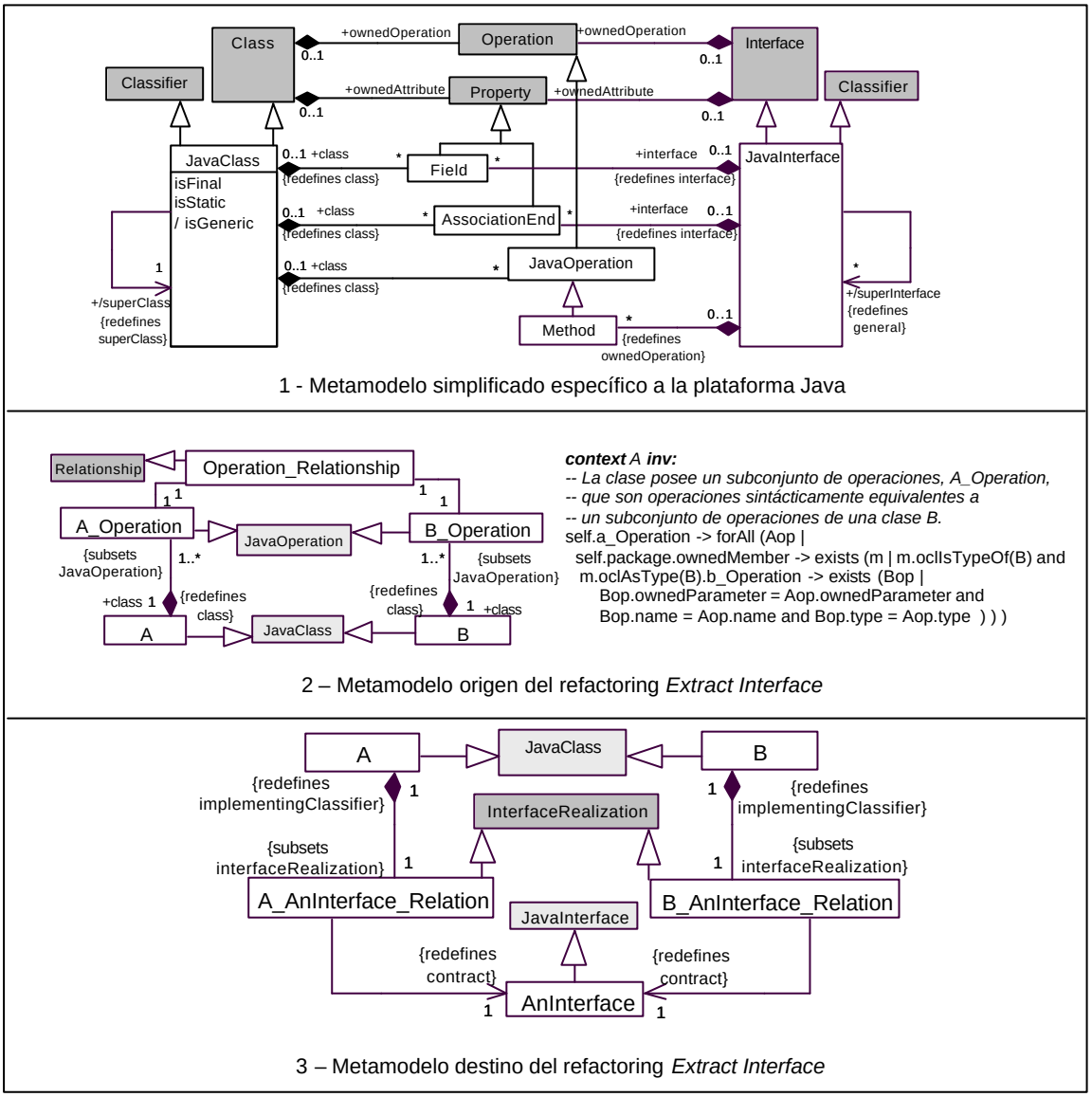


Figura 6: Metamodelos MOF para el refactoring *Extract Interface*

Cuando en un modelo una clase implementa una interfaz pero sólo provee código para algunos de los métodos, la aplicación del refactoring *Adapt Interface* [9] permite reestructurar este tipo de código a través del patrón *Adapter*. Se dispone de una clase *Adapter* que implementa la interfaz y que posee métodos con código nulo para cada uno de los métodos definidos en la interfaz y una subclase del *Adapter* que provea el código necesario. Este refactoring puede ser usado para adaptar múltiples interfaces particionando métodos en cada uno de sus respectivos *adapters* (figura 7). En Java, no se necesita declarar formalmente una subclase de *Adapter*, basta con crear una clase anidada anónima *Adapter* y proveer una referencia a la misma a través de un método de creación.

