Formalización de refactorings en el contexto de MDA

Claudia Teresa Pereira

Directora: **Lic. Liliana María Favre**
Codirector: **Dr. Gustavo Rossi**

Tesis presentada para obtener el grado de Magíster en Ingeniería de Software.

Facultad de Informática

Universidad Nacional de La Plata

2008

ÍNDICE

CAPÍTULO 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. El término surgió en el contexto de transformaciones de diseños de código orientado a objetos que consideraban factores de calidad no funcionales como simplicidad, mantenimiento, extensibilidad, modularidad, reusabilidad, complejidad y eficiencia. La aplicación de esta técnica se extendió a procesos de desarrollo ágiles como XP (eXtreme Programming) y a procesos de reingeniería de software que requieren una visión más general de la evolución de todos los artefactos generados a lo largo de dichos procesos, aún aquellos más abstractos que el código fuente.

Actualmente, refactoring es una técnica relevante en los procesos de desarrollo basados en la arquitectura Model Driven (Model Driven Architecture - MDA) (MDA, 2007). MDA es una iniciativa propuesta por Object Management Group (OMG, 2007) 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.

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 (UML-Superstructure, 2007; UML-Infrastructure, 2007) enriquecidos con expresiones OCL (OCL, 2006).

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, 2006) 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, que incluye: construir un CIM, transformar un CIM en un PIM, transformar un PIM en uno o más PSMs, y a partir de ellos generar componentes ejecutables y aplicaciones. Las transformaciones se especifican utilizando Query, View, Transformation (QVT, 2007) que permite realizar consultas sobre modelos, crear vistas sobre un modelo y escribir definiciones de transformaciones.

La técnica de refactoring se adecua a los distintos procesos de desarrollo permitiendo la evolución de modelos. En los procesos de ingeniería forward se transforman modelos

abstractos en modelos concretos, es decir, de CIM a ISM. En los procesos de ingeniería reversa se analiza un sistema para identificar sus componentes y sus interrelaciones y se crean modelos en un mayor nivel de abstracción, es decir, las transformaciones van desde ISM a CIM. En los procesos de reingeniería, cuyo objetivo es reestructurar software *legacy* o migrar código a un lenguaje de programación, plataforma o paradigma diferente, se transforma una representación de bajo nivel en otra, mientras se construyen modelos de mayor nivel de abstracción a lo largo del proceso.

Los lenguajes UML y OCL, y en particular, los metamodelos MOF, son expresivos y flexibles, lo cual se refleja en el aumento en la productividad del diseñador, pero son imprecisos y ambiguos para simular, verificar y validar propiedades del sistema o para generar modelos o implementaciones a partir de transformaciones. Por otra parte, los lenguajes formales permiten producir especificaciones de software precisas y analizables y automatizar las transformaciones de modelo a modelo, sin embargo, requieren cierta familiaridad con la notación formal que la mayoría de los diseñadores no poseen. Por lo tanto, ambos tipos de lenguajes se complementan y su integración permitiría aprovechar las ventajas que cada uno ofrece.

Por lo anteriormente expuesto, en esta tesis se propone una técnica de metamodelado, alineada con MOF y MDA, para la definición de refactorings de modelos en diferentes niveles de abstracción (PIM, PSM, ISM). Además, se propone un método para construir una especificación formal de refactorings usando el lenguaje de metamodelado NEREUS (Favre, 2005).

En la próxima sección se describe el objetivo y la descripción general de esta propuesta.

1.1 Propuesta de tesis

1.1.1 Objetivo

El objetivo principal de esta tesis es presentar una técnica de especificación de refactorings alineados con MDA aplicables a sus diferentes tipos de modelos de diseño UML, es decir, a modelos independientes de la computación, modelos dependientes de una plataforma y modelos dependientes de la implementación. En particular, en este trabajo, se muestra la técnica de especificación de refactorings aplicados a modelos de diagramas de clases UML enriquecidos con anotaciones OCL en los distintos niveles de abstracción.

Los refactorings se especifican como contratos OCL utilizando la técnica de metamodelado. Los metamodelos son expresados usando Meta Object Facility (MOF).

Asimismo, en esta tesis, se analiza la integración de la especificación de los refactorings con técnicas formales. Se detalla la traducción de los contratos OCL de refactorings y los metamodelos MOF al lenguaje de especificación formal NEREUS.

1.1.2 Descripción general

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 fuente y destino para cada refactoring. El metamodelo fuente 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. La figura 1.1 muestra esta propuesta.

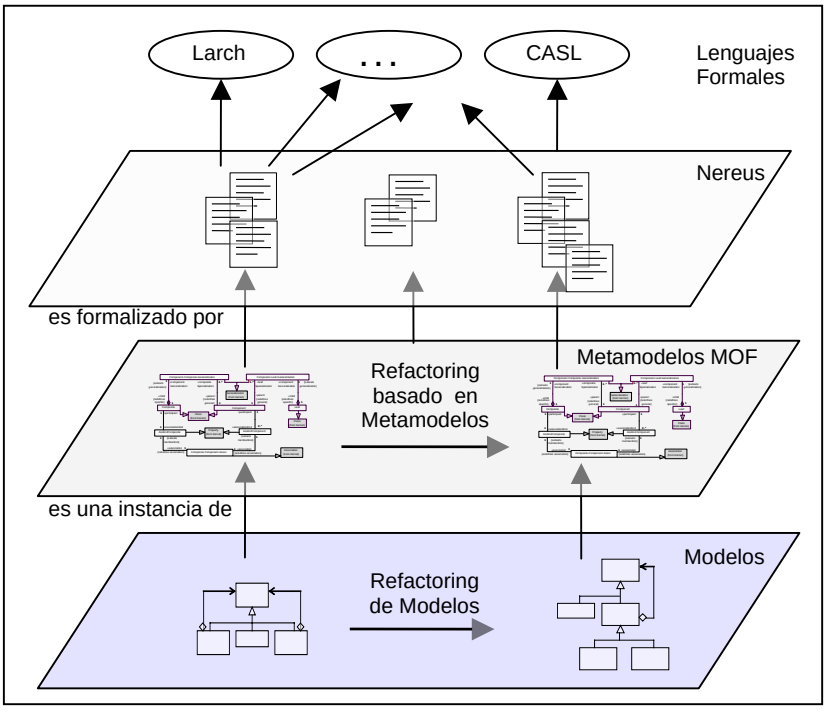


Figura 1.1 - Refactorings basados en MDA

A continuación se detallan cada uno de los niveles de abstracción.

- Nivel de modelos:

  El desarrollo basado en modelos (MDD) propone el uso de modelos para direccionar el curso de cada etapa en el proceso de desarrollo de sistemas. MDA, realización de MDD propuesta por el OMG, define un framework que separa la especificación de la funcionalidad del sistema de su implementación sobre una plataforma específica. Distingue al menos tres tipos diferentes de modelos de diseño:

  - modelos independientes de la plataforma (Platform Independent Model - PIM),
  - modelos específicos a la plataforma (Platform Specific Model - PSM),
  - modelos específicos a la implementación (Implementation Specific Model - ISM).

MDA distingue dos tipos de transformaciones para soportar la evolución de modelos: refinamientos y refactorings. Un refinamiento es el proceso de construir una especificación más detallada que conforma otra más abstracta. Refactoring significa realizar cambios en un modelo sin cambiar su comportamiento pero mejorando factores de calidad no funcionales tales como simplicidad, flexibilidad, legibilidad y performance. La figura 1.2 exhibe los refactorings dentro del framework básico de la MDA.

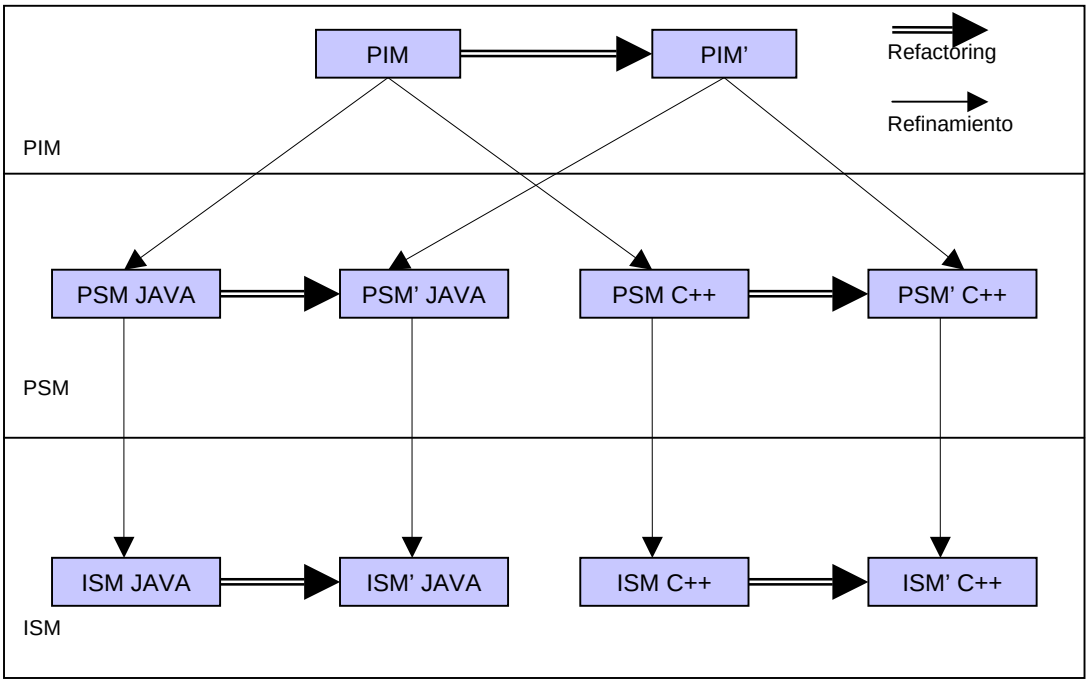


Figura 1.2 – Transformaciones de modelos en MDA

En esta tesis se define un conjunto de refactorings aplicables a diagramas de clases UML con restricciones OCL para los diferentes tipos de modelos propuestos por MDA. Éstos están basados en reglas de transformación que preservan la funcionalidad de los modelos.

- Nivel de metamodelos:

  En MDD todos los artefactos generados durante el desarrollo de software son representados usando un lenguaje de metamodelado común. En MDA, los metamodelos son expresados usando MOF, un meta-metamodelo que define una manera común de capturar toda la diversidad de estándares de modelado y construcciones de intercambio. Los metamodelos MOF se basan en las nociones de clases, asociaciones, tipos de datos y paquetes.

  Los refactorings se especifican a través de metamodelos fuente y destino, representados por metamodelos MOF. El metamodelo fuente describe una familia de modelos a los cuales puede aplicarse el refactoring correspondiente. El metamodelo destino describe una familia de modelos que resultan de aplicar dicho refactoring. Los modelos, PIMs, PSMs e ISMs, a los cuales puede aplicarse un refactoring son instancias del

metamodelo fuente correspondiente. Los modelos resultantes de la transformación son instancias del metamodelo destino.

Los refactorings aplicables a nivel de modelos se definen a nivel de metamodelo, es decir, imponiendo relaciones entre el metamodelo fuente y el metamodelo destino. Estas relaciones se describen vinculando las metaclases de los elementos del modelo fuente con la/s metaclase/s de los elementos del modelo destino. Debido a la relación instancia-tipo entre los elementos del modelo y las metaclases del metamodelo, cada ocurrencia de la metaclase en los modelos debería conformar las restricciones establecidas para su metaclase. Estas restricciones o reglas de transformación constituirán la definición del refactoring a nivel de metamodelos. Estos refactorings se definen como contratos OCL entre metamodelos, consistiendo de precondiciones y postcondiciones que deben verificar los modelos antes y después de la transformación respectivamente.

Los metamodelos MOF controlan la consistencia de las transformaciones. El nivel de metamodelo es necesario para establecer los elementos que deben estar presentes en los modelos fuente y destino y sus restricciones.

La figura 1.3 muestra los modelos en sus diferentes niveles de abstracción y la relación con los correspondientes metamodelos, como así también, la relación entre el refactoring de modelos y su definición basada en metamodelos.