El metamodelo específico a la implementación en código Java, obtenido por especialización del metamodelo UML, refleja que un paquete Java contiene archivos, clases e interfaces. Una clase contiene atributos y métodos, puede heredar de una única clase base pero puede implementar varias interfaces. Las operaciones de una clase pueden tener una implementación que consta de un bloque de una o varias sentencias. Una interfaz puede contener atributos estáticos y finales, métodos sin implementación y puede heredar de varias interfaces. La figura 8.1 muestra el diagrama de operaciones de este metamodelo.

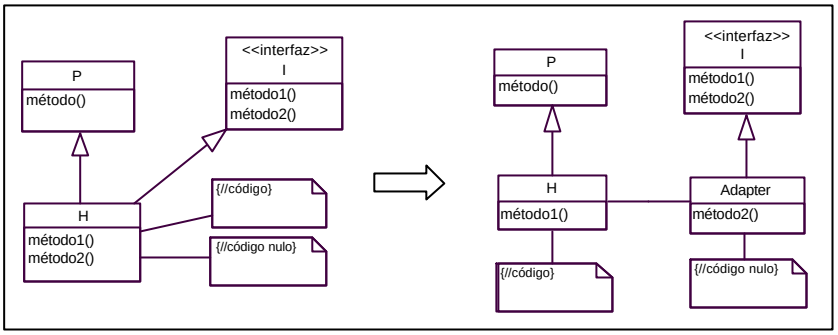


Figura 7: Refactoring *Adapt Interface*

Este metamodelo Java, a su vez, es especializado para obtener los metamodelos origen y destino. El metamodelo origen (figura 8.2) sugiere que una instancia de *AClass* tiene una o más instancias de *Class-Interface-Relation* donde cada una de éstas tiene asociada una única instancia de *AnInterface*, es decir, una instancia de *AClass* implementa por lo menos una interfaz. La interfaz implementada tiene por lo menos dos métodos. La instancia de *AClass* tiene dos o más operaciones Java, donde un subconjunto de ellas son operaciones implementadas con código y otro subconjunto de operaciones implementadas con código nulo. El metamodelo destino expresa que existirá una clase de tipo *superAdapter* (una clase que implementa una interfaz) que tendrá las operaciones con código nulo y *AClass* tendrá una clase anidada anónima *AnAdapter* (que extiende *superAdapter*) con las operaciones con código (figura 8.3). En ambos metamodelos las metaclases sombreadas en gris claro corresponden al metamodelo ISM-Java y en gris oscuro corresponde al metamodelo UML.

La regla de transformación se aplica sobre un modelo que contiene una clase que implementa una interfaz pero sólo provee código para algunos de los métodos de la interfaz. La transformación consistirá de los siguientes pasos: usar un *Adapter* que implemente la interfaz y provea el comportamiento nulo, crear en la clase un método de creación que retorne una referencia a una instancia que extiende al *Adapter*, borrar de la clase los métodos con comportamiento nulo, mover los métodos implementados a la nueva instancia del *Adapter*, eliminar la relación de implementación de interfaz de la clase, proveer la instancia del *Adapter* a los clientes que lo necesiten. Estos cambios serán especificados como contrato OCL.

5 TRABAJOS RELACIONADOS

La primera publicación relevante sobre refactoring fue realizada por Opdyke, mostrando cómo las funciones y atributos pueden migrar entre clases, estableciendo las precondiciones que deben ser mantenidas para asegurar que la transformación preserva el comportamiento del programa [14]. Roberts completó este trabajo describiendo técnicas basadas en contratos de refactorings [18]. Fowler describió principios y prácticas sobre refactorings aplicables sobre código fuente Java [7].

Una amplia descripción de las investigaciones existentes en el área de refactoring se describe en [11]. Existen trabajos sobre refactorings de modelos UML [19] [21]. Algunos autores definen e implementan refactorings de modelos en base a transformaciones basadas en reglas [16]; otros autores presentan una técnica para describir transformaciones a nivel de metamodelo basada en patrones de transformación [8]. Folli y Mens proponen utilizar transformaciones de grafos como base para la especificación de refactoring de modelos [6].

Las herramientas CASE basadas en MDA existentes proveen limitadas facilidades para el refactoring sobre código fuente a través de una selección explícita por parte del diseñador [1].

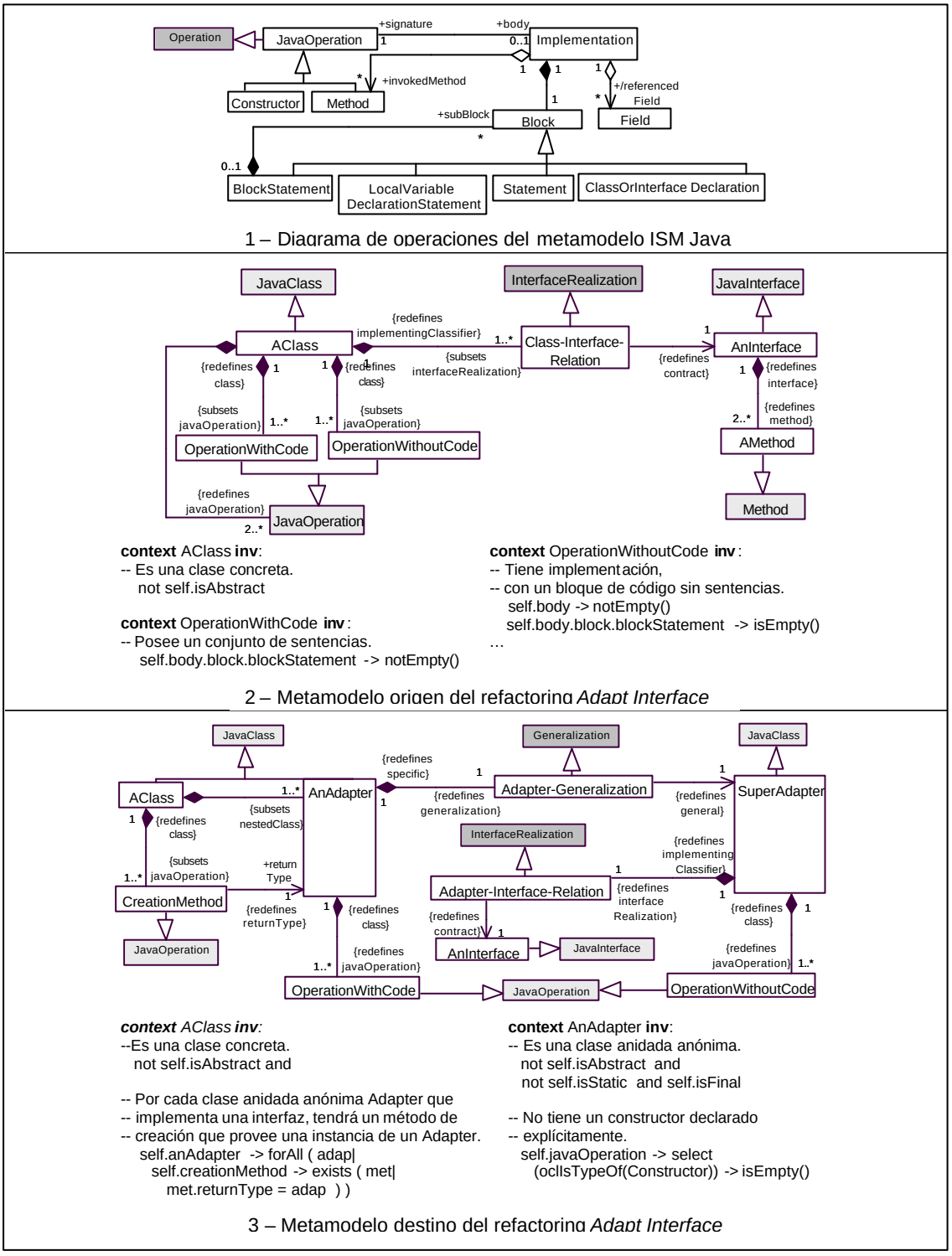


Figura 8: Metamodelos MOF para el refactoring *Adapt Interface*

6 CONCLUSIONES Y FUTUROS TRABAJOS

Nuestra principal contribución radica en la especificación de refactorings alineados con MDA aplicables a los distintos modelos propuestos por esta arquitectura. La descripción de los refactorings consiste, por un lado, en la especificación de los metamodelos origen y destino para cada refactoring en particular y por otro lado, en la especificación de la regla de transformación

como contrato OCL entre metamodelos. El nivel de metamodelos es necesario para controlar la consistencia de los refactorings y establecer los elementos que deben estar presentes en los modelos fuente y destino y sus restricciones. La especificación de los refactorings utiliza estándares propuestos por OMG. Los metamodelos se describen utilizando diagramas de clases UML y restricciones OCL y las reglas de transformación utilizan un subconjunto mínimo de OCL. Esta especificación, alineada con MOF, facilita la interoperabilidad entre las distintas plataformas.