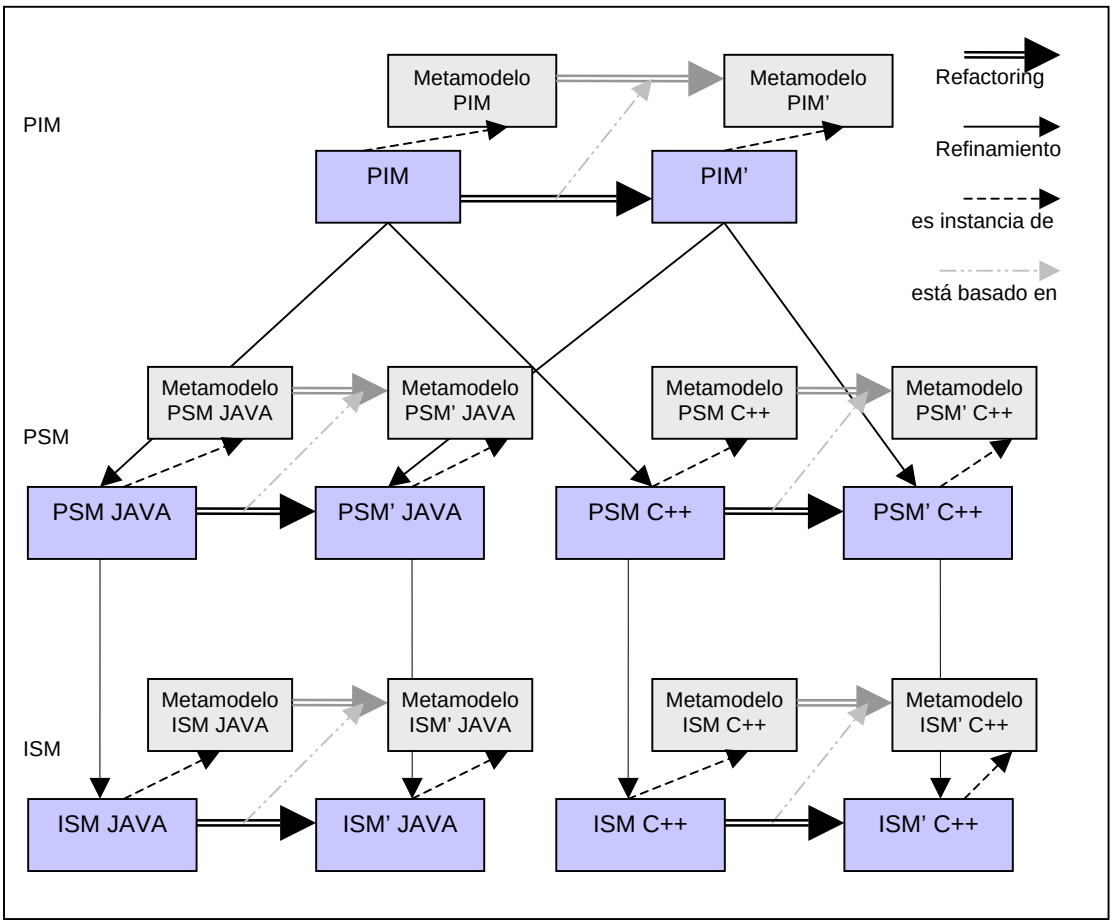


Figura 1.3 – Relación entre modelos y metamodelos

- Nivel de especificaciones formales:

  Los metamodelos MOF son expresados utilizando los lenguajes UML y OCL. Pero estos lenguajes son imprecisos y ambiguos cuando se trata de simular, verificar y validar propiedades de un sistema. A partir del surgimiento de UML y OCL, diferentes autores han discutido la ambigüedad e incompletitud de ciertos aspectos de estos lenguajes debido a que la semántica de cada uno de ellos no está definida con precisión. Algunos trabajos, (Hamie y otros, 1998; Mandel y Cengarle, 1999), muestran aspectos sobre OCL y otras investigaciones revelan imprecisiones semánticas de UML (Saksena y otros, 1998; Barbier y otros, 2001; Stevens, 2001).

  Como respuesta a las investigaciones anteriores, surgen varias propuestas que analizan aspectos de la semántica de OCL (Ritchers y Gogolla, 1998; Cengarle y Knapp, 2001; Bottoni y otros, 2000; Beckert y otros, 2002). Otras propuestas presentan formalizaciones de UML (o subconjuntos de UML) basadas en diferentes lenguajes de especificación existentes (Kim y Carrington, 1999, 2002; Padawitz, 2000; Fernández y Toval, 2001; Kuske y otros, 2002).

  En esta tesis, el nivel de especificaciones formales, vincula los metamodelos MOF y los refactorings basados en metamodelos con especificaciones formales como se muestra en la figura 1.4.

  Para definir los refactorings de manera más precisa y consistente se especifican formalmente utilizando el lenguaje NEREUS. Éste es un lenguaje alineado con MOF, es decir, la mayoría de los conceptos de los metamodelos MOF pueden ser traducidos a NEREUS. Además, este lenguaje puede ser visto como una notación intermedia abierta a otras especificaciones formales. Se definió, en particular, la traducción de construcciones NEREUS a CASL (Favre, 2006) lo que permitiría razonar sobre la consistencia de los refactorings utilizando las herramientas que provee este último, tales como prototipo de consistencia de especificaciones algebraicas, asistente para pruebas de teoremas Isabelle (COFI, 2008).

  NEREUS puede ser usado como un lenguaje de especificación intermedio y está conectado con diferentes lenguajes de programación semiformales y formales. Por lo tanto, al especificar en NEREUS los metamodelos MOF y los refactorings basados en metamodelos se facilita su conexión con diferentes lenguajes formales. Este paso extra provee ciertas ventajas. NEREUS elimina la necesidad de definir formalizaciones y transformaciones específicas para cada lenguaje formal diferente. Las especificaciones de metamodelos y transformaciones pueden ser reutilizadas en los distintos niveles de MDA y pueden relacionarse unos con otros porque están definidos de la misma manera a través de una sintaxis textual.

  Se define, en el nivel de especificación formal, un sistema de transformación que traduce automáticamente los metamodelos MOF y los refactorings basados en metamodelos a NEREUS. Esta propuesta provee una base rigurosa para MDD para desarrollar herramientas que, por un lado, toman la ventaja del poder de los lenguajes formales y, por otro lado, permite a los desarrolladores manipular directamente sus modelos UML/OCL, desconociendo el nivel de especificación formal que subyace a los modelos.

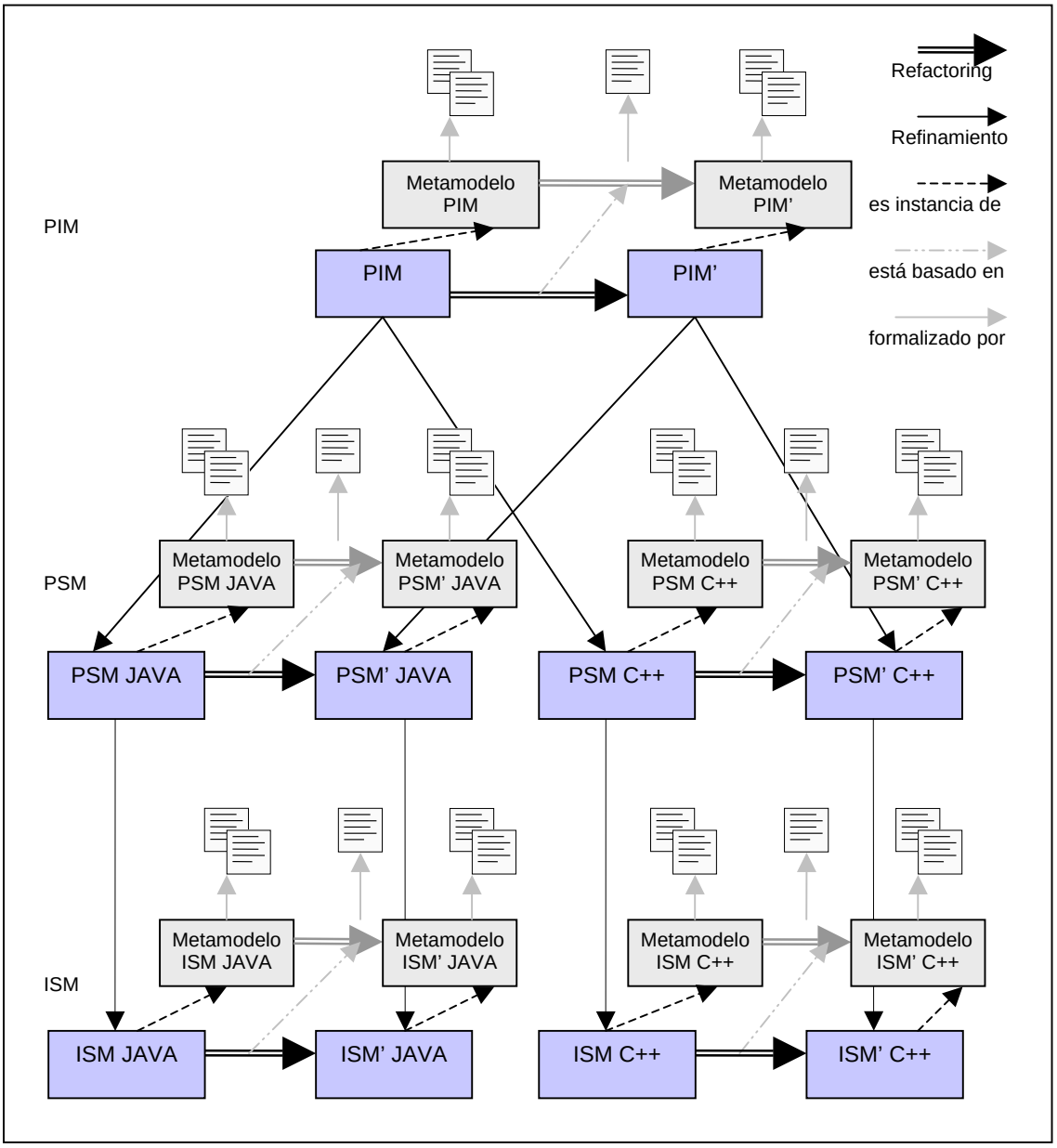


Figura 1.4 – Relación entre modelos, metamodelos y especificación

1.2 Publicaciones vinculadas a esta tesis

- "Foundations for MDA CASE Tools" Co-Autores: Favre Liliana, Martinez Liliana. Artículo aceptado para: Encyclopedia of Information Science and Technology, Second Edition, IGI Global, USA. A ser publicado en 2008.
- "Formalizing MDA-based Refactoring". Co-autor: Favre Liliana. Aceptado para su publicación en: Proceedings of the 19th Australian Conference on Software Engineering (ASWEC 2008). Perth, Australia. IEEE Conference Publishing Services Editors. ISBN-10:07695-3100-8. Páginas 377-386.

- "Improving MDA-based Process Quality through Refactoring Patterns". Co-autor: Favre Liliana. Publicado en: Proceedings of the 1st International Workshop on Software Patterns and Quality (SPAQu'07). Nagoya, Japón. Diciembre, 2007. ISBN 978-4-915256-69-1 C3040. Editor: Information Processing Society of Japan. Páginas 17-22.
- "Specifying Refactorings as Metamodel-based Transformation". Co-autor: Favre Liliana. Publicado en: Proceedings of 2006 Information Resources Management Association International Conference (IRMA 2006). Washington, D.C., USA. 21-24 de Mayo, 2006. ISBN 1-59904-019-0. Páginas 264-268.
- "Forward Engineering of UML Static Models". Co-Autores: Favre Liliana, Martinez Liliana. Artículo invitado en: Encyclopedia of Information Science and Technology, Volume I-III Mehdi Khosrow-Pour (Editor). Idea Group Publishing, USA. 2005. ISBN 1-59140-533-X. Páginas 1212-1217.
- "Refactoring de Diagramas de Clases UML". Co-autor: Liliana Favre. Publicado en: VI Workshop de Investigadores en Ciencias de la Computación (WICC 2004). Neuquén, Argentina. Mayo 2004. Páginas 225-229.
- "Refactoring UML Class Diagram". Co-autores: Favre Liliana, Martinez Liliana. Publicado en: Proceedings of 2004 Information Resources Management Association International Conference (IRMA 2004). New Orleans, USA. 23-26 de Mayo, 2004. ISBN 1-59140-261-1. Páginas 506-510.
- "Forward Engineering and UML: From UML Static Models to Eiffel Code". Co-autores: Favre Liliana, Martinez Liliana. Publicado en: UML and the Unified Process (Liliana Favre editor). Capítulo IX. IRM Press. USA. 2003. ISBN 1-931777-44-6. Páginas 199-217.
- "Forward Engineering and UML: From UML Static Models to Eiffel Code". Co-autores: Favre Liliana, Martinez Liliana. Publicado en: Proceedings of 2002 Information Resources Management Association (IRMA 2002). Seattle, USA. Mayo de 2002. ISBN 1-930708-39-4. Páginas 584-588.
- "Una integración de modelos estáticos UML y Eiffel". Co-autores: Favre Liliana, Martinez Liliana. Publicado en: Proceedings del VII Congreso Argentino de Ciencias de la Computación (CACIC 2001). Calafate, Argentina. Octubre de 2001. Páginas 521-530.
- "Transforming UML Static Models to Object Oriented Code". Co-autores: Favre Liliana, Martinez Liliana. Publicado en: Proceedings of Technology of Object Oriented Languages and Systems (TOOLS 37). Editorial: IEEE Computer Society. ISBN 0-7695-0918-5. TOOLS-37/PACIFIC 2000. Sydney, Australia. Noviembre de 2000. Páginas 170-181.

1.3 Trabajos relacionados

La primera publicación relevante sobre refactoring fue realizada por Opdyke (1992). En su tesis doctoral presentó un conjunto de refactorings básicos que muestran cómo funcionalidades y atributos pueden migrar entre clases, estableciendo las precondiciones que deben ser mantenidas para asegurar que la transformación preserva el comportamiento

del programa. Además presentó refactorings más complicados como composiciones de los más simples. Los refactorings fueron definidos en términos de C++ y se ejemplificaron utilizando diagramas de clases (subconjunto de UML).

Roberts (1999) definió los refactorings como transformaciones de programa los cuales tienen precondiciones que deben mantenerse para poder llevar a cabo dichas transformaciones. Extiendió el trabajo de Opdyke especificando varios refactorings en términos de precondiciones y postcondiciones aplicados esencialmente a tres conceptos: clases, métodos y variables. Las precondiciones fueron expresadas en lógica de predicado de primer orden. Las postcondiciones fueron definidas en términos de funciones que forman la base para el análisis de programa que aseguran refactorings válidos. Como parte de su investigación desarrolló una herramienta para el refactoring de programas en lenguaje Smalltalk. En (Roberts y otros, 1997) se describe este browser avanzado para VisualWork que realiza automáticamente transformaciones que preservan el comportamiento.

Fowler (1999) incorporó los resultados de Roberts y Opdyke y describió principios y prácticas sobre refactoring. Detalló una serie de situaciones en el código que pueden presentar algún tipo de conflicto y que pueden resolverse aplicando un refactoring determinado. Ofreció un catálogo de refactorings, describiendo para cada uno de ellos la motivación y los mecanismos de transformación de código para realizarlos de manera controlada y eficiente. Aplicó informalmente las 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) estudiaron el refactoring de código en C++. Desarrollaron un conjunto de herramientas para transformar código C en clases C++, que permiten transformaciones automáticas y para tareas de reestructuración más complejas usaron escenarios combinando las herramientas con la intervención del programador. Las herramientas permiten realizar chequeos de precondiciones, cambios y compensaciones debido a los efectos colaterales.

En estas primeras publicaciones, los autores definen un conjunto de refactorings aplicados a código fuente sobre una plataforma particular: Opdyke los definió en términos de C++, Roberts sobre Smalltalk, Fowler los aplicó sobre Java y Fanta y Rajlich sobre C++. Estos trabajos se enmarcan dentro del nivel de modelos (primer nivel) del framework arquitectural propuesto en esta tesis (ver figura 1.1), concretamente trabajan en el nivel de modelos específicos a la implementación (ISM) (ver figura 1.2). La presente tesis propone una clasificación de refactorings basada en los diferentes tipos de modelos propuestos por MDA (PIM, PSM e ISM), asimismo fueron definidos como contratos entre metamodelos MOF y especificados formalmente.

A continuación se detallan varias propuestas que tratan sobre la reestructuración de modelos UML.

En (Gogolla y Ritchers, 1998) se propone transformar características avanzadas de diagramas de clases UML en construcciones más básicas con restricciones OCL. Las restricciones de cardinalidad, calificadores, clases asociación, agregaciones, composiciones y generalizaciones presentes en un diagrama de clases UML son transformadas usando sólo asociaciones n-arias y restricciones OCL. Esta propuesta provee un mejor

entendimiento de las características de UML mediante la explicación de las características más complejas en términos de las básicas.

En (Evans, 1998) se propone una técnica de análisis rigurosa para UML basada en el uso de transformaciones diagramáticas. El objetivo es mostrar cómo las técnicas de razonamiento riguroso pueden ser incorporadas a UML a nivel de las representaciones y abstracciones de sus componentes. Como estas abstracciones son presentadas usando diagramas, se investiga la manipulación diagramática como técnica de prueba para UML. Se proponen reglas de manipulación de modelos estáticos basadas en transformaciones diagramáticas. Estas reglas permiten que un diagrama que representa el modelo estático de un sistema sea transformado a un diagrama representando alguna propiedad o conclusión deducida sobre el sistema. El resultado final es un conjunto de transformaciones diagramáticas simples, que proveen las bases para el entendimiento y verificación de propiedades de diagramas de clases UML.

En (Sunyé y otros, 2001) se presenta un conjunto de refactorings y se explica cómo pueden ser diseñados para preservar el comportamiento de modelos UML, basados en restricciones OCL a nivel de metamodelos. Los refactorings presentados pueden ser resumidos en cinco operaciones básicas: adición, remoción, movimiento, generalización y especialización de elementos de modelado.

En (Whittle, 2002) se investiga el rol de las tranformaciones sobre diagramas de clases UML con restricciones OCL. Se presenta un framework para expresar transformaciones. Por ejemplo, la inferencia de nuevos vínculos de herencia a partir de la comparación de las restricciones, asociaciones, atributos y operaciones de dos clases existentes; o la expresión de información contenida en el diagrama a través de restricciones. Se describe una técnica para comprobar que un diagrama de clase UML modificado es el resultado de aplicar un refactoring a otro diagrama de clase.

Existe una propuesta que define e implementa refactorings de modelos en base a transformaciones basadas en reglas (Porres, 2003). Cada regla acepta uno o más elementos del modelo como parámetros y realiza acciones de transformación básicas basadas sobre estos parámetros. Las reglas se definen en base al metamodelo UML. Se presenta un algoritmo de ejecución para las reglas de transformación y se describe un mecanismo que asegura que los modelos transformados son bien formados. Sin embargo, no se discute si el refactoring es una mejora real del diseño o si éste preserva las propiedades de comportamiento del modelo.

Van Gorp y otros (2003) sostienen que el metamodelo UML es inadecuado para mantener la consistencia entre un modelo de diseño y su correspondiente código de programa, ya que no permite expresar los efectos que ciertos refactorings tienen sobre el código fuente. Por lo tanto, proponen un conjunto de extensiones mínimas al metamodelo UML, suficientes para razonar sobre refactoring para todos los lenguajes OO. Establecen contratos de refactoring sobre el metamodelo extendido basados en precondiciones y postcondiciones escritas en OCL. La implementación de dichos contratos de refactorings posibilita la automatización, la composición de refactorings primitivos y la verificación del comportamiento del programa. Sus investigaciones muestran que tales extensiones pueden aplicarse en herramientas CASE con OCL para componer secuencias de refactorings primitivos y detectar dónde puede aplicarse un refactoring dado. De esta manera, perfilan futuras herramientas MDA con la habilidad de transformar diseños UML existentes, manteniéndolos sincronizados con el código de programa.

Una técnica para describir transformaciones a nivel de metamodelo basadas en patrones de transformación se propone en (Judson y otros, 2003). Los autores sostienen que propuestas ad-hoc de refactoring de modelos UML pueden conducir a diseños difíciles de evolucionar, analizar y refinar. Para evitar esto, proponen refactorings que son llevados a cabo mediante el desarrollo de metamodelos para los refactorings. La definición de transformaciones a nivel de metamodelo se basa en *Patrones de Transformación*. Un patrón consiste de tres partes:

- *patrón fuente:* definen un conjunto de modelos fuente sobre los cuales pueden aplicarse las transformaciones,
- *esquema de transformación*: identifica las estructuras del modelo que son creadas y borradas por la transformación, determina la estructura básica del modelo destino,
- *restricciones de transformación*: define las relaciones que se deben mantener entre los elementos de los modelo fuente y destino.

La propuesta permite definir uno o más metamodelos para transformaciones basadas en patrones, donde cada metamodelo caracteriza una familia de transformaciones a nivel de modelo. Se ejemplifica con el desarrollo de un *Patrón de Transformación* para el refactoring del patrón de diseño Abstract Factory sobre diagramas de clases UML.

En (Judson y otros, 2004) se extiende la propuesta anterior para su aplicación sobre diagramas de clases y de interacción expresados en UML. Se definen Patrones de Transformación para el refactoring del patrón de diseño Abstract Factory para transformaciones sobre diagramas de clase y de interacción UML.

En los trabajos mencionados anteriormente se especifican refactorings de modelos UML empleando diferentes técnicas detalladas en la descripción de cada trabajo en particular. En esta tesis se los especifica utilizando los estándares propuestos por MDA más utilizados por la comunidad de diseñadores, se los clasifica en diferentes niveles de abstracción y se los integra con especificaciones formales.

Kerievsky (2004) detalló una serie de refactorings donde la transformación involucra patrones de diseño. Por cada uno de ellos presentó su motivación, los pasos a seguir para la transformación expresados en lenguaje natural y un ejemplo de su aplicación en el lenguaje de programación Java. Este autor define informalmente refactorings a nivel de modelos ISM (ver figura 1.2) y sobre una plataforma específica (lenguaje de programación Java).

Correa y Werner (2004) discuten la aplicación de técnicas de refactoring para mejorar la legibilidad de los modelos UML/OCL y para soportar su evolución. Presentan una colección de refactorings para diagramas UML y expresiones OCL. Especifican los refactorings como una operación sobre el grafo de instancias que representa al modelo UML/OCL. La propuesta de tesis además de usar una técnica de especificación alineada con MDA, los integra con especificaciones formales y propone una clasificación más amplia de los refactorings en base a modelos en distintos niveles de abstracción.

En (Long y otros, 2005) se formaliza y se prueba un conjunto de reglas de refactoring de Fowler. Se formalizan las reglas de refactorings como leyes de refinamiento utilizando Relational Calculus of Object System (rCOS). Las leyes de refinamiento se basan en el modelo semántico relacional de rCOS. rCOS incluye un lenguaje de especificación orientado a objetos que soporta características tales como subtipos, herencia, binding dinámico y polimorfismo; además soporta transformación y evolución de modelos a través

de cálculo de refinamiento estructural, refinamiento funcional y de propiedad, refinamiento de interacción y refinamiento de comportamiento de especificaciones, diseños y programas orientados a objetos. Si bien estos autores han formalizado los refactorings propuestos por Fowler, la presente tesis los enmarca dentro del contexto MDA, es decir, propone la definición de refactorings de modelos en distintos niveles de abstracción y posteriormente su formalización.

Markovic y Baar (2005) presentaron un conjunto de refactoring para diagramas de clase UML con anotaciones OCL, y los clasificaron según su impacto sobre las restricciones OCL. Describieron cómo las restricciones OCL deben ser reestructuradas para preservar la correctitud sintáctica. Las reglas de refactorings fueron definidas usando QVT para la especificación de transformaciones de modelos. Markovic y Baar (2007) investigaron cómo las expresiones OCL en diagramas de clase UML pueden ser sincronizadas cada vez que el diagrama subyacente es reestructurado. Las reglas de refactoring fueron expresadas en un formalismo gráfico basado en QVT y fueron implementadas en la herramienta RoclET. Los autores presentan una clasificación de refactorings y una técnica de especificación diferentes a las propuestas en esta tesis. Además no se integran con especificaciones formales ni se enmarcan en la arquitectura model-driven.

En (Massoni y otros, 2005) se propone refactoring de sistemas dirigido por modelos. El refactoring aplicado a un modelo es vinculado a una secuencia de transformaciones que automáticamente reestructuran el código fuente subyacente, basándose en propiedades estructurales que deben ser implementadas por el programa.

Jeanneret y otros (2006) describieron la arquitectura y funcionalidad de una herramienta llamada RoclET (Refactoring OCL Expressions by Transformations). Esta herramienta analiza gramaticalmente expresiones OCL, evalúa restricciones OCL en diagramas de objeto, realiza refactoring de modelos UML/OCL analizando el impacto que los cambios en el diagrama de objetos puedan tener sobre la evaluación de las restricciones OCL y un simplificador de expresiones OCL.