En [5] se muestra la integración de la especificación de los refactorings con técnicas formales, traduciendo los metamodelos MOF y la regla de transformación al lenguaje NEREUS.

Como futuras extensiones a este trabajo se propone analizar refactorings sobre otros tipos de modelos UML, como así también analizar el impacto que las transformaciones aplicadas sobre un tipo de modelo producen sobre otras vistas. Como limitación a esta propuesta cabe mencionar la ausencia de una herramienta que la soporte.

REFERENCIAS

[1] CASE TOOLS. 2008. Disponible en: www.objectsbydesign.com/tools/umltools_byCompany.html

[2] Favre, L. "A Rigorous Framework for Model Driven Development". Siau, K. (ed.). Advanced Topics in Database Research, Vol. 5. Cap. I, IGP, USA. 2006. Páginas: 1-27.

[3] Favre, L. "Foundations for MDA-based Forward Engineering". Journal of Object Technology (JOT) 4. 2005. Páginas: 129-153.

[4] Favre, L y Pereira, C. "Improving MDA-based Process Quality through Refactoring Patterns". Proceedings of the 1st International Workshop on Software Patterns and Quality (SPAQu'07). Nagoya, Japón. 2007. ISBN 978-4-915256-69-1 C3040. Páginas 17-22.

[5] Favre, L y Pereira, C. "Formalizing MDA-based Refactoring". Proceedings of the 19th Australian Conference on Software Engineering (ASWEC 2008). Perth, Australia. IEEE Conference Publishing Services Editors. 2008. ISBN-10:07695-3100-8. Páginas: 377-386.

[6] Folli, A y Mens, T. "Refactoring of UML models using AGG". Proceedings of the Third International ERCIM Workshop on Software Evolution. Francia. 2007.

[7] Fowler, M. "Refactoring: Improving the Design of Existing Programs". Addison-Wesley, 1999.

[8] Judson, S., Carver, D. y France, R. "A Metamodeling Approach to Model Refactoring". OOPSLA. California. USA. 2003.

[9] Kerievsky, J. "Refactoring to Patterns". Addison-Wesley. 2004.

[10] MDA. The Model Driven Architecture. 2008. Disponible en: www.omg.org/mda.

[11] Mens, T.; Demeyer, S.; Du Bois, B.; Stenten, H. y Van Gorp, P. "Refactoring: Current Reasearch and Future Trends". Proceedings of Third Workshop on Language Descriptions, Tools and Applications (LDTA 2003) 2003. Páginas: 120–130.

[12] MOF. Meta Object Facility. Documento: formal/2006-01-01. Disponible en: www.omg-org/mof.

[13] OCL. Object Constraint Language. Documento: formal/06-05-01. Disponible en: www.omg.org

[14] Opdyke, W. "Refactoring Object-Oriented Frameworks". Tesis Doctoral. University of Illinois. Urbana-Champaign. 1992.

[15] Pereira, C. "Formalización de Refactorings en el contexto MDA". Tesis de Magíster en Ingeniería de Software. Facultad de Informática, Universidad Nacional de La Plata, Argentina. 2008.

[16] Porres, I. "Model Refactorings as Rule-Based Update Transformations". Lecture Notes in Computer Science, Vol. 2863, Springer Verlag. 2003. Páginas: 159-174.

[17] QVT. Query/Views/Transformation Specification. Documento:ptc/07/07/07. Disponible:www.omg.org

[18] Roberts, D. "Practical Analysis for Refactoring". Tesis Doctoral. University of Illinois. 1999.

[19] Sunyé, G.; Pollet, D.; LeTraon, D. y Jézéquel, J. "Refactoring UML Models". Lecture Notes in Computer Science, Vol. 2185, Springer-Verlag. 2001. Páginas: 134-138.

[20] UML. Unified Modeling Language Superstructure. 2008. Disponible en: www.omg.org.

[21] Van Gorp, P.; Stenten, H.; Mens, T. y Demeyer, S. "Towards automating source-consistent UML refactorings". Lecture Notes in Computer Science, Vol. 2863, Springer Verlag. 2003, Páginas: 144-158.