Ivkovic y Kontogiannis (2006) propusieron un framework para el refactoring de arquitecturas de software usando transformaciones de modelos y anotaciones semánticas relacionadas con mejoras de calidad. Describieron el contexto de refactoring utilizando *profiles* UML. Los modelos de arquitectura instanciados fueron anotados con elementos del contexto de refactoring incluyendo objetivos, métricas y restricciones. A partir de un conjunto de posibles refactorings se identifican aquellos que mejoran alguna característica específica. El aporte de estos autores se enmarca dentro del segundo nivel (nivel de metamodelo) del framework propuesto en la tesis (ver figura 1.1), es decir, la definición de refactorings de arquitecturas de software con una técnica diferente a la presentada en la tesis.

En (Stroggylos y Spinellis, 2007) se analizan sistemas de software de código abierto para detectar cambios marcados como refactorings y examinan si han sido afectadas las métricas del software. De esta manera, se evalúa si los refactorings efectivamente son usados para mejorar la calidad del software en la comunidad de código abierto.

Folli y Mens (2007) propusieron utilizar transformaciones de grafos como base para la especificación de refactoring de modelos. Esta propuesta permite representar transformaciones complejas de una manera visual compacta y provee una base formal para el análisis y la aplicación automática e interactiva de las transformaciones de modelos. Utilizaron una herramienta de transformación de grafo que permitió verificar la utilidad de la propuesta. Dicha herramienta es un entorno de programación visual basado en reglas y

soporta una propuesta algebraica para la transformación de grafos. En la tesis se propone una manera diferente de especificar los refactorings por emarcarse dentro del contexto MDA, es decir, basándose en los estándares más utilizados propuestos por esta arquitectura.

Varias publicaciones describen el estado del arte en el área de refactoring.

En (Demeyer y otros, 2002) se provee una amplia descripción de las investigaciones existentes en el área de refactoring, de herramientas y aplicaciones y de futuras tendencias.

Mens y Van Deursen (2003) identificaron tendencias en las investigaciones sobre refactoring y enumeraron una serie de problemas abiertos.

En (Du Bois y otros, 2004) se provee una visión general sobre reestructuración y refactoring tanto desde un punto de vista práctico como formal. Se provee un resumen sobre las actividades e investigaciones sobre refactoring y de las herramientas de soporte.

En (Mens y Tourwé, 2004) se provee una amplia descripción de las investigaciones existentes en el área de refactoring clasificadas según las actividades de refactorings soportadas, técnicas específicas y formalismos utilizados para soportar dichas actividades, clases de artefactos de software sobre los que realizan refactorings y sus efectos en los procesos de desarrollo de software.

Thomas (2005) analizó el estado del arte sobre refactoring y tópicos tales como el impacto de los lenguajes y herramientas sobre los refactorings, refactoring como meta programación y refactoring e instancias persistentes.

Varios trabajos abordan las distintas herramientas sobre refactoring existentes.

En (Moore, 1995) se describe una herramienta que soporta transformaciones automáticas para reestructurar jerarquías de herencia de objetos SELF preservando el comportamiento de los mismos.

Simmonds y Mens (2002) realizaron un estudio comparativo de distintas herramientas que soportan refactoring de software. Las herramientas analizadas son: Smalltalk VisualWork 7.0, Eclipse 2.0, Guru (para SELF 4.0) y Together ControlCenter 6.0.

Mealy y otros (2007) analizaron varias herramientas de refactoring de software para presentar un estudio sobre su funcionalidad. Las herramientas analizadas son: Eclipse 3.2, Condenser 1.05, RefactorIT 2.5.1 y Eclipse 3.2 con Simian UI 2.2.12 plugin.

Favre y otros (2008) analizaron las limitaciones de las herramientas CASE MDA existentes. Las herramientas analizadas son: OptimalJ, AndroMDA, Ameos, Together Architect, Codagen, ArcStyler, MDE Studio y Objecteering.

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. Borland Together (2007) aplica refactoring de código sobre requerimientos del usuario.

1.4 Aporte de esta tesis

Los primeros trabajos relacionados a refactoring se refieren a la aplicación de los mismos directamente sobre código en algún lenguaje de programación en particular. Otros describen transformaciones de distintos modelos UML, como diagramas de clase o diagramas de interacción, utilizando diferentes técnicas para la especificación de las transformaciones. En general, los refactorings se muestran como contratos OCL.

El aporte de esta tesis es la especificación de refactorings de modelos alineados con MDA, es decir, refactorings clasificados según los distintos modelos de diseño propuestos por la arquitectura MDA: modelos independientes de la plataforma, modelos específicos a una plataforma y modelos específicos a la implementación.

Otro aporte considerable de esta tesis es la integración de la especificación de los refactorings con especificaciones formales. El presente trabajo propone un sistema de traducción automática de la especificación de los refactorings a especificaciones en un lenguaje formal.

El presente trabajo de tesis se enmarca dentro de un proyecto donde uno de los objetivos es analizar la integración de diseños orientados a objetos basados en UML con especificaciones formales a fin de definir procesos basados en el paradigma transformacional para ingeniería directa (forward engineering), ingeniería reversa (reverse engineering) y refactoring de modelos UML dentro del contexto de la arquitectura model-driven.

1.5 Organización

En el capítulo 2, “Marco Teórico”, se incluye una breve descripción de la arquitectura Model Driven y el lenguaje de especificación NEREUS. Se introduce además las principales características de refactoring.

En el capítulo 3, “Refactoring en MDA”, se enmarcan los refactorings de modelos dentro del desarrollo Model Driven y se describen las especificaciones de los mismos como contratos OCL basados en metamodelos.

En los capítulos 4, 5 y 6 se especifican ejemplos de refactorings de modelos aplicables a nivel PIM, PSM e ISM respectivamente. Por cada refactoring se especifican los metamodelos origen y destino y se detalla la transformación como contrato OCL.

En el capítulo 7, “Formalización de Refactorings”, se detalla la traducción al lenguaje de especificación NEREUS de los metamodelos MOF y de los contratos de refactorings.

El capítulo 8, “Conclusiones”, incluye limitaciones al trabajo y futuros desarrollos.

CAPÍTULO 2

MARCO TEÓRICO

2.1 Model Driven Architecture

Debido a la proliferación y constante evolución de las plataformas y tecnologías, es difícil y costoso lograr la integración e interoperabilidad de los sistemas de software existentes.

Como respuesta a estos problemas, OMG ha introducido recientemente la arquitectura Model Driven como una propuesta para la especificación de sistemas basada en el uso de modelos.

El desarrollo de software en MDA comienza con un modelo de la funcionalidad y comportamiento de un sistema. Este modelo es transformado en modelos específicos de cada plataforma de interés y luego son traducidos a código.

Se detallan a continuación los beneficios que otorga la arquitectura Model-Driven.

- **Portabilidad**: MDA parte de un modelo de alto nivel de abstracción, independiente de cualquier plataforma, y ese modelo puede ser transformado en numerosos modelos que dependen de plataformas específicas. Cada modelo especificado independientemente de las plataformas es un modelo completamente portable. Para cada nueva tecnología y plataforma que surja en el futuro, se necesita definir las transformaciones correspondientes entre los distintos niveles de modelos. Esto permite que se desarrollen nuevos sistemas en las nuevas tecnologías y plataformas basados en los modelos existentes de alto nivel.
- **Productividad**: los desarrolladores se concentran en los modelos de alto nivel de abstracción sin preocuparse de los detalles específicos de las plataformas de destino, ya que esto es tomado en cuenta por la definición de transformación de modelos. El aumento en la productividad está dado por las herramientas que permiten automatizar gran parte de las transformaciones entre modelos de distintos niveles de abstracción.
- **Interoperabilidad**: se necesita establecer la relación entre los distintos modelos dependientes de plataformas, para alcanzar la interoperabilidad entre diferentes plataformas. MDA direcciona este problema estableciendo relaciones entre las distintas plataformas, determinando cómo los conceptos de una plataforma son usados en otra.
- **Mantenimiento y documentación**: los desarrollos model-driven se centran en los modelos de alto nivel de abstracción, éstos son usados para generar modelos dependientes de una plataforma y estos últimos para generar código. Los modelos independientes de plataforma proveen la documentación de alto nivel necesaria para cualquier sistema de software.

2.1.1 El framework MDA básico

MDA es un framework para el desarrollo de software donde los modelos son la base en el proceso de desarrollo de software. Las ideas centrales en MDA son:

- separar la especificación de la funcionalidad del sistema de su implementación sobre una plataforma en una tecnología específica y
- controlar la evolución de los modelos, desde modelos abstractos a implementaciones, tendiendo a aumentar el grado de automatización.

La Figura 2.1 muestra el framework MDA básico. A continuación se describen los elementos de dicho framework y sus roles (MDA, 2003).

- Un **modelo** es una descripción de la función, estructura y/o comportamiento de un sistema. Se distinguen los siguientes tipos de modelos:
  - Computer Independent Model (CIM): es un modelo con un alto grado de abstracción que muestra una perspectiva del sistema desde un punto de vista independiente de la computación. Un CIM no muestra detalles de la estructura del sistema, es comúnmente llamado modelo del dominio. El vocabulario usado en su especificación es el vocabulario familiar perteneciente al dominio en cuestión.
  - Platform Independent Model (PIM): describe la funcionalidad del sistema desde un punto de vista independiente de las características de plataformas de implementación específicas.
  - Platform Specific Model (PSM): describe un sistema desde el punto de vista de una plataforma específica, por ejemplo, .NET, J2EE, relacional. Un PSM combina las especificaciones de un PIM con detalles que especifican cómo el sistema usa una plataforma particular.
  - Implementation Specific Model (ISM): es la descripción del sistema a nivel de código, por ejemplo, Java, C#.
- Una **plataforma** es un conjunto de subsistemas y tecnologías que proveen un conjunto coherente de funcionalidades que pueden ser usadas en cualquier aplicación sin tener en cuenta detalles de cómo dichas funcionalidades son implementadas.
- Los modelos se escriben en un **lenguaje**. Algunos lenguajes son más expresivos que otros y/o más capaces de representar ciertos aspectos de un sistema. Si bien el lenguaje de modelado más comúnmente usado es UML, no debe restringirse el modelado a este lenguaje.
- Las **transformaciones** de modelos son esenciales en el desarrollo dirigido por modelos. Una transformación es el proceso de convertir un modelo (modelo origen) expresado en un lenguaje, en un nuevo modelo (modelo destino) expresado en otro lenguaje o eventualmente en el mismo.
- Una **definición de transformación** es la especificación de cómo un elemento del lenguaje origen puede ser transformado en uno o más elementos del lenguaje destino.
- Una herramienta **de transformación** soporta el proceso de transformar un modelo en otro basada en la definición de transformación.

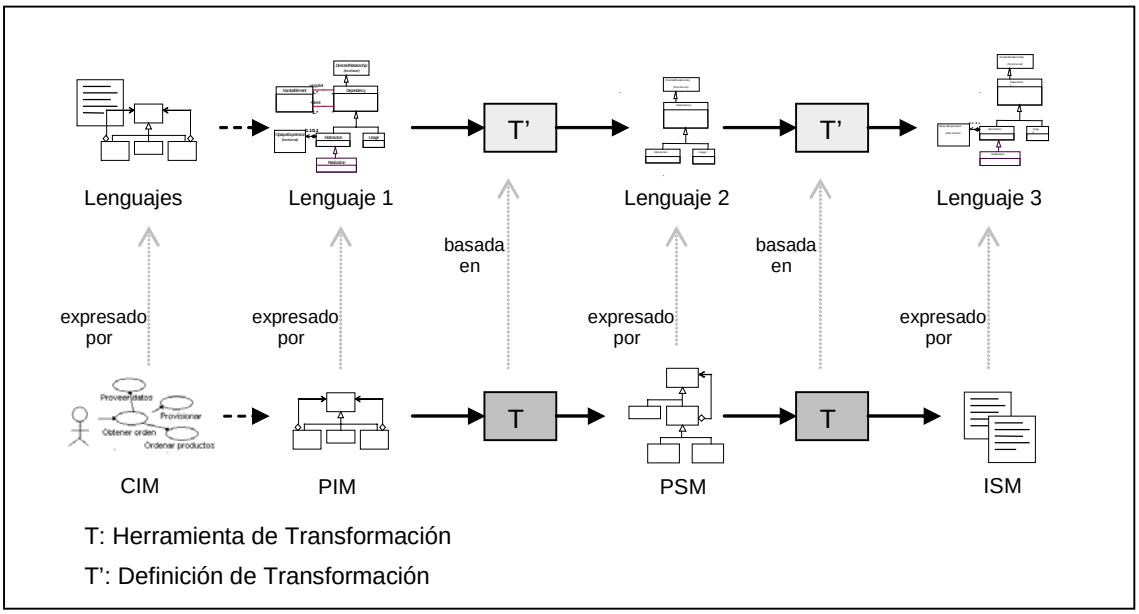


Figura 2.1 - Framework básico de MDA

2.1.2 Metamodelado en MDA

Un modelo es una descripción o especificación del sistema o parte del mismo. Los lenguajes de modelado también necesitan ser definidos en el contexto de MDA. El mecanismo que permite definir la sintaxis y semántica de los lenguajes de modelado es el metamodelado, es decir, construir un modelo del lenguaje de modelado llamado metamodelo. Por ejemplo, el lenguaje de modelado UML está escrito en UML (el metamodelo UML), dando una definición del lenguaje en términos de sí mismo. Como un metamodelo es también un modelo, puede describirse mediante un lenguaje, el metalenguaje (ver Figura 2.2).

Un modelo define qué elementos pueden existir en un sistema. Un lenguaje define qué elementos pueden existir en un modelo, por ejemplo, el lenguaje de modelado UML define los conceptos “Class”, “Attribute”, “Association”, etc., que pueden ser usados en los modelos estáticos UML. De esta manera se puede describir un lenguaje por medio de un modelo, describiendo qué elementos pueden ser usados en el lenguaje.

El metamodelado es una técnica importante dentro de MDA por las siguientes razones:

- permite definir los lenguajes de modelado, permitiendo que las transformaciones puedan ser realizadas automáticamente por herramientas capaces de leer, escribir y entender los modelos;
- las reglas de transformación, que definen transformaciones relacionando cada elemento del modelo fuente con uno o más elementos del modelo destino, usan los metamodelos de los lenguajes fuente y destino para definir transformaciones relacionando las metaclases de los elementos en los respectivos lenguajes.

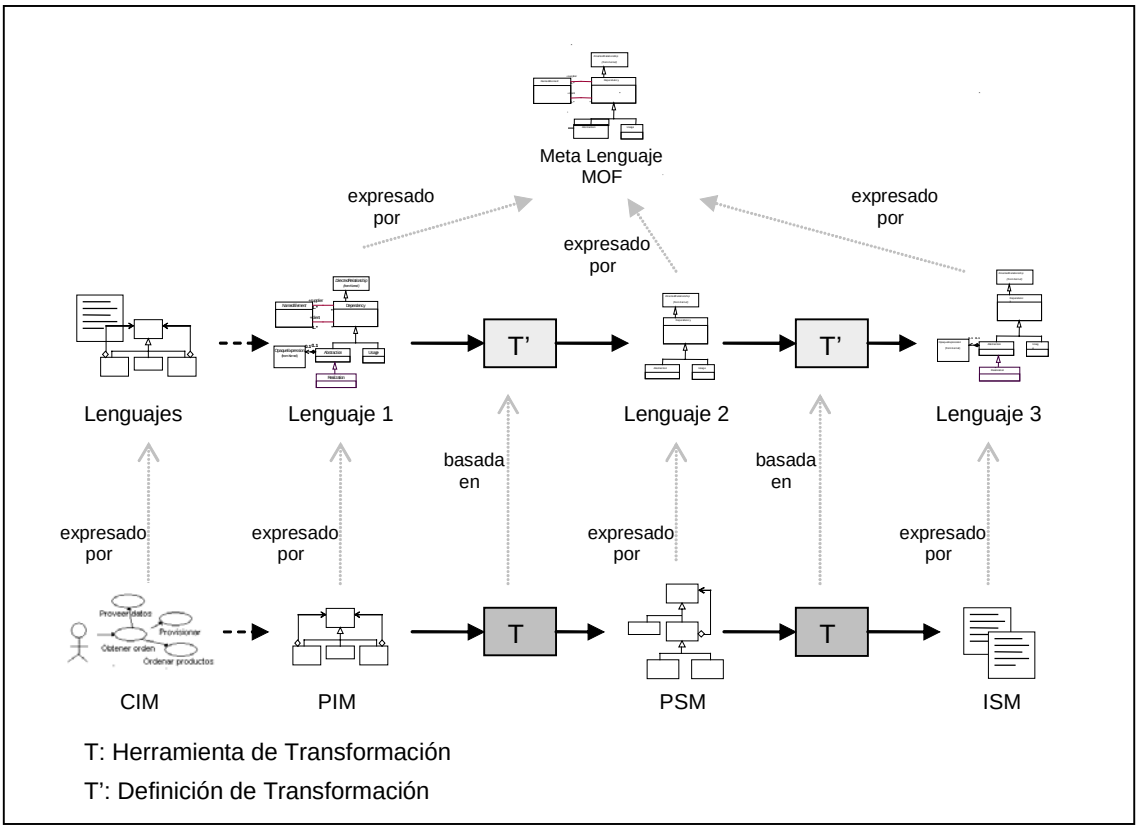


Figura 2.2 – Framework extendido de MDA

La Figura 2.3 muestra la jerarquía de niveles de modelos de cuatro capas que el OMG utiliza para sus estándares, donde cada nivel (excepto el superior) se caracteriza por ser una *instancia-de* el nivel que está sobre él. (Kleppe y otros, 2003).

En la terminología OMG, los niveles de modelado son los siguientes:

Nivel M0: Sistema

En este nivel se encuentran los datos reales que el sistema manipula.

Nivel M1: El modelo del sistema

Este nivel contiene modelos, por ejemplo, el modelo UML del sistema de software del nivel M0. Los conceptos que aparecen en este nivel son clasificaciones o categorizaciones de las instancias que aparecen en el nivel M0, por lo tanto, cada elemento en el nivel M0 será una instancia de un elemento del nivel M1.

Nivel M2: El modelo del modelo

Cada elemento del nivel M1 es una instancia de un elemento de M2, y cada elemento de M2 clasifica elementos de M1. Por ejemplo, cada modelo UML del nivel M1 es una instancia del metamodelo UML. El modelo que reside en este nivel se llama metamodelo. Cuando se construye un metamodelo se está definiendo un lenguaje con el cual escribir modelos. En este nivel se encuentran el metamodelo UML, CWM (Common Warehouse Model) y SPEM (Software Process Engineering Metamodel) entre otros estándares de OMG.

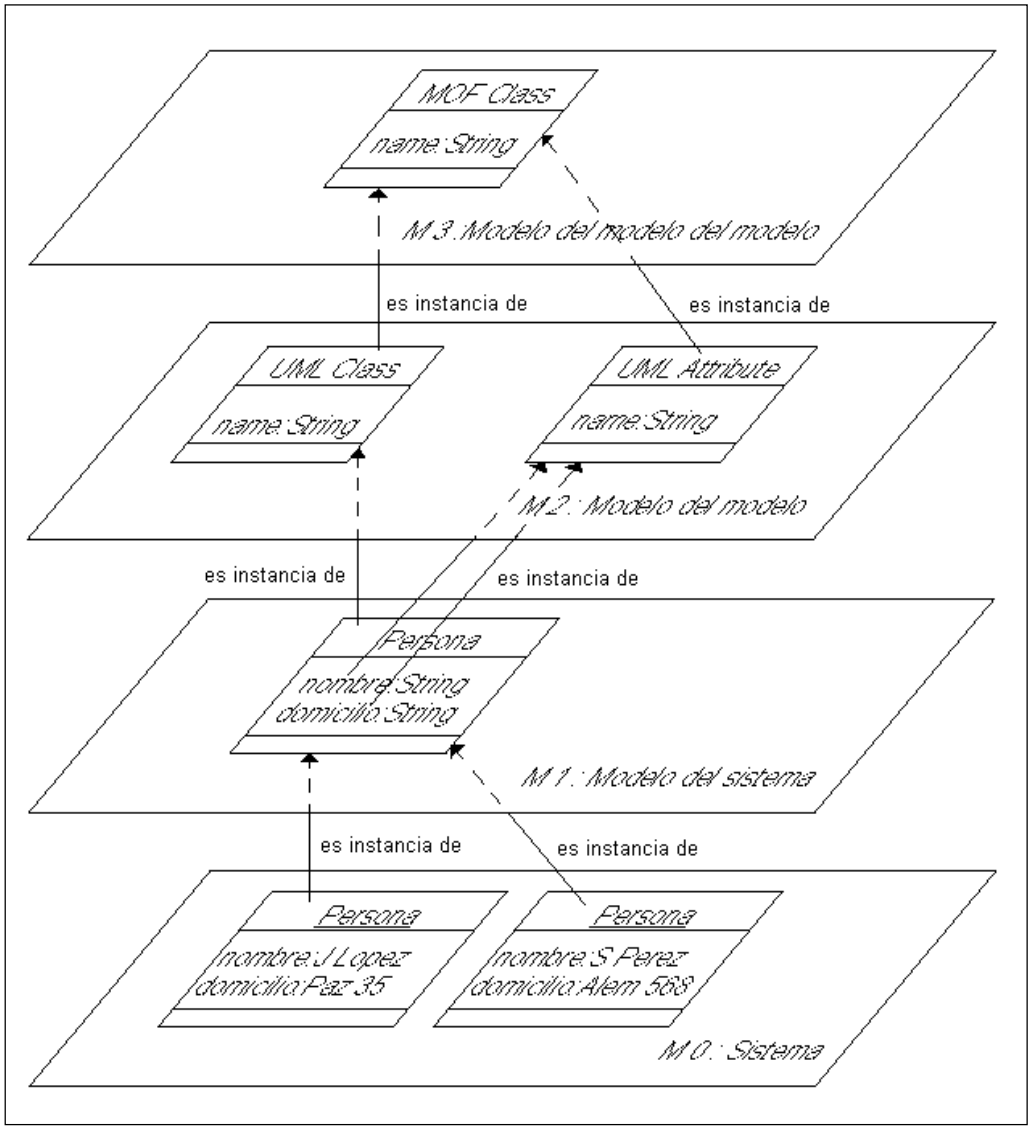


Figura 2.3 – Arquitectura de modelado del OMG

Nivel M3: El modelo del modelo del modelo

Cada elemento del nivel M2 es una instancia de un elemento de M3, y cada elemento de M3 clasifica elementos de M2. El modelo en este nivel se llama metametamodelo que define el lenguaje para definir lenguajes de modelado. El lenguaje estándar de OMG del nivel M3 es el MOF. El metamodelo UML, CWM y SPEM son instancias del MOF.

Se podría seguir agregando niveles en la jerarquía, pero como esta separación de niveles es superficial, ya que simplemente ayuda a razonar sobre modelos y clasificaciones, OMG ha declarado que todos los elementos del nivel M3 deben ser definidos como instancias de los conceptos del mismo nivel M3.

2.1.3 Estándares de OMG

OMG ha adoptado diferentes tecnologías que posibilitan la propuesta model-driven. Se describen a continuación las más relevantes para aplicar el enfoque MDA.

Los modelos orientados a objetos usados en MDA pueden ser expresados usando UML, el cual es el lenguaje de modelado estándar para visualizar, especificar y documentar componentes de un sistema de software. UML permite definir, a través de distintos diagramas, las distintas vistas que componen un modelo. Estos diagramas proveen múltiples perspectivas de un sistema que está siendo analizado o desarrollado. El modelo subyacente integra estas perspectivas permitiendo de esta manera la construcción de un sistema consistente y autocontenido. Los modelos pueden enriquecerse con anotaciones en OCL que permite escribir expresiones sobre cada elemento de los diagramas obteniendo modelos más precisos y completos. Este lenguaje tiene una sintaxis simple, su contexto sintáctico está determinado gráficamente y no tiene efectos colaterales por ser un lenguaje de expresiones puro.

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. MOF usa un framework de modelado que es un subconjunto del núcleo de UML. Las construcciones de modelado son: clases, que modelan metaobjetos MOF; asociaciones, que modelan relaciones binarias entre metaobjetos; tipos de datos, que modelan otros datos (por ejemplo tipos primitivos, tipos externos, etc); packages, que modularizan los modelos. Los metamodelos MOF se expresan como una combinación de diagramas de clases UML y especificaciones OCL.

Las transformaciones en MDA se especifican con QVT. Este estándar direcciona las transformaciones entre modelos cuyos lenguajes se definen usando MOF. QVT consta de un lenguaje para realizar consultas sobre modelos, un lenguaje para crear vistas sobre un modelo y un lenguaje para escribir definiciones de transformaciones. QVT depende de MOF y OCL para especificar consultas, vistas y transformaciones. Una consulta selecciona elementos específicos de un modelo, una vista es un modelo derivado a partir de otro modelo y una transformación es la especificación de un mecanismo que convierte elementos de un modelo (instancia de un metamodelo particular) en elementos de otro modelo que puede ser una instancia del mismo o diferente metamodelo.

2.2 El lenguaje de especificación NEREUS

Los metamodelos MOF se basan en los conceptos de entidades, interrelaciones y sistemas. La mayoría de estos conceptos son traducidos a NEREUS de forma directa. NEREUS es un lenguaje de metamodelado algebraico que posee construcciones para expresar clases, asociaciones y paquetes y un repertorio de mecanismos para estructurarlos. Es un lenguaje centrado en las relaciones, es decir, puede expresar diferentes tipos de relaciones (dependencia, asociación, agregación, composición) como primitivas para desarrollar especificaciones (Favre, 2005).

La semántica de NEREUS fue dada por traducción a CASL (Common Algebraic Specification Language) (Favre, 2006, 2007). CASL está basado en conceptos estándares de especificación algebraica y brinda un *framework* unificado para la especificación algebraica (Bidoit y Mosses, 2004; Mosses, 2004).

Se presenta a continuación la sintaxis del lenguaje NEREUS.

2.2.1 Definición de clases

La clase es la unidad básica de especificación y puede declarar tipos, operaciones y axiomas (fórmulas en lógica de primer orden). Están estructuradas por medio de diferentes tipos de relaciones: importación, herencia y subtipo. La figura 2.4 muestra su sintaxis.

```
CLASS  className [<parameterList>]
    IMPORTS  <importList>
    INHERITS <inheritList>
    IS-SUBTYPE-OF <subtypeList>
    GENERATED-BY <basicConstructors>
    ASSOCIATES <associatesList>
    DEFERRED
        TYPES <typesList>
        FUNCTIONS <functionList>
    EFFECTIVE
        TYPES <typesList>
        FUNCTIONS <functionList>
        AXIOMS <varList> <axiomList>
END-CLASS
```

Figura 2.4 - Sintaxis de una clase en NEREUS

La definición de una clase comienza con la palabra reservada CLASS seguida del nombre de la clase. NEREUS distingue partes variables en una especificación por medio de la parametrización explícita. Los elementos de <*parameterList*> son pares *C1:C2* donde C1 es el parámetro genérico formal restringido por una clase existente C2 (sólo subclases de C2 podrán ser parámetros reales).

La cláusula IMPORTS expresa relaciones de dependencia. La especificación de la nueva clase está basada en las especificaciones importadas declaradas en <*importList*> y las operaciones públicas de dichas especificaciones importadas pueden ser usadas en la nueva especificación.

La cláusula INHERITS especifica que la clase es construida a partir de la unión de las clases que aparecen en <*inheritList*>. Los componentes de cada una de ellas serán componentes de la nueva clase, y sus propios tipos y operaciones serán tipos y operaciones de la nueva clase. NEREUS distingue dos conceptos: herencia y subtipo. Subtipo es herencia de comportamiento, mientras que la herencia se basa en la visión de módulo de las clases. La herencia es expresada con la cláusula INHERITS, donde la especificación de la clase es construida a partir de la unión de las especificaciones de las clases que aparecen en <*inheritList*>. La relación de subtipo se declara con la cláusula IS-SUBTYPE-OF, y satisface la propiedad de que cada objeto de una subclase es al mismo tiempo un objeto de su superclase.

La claúsula GENERATED-BY lista las operaciones constructoras básicas.

La cláusula ASSOCIATES especifica las relaciones que posee la clase.

NEREUS distingue partes diferidas y efectivas. La cláusula DEFERRED declara tipos y operaciones que no están completamente definidos. La cláusula EFFECTIVE agrega tipos

y operaciones completamente definidos o completa la definición de algún tipo u operación definido en forma incompleta en alguna superclase.

En la claúsula TYPES se declaran los tipos de la clase.

En la cláusula FUNCTIONS se declaran las funcionalidades de las operaciones, la lista de los argumentos y el tipo del resultado. Es posible definir operaciones en forma parcial, especificando su dominio por medio de la cláusula PRE que indica las condiciones que deben satisfacer los argumentos de la función para pertenecer al dominio de la función. La visibilidad de una operación puede ser pública, protegida o privada y se denota con los símbolos +, # y – respectivamente precediendo al nombre de la operación.

NEREUS permite especificar signaturas de operaciones de manera incompleta. La notación _ (guión bajo*)* en una funcionalidad de operación de una clase indica que es incompleta, es decir, cualquier subclase de ésta puede completarla. De la misma manera, la subclase podría usar el símbolo * (asterisco) para indicar que hereda la funcionalidad de la operación heredada.

NEREUS soporta operaciones *higher-order* cuando al especificar requerimientos es necesario escribir operaciones que dependen de otras operaciones, estas últimas pueden ser vistas como parámetros de la primera.

Cuando se especifica una función, puede utilizarse la construcción LET...IN para introducir nombres locales para subtérminos que ocurren en el cuerpo de la función ya sea porque son comunes o porque es necesario definirlos localmente.

Tras la palabra clave AXIOMS se declaran pares de la forma *v1: C1* donde v1 es una variable universalmente cuantificada de tipo C1. Los axiomas incluidos a continuación de esta declaración expresan las propiedades requeridas por la especificación a partir de fórmulas de primer orden construidas sobre términos y fórmulas.

2.2.2 Definición de asociaciones

La figura 2.5 muestra la sintaxis que permite definir asociaciones:

```
ASSOCIATION <relationName>
  IS <typeConstructorName>
  […:Class1;…:Class2;…:Role1;…:Role2;
  …:Mult1;…:Mult2;…:Visibility1;…:Visibility2]
  CONSTRAINED BY <constraintList>
END
```

Figura 2.5. Sintaxis de una asociación en NEREUS

La cláusula IS expresa una relación de instanciación de constructores de tipos <*typeConstructorName*> a partir de una lista de parámetros. La lista contiene pares de la forma *A: B* donde *B* es un parámetro del esquema vinculado a la asociación y *A* su instanciación. El tipo de interés de <*typeConstructorName*> es renombrado por <*relationName*>. Los esquemas están parametrizados en las clases que intervienen, el rol, la visibilidad y mutiplicidad de cada *association-end.*

La cláusula CONSTRAINED-BY posibilita especificar *constraints* estáticos en lógica de primer orden que pueden ser aplicados a la asociación y/o a los extremos de asociación.

NEREUS provee primitivas para la definición de asociaciones a través de una clasificación de constructores de tipos que permiten clasificar asociaciones binarias de acuerdo a su:

- tipo: agregación (ordinaria o composición), asociación ordinaria, asociación calificada y clase asociación.
- navegabilidad: unidireccional o bidireccional.
- conectividad: uno-a-uno, muchos-a-muchos, uno-a-muchos, etc.

La Figura 2.6 grafica la clasificación de constructores de tipos.

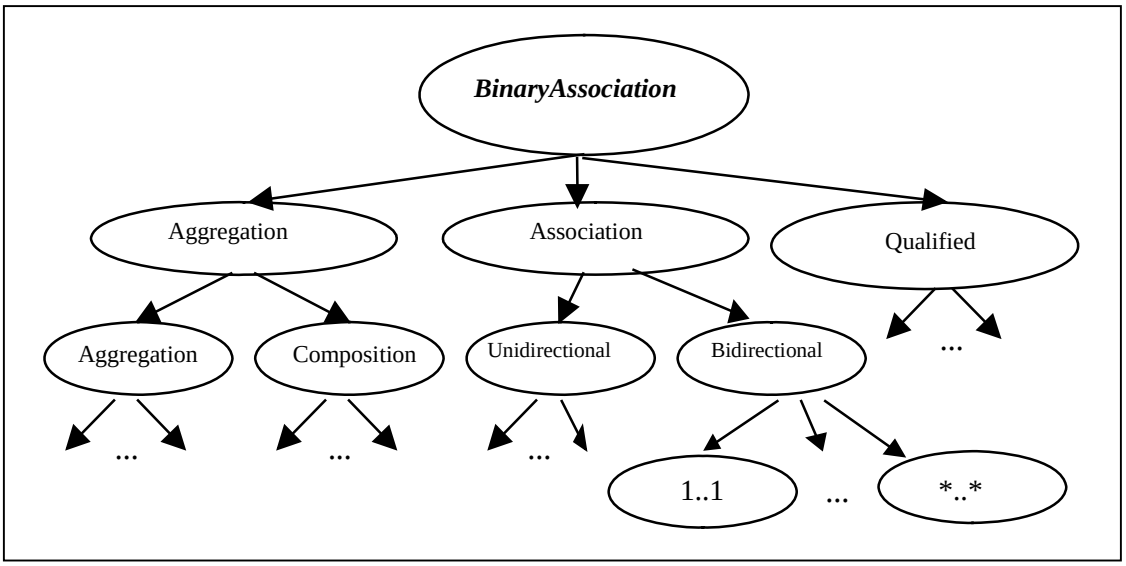


Figura 2.6 - La jerarquía de constructores de tipos *BinaryAssociation*

Para definir constructores de tipos se especifica un conjunto de esquemas reusables que permiten definir relaciones concretas por mecanismos de instanciación. La figura 2.7 muestra los esquemas de una rama de la jerarquía de la figura 2.6.

2.2.3 Definición de paquetes

El paquete es el mecanismo que provee NEREUS para agrupar clases y asociaciones y controlar su visibilidad. La sintaxis de un paquete en NEREUS se muestra a continuación, donde <*importsList*> lista los paquetes importados, <*inheritsList*> lista los paquetes heredados y <*elements*> son clases, asociaciones y paquetes.

```
PACKAGE packageName
    IMPORTS <importsList>
    INHERITS <inheritsList>
    <elements>
END-PACKAGE
```

RELATION SCHEME BinaryAssociation
IMPORTS *Class1, Class2,* Boolean, Multiplicity, Visibility, String, Typename
DEFERRED
TYPES BinaryAssociation
FUNCTIONS
name: BinaryAssociation → TypeName
frozen : BinaryAssociation → Boolean
changeable : BinaryAssociation → Boolean
addOnly : BinaryAssociation → Boolean
getRole1: BinaryAssociation → String
getRole2: BinaryAssociation → String
getMult1: BinaryAssociation → Multiplicity
getMult2: BinaryAssociation → Multiplicity
getVisibility1: BinaryAssociation → Visibility
getVisibility2: BinaryAssociation → Visibility
END-RELATION

RELATION SCHEME Aggregation
IS-SUBTYPE-OF BinaryAssociation
[*Whole:* Class1, *Part*: Class2]
DEFERRED
FUNCTIONS
is*Part* : Aggregation x *Whole* x *Part* → Boolean
isEmpty: Aggregation → Boolean
isLinkedWhole:Aggregation x *Whole* → Boolean
isLinkedPart:Aggregation x *Part* → Boolean
END-RELATION

RELATION SCHEME Composition-1
--composition/simple/ 1..1/not frozen
IS-SUBTYPE-OF Aggregation
GENERATED-BY create, add
EFFECTIVE
TYPES Composition-1
FUNCTIONS
name, frozen, changeable, addOnly ,
getRole1, getRole2, getMult1, getMult2,
getVisibility1, getVisibility2, get*Part*, get*Whole*,
is*Part*, isEmpty, isLinkedPart, isLinkedWhole
create: TypeName → Composition-1
add*Part*: Composition-1(c) x *Part(p)* x *Whole(w)* → Composition-1
pre: rightCardinality(a,w) <1
and leftCardinality(a,p) <1

remove*Part*: Composition-1 (a) x *Whole* (w) x *Part*(p) → Composition-1
pre: isLinkedPart(a,w,p) and not addOnly (a) and not frozen (a)
leftCardinality: Composition-1 x *Part* → Nat
rightCardinality: Composition-1 x *Whole* → Nat
getPart: Composition-1(c) x Whole(w) → Part
pre: isLinkedWhole(c,w)
getWhole: Composition-1(c) x Part (p) → Whole
pre: isLinkedPart(c,p)
AXIOMS a: Composition-1; p, p1: *Part*; w, w1: *Whole*; t: TypeName
name(create(t)) = t
name(add*Part*(a, p, w)) = name (a)
frozen (a) = False
changeable (a) = True
add-Only (a) = <*True or False*>
getMult1 (a) = 1
getMult2 (a) = 1
getRole1(a) = <*name-role1*>
getRole2(a) = <*name-role2*>
getVisibility1= <*visibility*>
getVisibility2= <*visibility*>
get*Part*(add*Part*(a, p, w)) = if w = w1 then p else getPart (a, w1)
get*Whole* (add*Part*(a, p, w), p1)) = if p = p1 then w else getWhole(a, p1)
is*Part* (create(t), p1) = False
is*Part*(add*Part*(a,p,w),w1,p1) = (w=w1 and p=p1) or isPart (a,w1,p1)
isEmpty (create (t)) = True
isEmpty(add*Part* (a, p, w)) = False
remove(add*Part*(a, p, w), p1, w1) = if (p = p1 and w = w1)
then a else remove (a, p1, w1)
isLinkedPart (create(t), p) = False
isLinkedPart(add*Part*(a, w, p), p1) = p=p1 or isLinkedPart (a, p1)
isLinkedWhole (create(t), p)= False
isLinkedWhole (add*Part*(a, w, p), w1) = w = w1 or isLinkedWhole (a, w1)
leftCardinality(create(t), p) = 0
leftCardinality (addPart (a, w, p), p1) = if p=p1 then 1
else leftCardinality(a, p1)
END-RELATION

Figura 2.7 – Esquemas reusables *BinaryAssociation*

2.3 Refactoring

El término **reestructuración** de software se refiere a transformaciones aplicadas sobre una forma de representación que producen otra nueva forma de representación del software. La reestructuración aplicada sobre programas fuente, se la llama transformación de programa, pero también podrían ser aplicadas en niveles más altos de abstracción, como por ejemplo, diseños o arquitecturas.

Refactoring es el término equivalente a reestructuración en el contexto de la programación orientada a objetos. Reestructuración es un término más general, puede ser aplicado a cualquier clase de artefacto de software, en cualquier lenguaje y en cualquier nivel de abstracción, no necesariamente dentro del paradigma de orientación a objetos.

Martin Fowler define al término como (Fowler, 1999):

> **refactoring (sustantivo)**: un cambio en la estructura interna del software para hacerlo más fácil de entender y menos costoso de modificar sin cambiar el comportamiento observable del sistema.
>
> **refactoring (verbo)**: es la reestructuración de software mediante la aplicación de una serie de refactorings sin cambiar el comportamiento obervable del sistema.

Refactoring desde el punto de vista de Fowler puede ser caracterizado por lo siguiente:

- trata de la estructura interna del software,
- preserva el comportamiento observable,
- mejora una situación dada de acuerdo a un objetivo expresado informalmente; ejemplos de dichos objetivos son la reducción de costo de desarrollo, mejoras en la legibilidad, mantenimiento y velocidad de ejecución, demanda de memoria, etc.,
- los pasos de refactoring son pequeños y pueden ser combinados sistemáticamente permitiendo construir estrategias más sofisticadas,
- es una técnica constructiva basada en reglas que parte de una situación dada, un objetivo y una serie de pasos constructivos para lograr dicho objetivo,
- es aplicado por desarrolladores de software,
- la corrección de la aplicación de las reglas de refactoring es responsabilidad del desarrollador.

Esto significa que los refactorings son considerados transformaciones de software que reestructuran un programa orientado a objetos mientras preservan su comportamiento. La idea clave es la redistribución de atributos y métodos en la jerarquía de clases para adecuar al software a futuras extensiones y cambios. La aplicación de refactorings sobre código tiene las siguientes ventajas.

- **Mejora el diseño del software**: cuando se realizan cambios para alcanzar objetivos a corto plazo sin tener una completa visión del diseño total se pierde la estructura del código. La pérdida de estructura tiene un efecto acumulativo, más difícil de ver el diseño en el código, más difícil es preservarlo y más rápidamente decae. Aplicar regularmente refactorings ayuda a que el código mantenga su forma. Otro aspecto importante para mejorar el diseño es la eliminación de la duplicación de código. Los diseños pobres usualmente poseen más código para hacer la misma cosa, porque encontramos código que hace lo mismo en varios lugares.
- **Mejora el entendimiento del software**: al generar código más legible, éste comunica más fácilmente el propósito para el cual fue diseñado.
- **Ayuda a encontrar errores**: al mejorar el entendimiento del software se pueden ver cosas sobre el diseño que antes no se observaban, por lo tanto se pueden detectar errores más claramente.

- **Ayuda a desarrollar código más rápidamente**, ya que los buenos diseños ayudan a desarrollar más rápidamente código al no tener que perder demasiado tiempo detectando y depurando errores.

Si bien los refactorings son típicamente aplicados a nivel de programa, es decir, sobre código fuente, pueden aplicarse también como transformaciones de modelos.

France y Bieman (2001) categorizan las transformaciones de modelos según dos dimensiones:

- **Transformación vertical**: ocurre cuando un modelo fuente es transformado en otro modelo destino en diferente nivel de abstracción, por ejemplo, cuando se transforma un modelo a código fuente en un lenguaje de programación. En el contexto de MDA, las transformaciones verticales son útiles para transformar un modelo independiente de plataforma (PIM) en un modelo dependiente de plataforma (PSM) y éste en un modelo dependiente de la implementación (ISM).
- **Transformación horizontal**: ocurre cuando un modelo fuente es transformado en otro modelo destino en el mismo nivel de abstracción. Estas transformaciones son realizadas para soportar la evolución de los modelos. Se identifican tres tipos de evolución de modelos:
  - **evolución correctiva**, se refiere a la corrección de errores en el diseño,
  - **evolución adaptativa**, se refiere a la modificación de un diseño para contemplar cambios en los requerimientos y
  - **evolución perfectiva**, se refiere a la modificación de un diseño para mejorar ciertas características del modelo.

Refactoring de modelos tiene que ver con transformaciones que soportan la evolución perfectiva de los modelos para mejorar algún aspecto de calidad. Todo artefacto de software tiene atributos de calidad tales como robustez, adaptabilidad, reusabilidad, compatibilidad, facilidad en su uso, portabilidad y comprensibilidad. Los refactorings pueden ser clasificados según los atributos de calidad que mejoran, permitiendo aplicar refactorings relevantes donde sea necesario y de esta manera incrementar la calidad del software. Las métricas de software asocian un valor numérico a cada atributo de calidad, pueden ser usadas para identificar áreas de problemas y para evaluar las mejoras resultantes después de la aplicación de un refactoring.

La aplicación de refactorings a nivel de modelos tiene las ventajas, además de las expuestas anteriormente, de que las modificaciones son realizadas en etapas tempranas del desarrollo de software y no están sujetas a ningún lenguaje de programación en particular.

Refactoring es una actividad importante en los procesos de desarrollo de software. A continuación se detallan los procesos en los cuales se adecua la aplicación de la técnica de refactoring (Mens, 2004).

- En la propuesta MDA, los modelos son los artefactos primarios del desarrollo de sistemas y las transformaciones de modelos bien definidas son claves para soportar la evolución de los modelos, su refinamiento y su realización en código.
- En los procesos de desarrollo ágiles como XP (eXtreme Programming), metodología desarrollada por Kent Beck, la idea central es trabajar sobre un único caso de uso por vez y sólo diseñar el software para manejar dicho caso de uso (Beck, 2000). Si un caso

de uso particular no se adecua correctamente a su diseño, se reestructura el diseño hasta lograr que el caso de uso sea implementado de manera razonable. Por lo tanto, los refactorings continuos y agresivos, son un aspecto clave en XP.

- En los procesos de ingeniería forward se transforman modelos de un alto nivel de abstracción en la implementación de un sistema.
- En procesos de ingeniería reversa se analiza un sistema para identificar sus componentes y sus interrelaciones y se crean modelos en un mayor nivel de abstracción.
- En los procesos de reingeniería**,** cuyo objetivo es reestructurar software *legacy*, se transforma una representación de bajo nivel en otra, mientras se reconstituyen modelos de mayor nivel de abstracción a lo largo del proceso.

CAPÍTULO 3

REFACTORING EN MDA

Se define *refactoring* como el proceso de reestructurar un modelo orientado a objetos aplicando una secuencia de transformaciones que preservan la funcionalidad del mismo a fin de mejorar algún factor de calidad. Es una propuesta transformacional para el desarrollo de software iterativo. Se basa en la idea de introducir cambios en un modelo en pasos pequeños y sistemáticos donde cada paso mejora el modelo de acuerdo a alguna métrica específica (figura 3.1). Los refactorings resultan una técnica poderosa cuando son aplicados repetidamente en el modelo (Philipps y Rumpe, 2003).

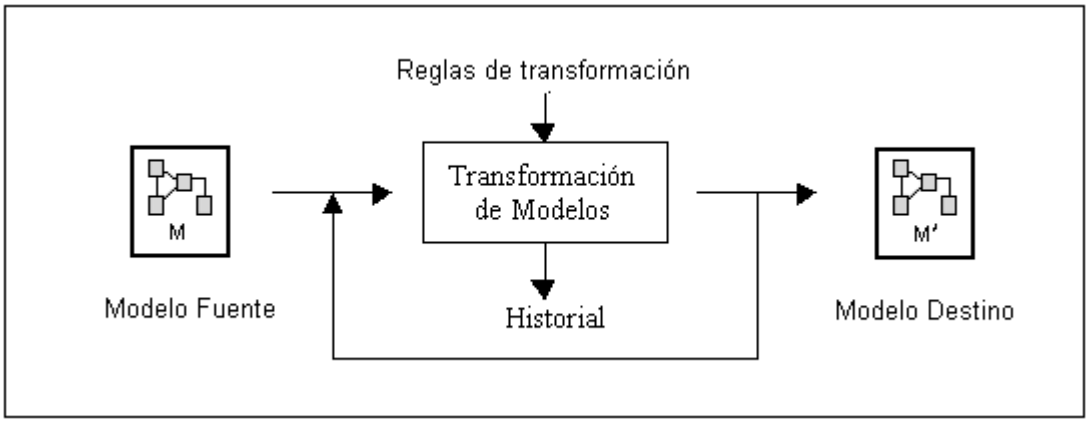


Figura 3.1 - Refactoring de modelos

Los refactorings definen un conjunto de reglas que:

- permiten transformar gradualmente y en forma automática tanto una jerarquía de clases como otros elementos de un modelo para lograr mejoras en el mismo,
- preservan el comportamiento del modelo resultante,
- garantizan la consistencia del modelo resultante,
- permiten la intervención del diseñador para aplicar distintas estrategias o alternativas de reestructuración.

La transformación de modelos es un proceso que posee las siguientes características:

- se basa en la aplicación de reglas para el refactoring de modelos,
- cada refactoring se realiza sobre un subconjunto del modelo fuente seleccionado por el diseñador,

- utiliza un conjunto de reglas para la reestructuración de modelos que permiten la transformación gradual y automática garantizando consistencia y equivalencia funcional,
- la aplicación de cada refactoring puede crear nuevos elementos en el modelo, actualizar o eliminar elementos existentes,
- la aplicación de cada refactoring debe producir un modelo destino funcionalmente equivalente al modelo fuente,
- permite mantener el historial de las transformaciones aplicadas para soportar *traceability.*

En (Pereira y Favre, 2004) y (Pereira y otros, 2004) se presenta un sistema transformacional basado en reglas y estrategias para el refactoring de modelos estáticos UML. Se describe un conjunto de reglas para reestructurar jerarquías de clases y asociaciones que preservan la semántica de los modelos.

3.1 Refactoring y MDD

El desarrollo model driven (MDD) se lleva a cabo a través de una secuencia de transformaciones de modelos que parte desde la construcción de un modelo independiente de la computación, la transformación a un modelo independiente de detalles de implementación, la transformación de este modelo en uno o más modelos dependientes de alguna plataforma de implementación y la derivación de éstos a código. MDD propone el uso de los modelos para direccionar el curso de cada etapa en el proceso de desarrollo de sistemas. Por esto es importante la aplicación de refactorings como herramienta para reestructurar los modelos que son el punto de partida en la secuencia de transformaciones. El objetivo de los refactorings es la evolución perfectiva de los modelos a fin de mejorar ciertos aspectos de calidad sin cambiar el comportamiento observable del sistema, es decir, manteniendo inalterable su funcionalidad y su semántica. La figura 3.2 muestra cómo los refactorings se adecuan dentro del framework básico MDD basado en MDA. En la figura se exhiben:

- **Modelos** en diferentes niveles de abstracción:
  - CIM (Computer Independent Model): modelo con un alto grado de abstracción, es una perspectiva del sistema desde un punto de vista independiente de la computación.
  - PIM (Platform Independent Model): modelo que describe la funcionalidad del sistema desde un punto de vista independiente de las características de plataformas de implementación específicas.
  - PSM (Platform Specific Model): describe un sistema desde el punto de vista de una plataforma específica, por ejemplo, .NET, J2EE, relacional.
  - ISM (Implementation Specific Model): es la descripción del sistema a nivel de código, por ejemplo, Java, C#.

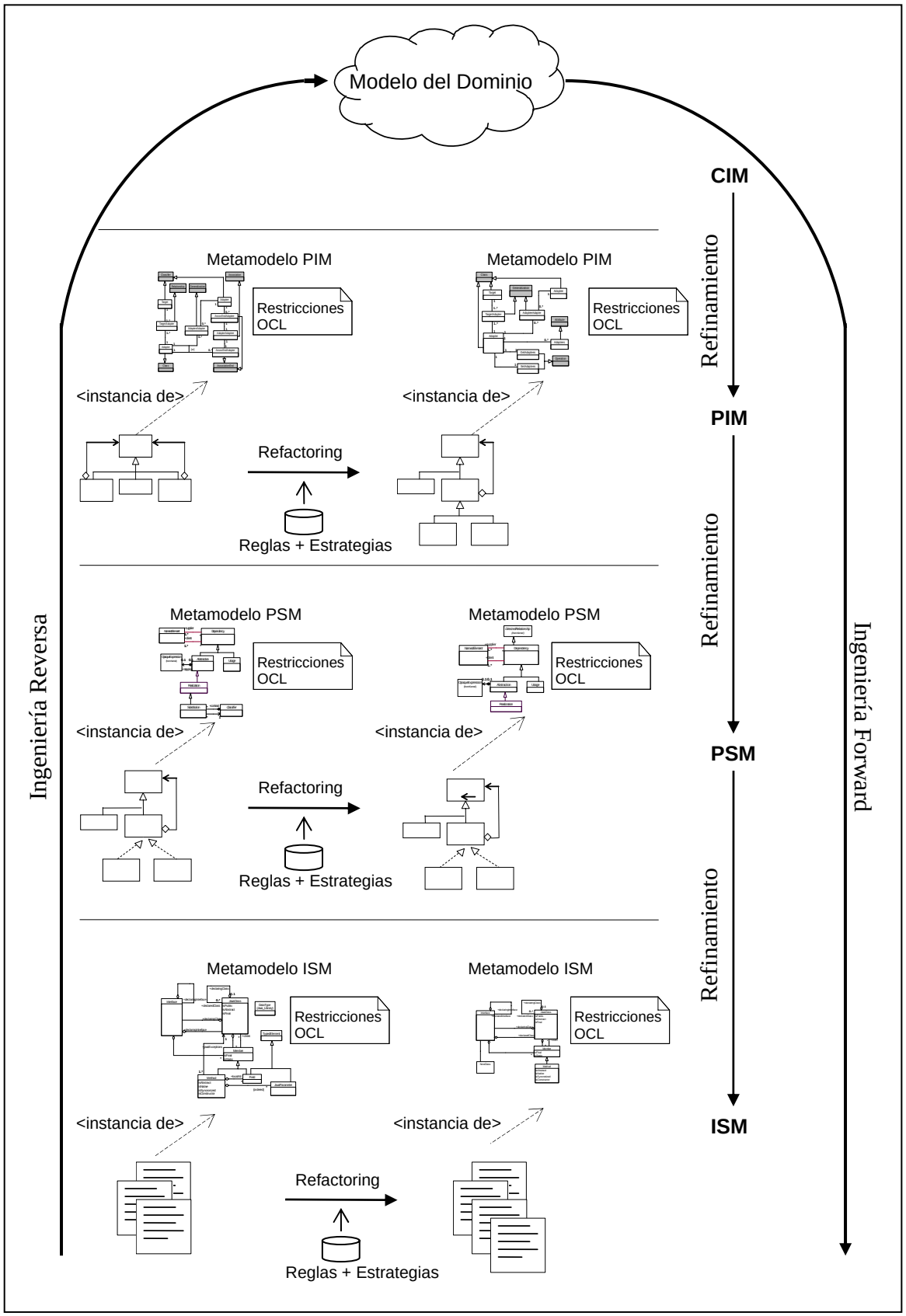


Figura 3.2 – Refactoring en MDD

- **Refactorings**, transformaciones de modelos de diseño que se aplican sobre un modelo en un nivel de abstracción dado generando un nuevo modelo en el mismo nivel. Los modelos origen y destino son instancias del mismo metamodelo MOF.
  - PIM → PIM, describe cómo un PIM es transformado en otro PIM teniendo en cuenta características que son independientes de cualquier plataforma y tecnología de implementación.
  - PSM → PSM: describe cómo un PSM es transformado en otro PSM teniendo en cuenta características de una plataforma particular.
  - ISM → ISM: describe cómo un ISM es transformado en otro ISM teniendo en cuenta una plataforma y lenguaje de programación particulares.
- **Refinamientos**, transformaciones de modelos en distintos niveles de abstracción:
  - CIM → PIM: describe cómo un PIM (instancia de un metamodelo MOF) es transformado en un PIM (instancia de un metamodelo MOF).
  - PIM → PSM: describe cómo un PIM es transformado en un PSM (instancia de un metamodelo MOF especializado para una plataforma específica),
  - PSM → ISM: describe cómo un PSM es transformado en un ISM (instancia de un metamodelo MOF especializado para una plataforma y lenguaje específicos).

La ingeniería forward, dentro del contexto de MDA, consiste en el proceso de transformación de modelos que va desde modelos independientes de la computación a modelos dependientes de la implementación. Los refactorings son importantes para reestructurar los modelos generados en cada una de las etapas del proceso. Análogamente, los refactorings tienen la misma importancia en el proceso inverso, es decir, en el proceso de ingeniería reversa, donde se construyen modelos de alto nivel de abstracción a partir de modelos de bajo nivel de abstracción. También resulta una técnica provechosa en los procesos de reingeniería de software donde se parte de modelos de código dependientes de una plataforma específica y se pretende migrarlo hacia otra plataforma construyendo modelos más abstractos durante este proceso.

Los distintos modelos generados en cada una de las etapas de los procesos de desarrollo pueden describirse por medio de un lenguaje de modelado. El estándar propuesto por OMG más ampliamente usado por la comunidad de desarrollo de software orientado a objetos es UML. En esta tesis se muestran los refactorings aplicados a modelos de diagramas de clases UML. La aplicación de refactorings sobre estos modelos permite la evolución de los mismos mejorando factores de calidad como extensibilidad, modularidad, reusabilidad, complejidad y legibilidad. Las reestructuraciones de modelos tienen que ver con el agregado, movimiento o eliminación de elementos de un modelo, como por ejemplo, clases, atributos y asociaciones, con la incorporación de patrones de diseño, y en modelos dependientes de una plataforma específica se aplican refactorings que tratan de mejorar cuestiones vinculadas a dicha plataforma.

A continuación se describe la técnica de especificación de refactorings (apartados 3.2 y 3.3) utilizada en esta tesis. La aplicación de esta técnica se muestra sobre modelos de diseño propuestos por MDA, más precisamente, sobre diagramas de clases UML en los tres niveles de abstracción: PIM, PSM e ISM.

3.2 Especificación de refactorings basada en metamodelos

Las transformaciones de modelos se aplican sobre un modelo fuente y producen un modelo destino. Estas transformaciones se definen por medio de contratos que establecen las restricciones para su aplicación y la especificación del resultado sin dar detalles de cómo serán implementadas. Se propone aprovechar los beneficios del diseño por contrato para el diseño de software confiable (Meyer, 1992). En el contexto de transformaciones de modelos los contratos son usados para su especificación y validación y se describen a través de:

- un conjunto de restricciones que deberá *matchear* el modelo fuente para poder ser transformado,
- un conjunto de restricciones que deberá *matchear* el modelo destino para ser considerado válido como producto de la transformación,
- un conjunto de restricciones que expresan las relaciones entre los elementos del modelo fuente y los elementos del modelo destino.

Se definirán las restricciones sobre los elementos de los modelos fuente y destino mediante técnicas de metamodelado. Los elementos que conforman un modelo y sus relaciones se describen mediante un metamodelo, es decir, un modelo de los modelos capaz de ser interpretado automáticamente por un computador. Un metamodelo contiene clases (metaclases), cada una de ellas define un elemento que puede existir en los modelos. Un metamodelo, además, describe relaciones entre metaclases que especifican las relaciones que deben existir entre los elementos de los modelos. La figura 3.3 muestra la relación entre un modelo y su correspondiente metamodelo. Los modelos M y M' son instancias de los correspondientes metamodelos. Esto significa, que un modelo fuente para ser un modelo al cual se le pueda aplicar una transformación debe conformar su metamodelo, y un modelo destino para ser modelo válido como resultado de la transformación debe conformar su metamodelo. El metamodelo fuente define una familia de modelos fuente a los cuales puede aplicarse la regla de transformación y el metamodelo destino caracteriza a los modelos generados.

Las restricciones que expresan las relaciones entre los elementos del modelo fuente y los elementos del modelo destino, especifican la transformación entre modelos y se expresan mediante reglas de transformación basadas en metamodelos. Estas reglas relacionan la metaclase del elemento del modelo fuente con la/s metaclase/s de los elementos del modelo destino. Debido a la relación instancia-tipo entre los elementos del modelo y las metaclases, cada ocurrencia de la metaclase en el modelo fuente debería conformar las restricciones establecidas para su metaclase. Estas restricciones o reglas de transformación constituirán la definición de transformación T'.

En el contexto MDA, los metamodelos se describen usando MOF, estándar de OMG que captura la diversidad de estándares de modelado para integrar diferentes tipos de modelos y metadatos e intercambiarlos entre diferentes herramientas. Sus cuatro construcciones de modelado son: clases, asociaciones, tipos de datos y paquetes. Por cada transformación de modelo (refactoring) presentada en esta tesis, se especifica un metamodelo fuente y un metamodelo destino como especialización del metamodelo UML. A continuación se detalla la especificación de la regla de transformación de modelos basada en metamodelos expresada como contrato OCL.

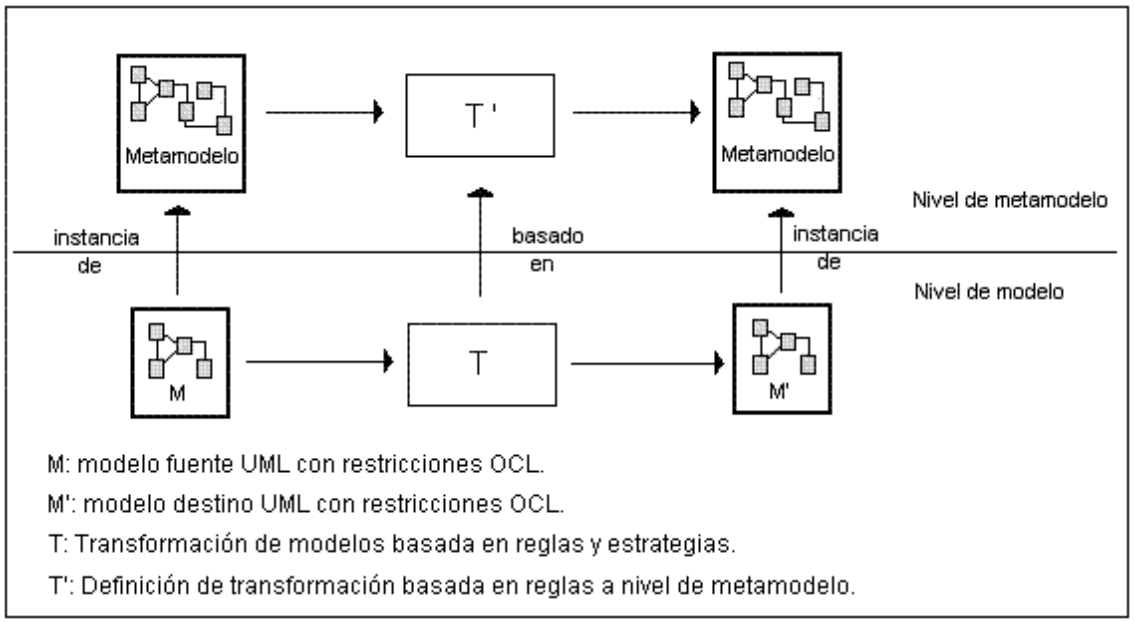


Figura 3.3 – Definición de transformaciones

3.3 Especificación de refactorings como contratos OCL

En los capítulos siguientes se presentarán ejemplos de especificaciones de refactorings de modelos en distintos niveles de abstracción. Cada refactoring se presenta en un formato que consta de cinco partes:

- **nombre** del refactoring;
- **descripción**, breve explicación de la situación en la cual se puede aplicar dicho refactoring y del resultado de su aplicación;
- **motivación**, describe el motivo de la aplicación del refactoring y bajo qué circunstancias no debería aplicarse;
- **metamodelo origen,** describe la familia de modelos a los cuales puede aplicarse el refactoring;
- **metamodelo destino**, describe a los modelos generados por la transformación;
- **regla de transformación**, describe las condiciones que deben cumplirse para la aplicación del refactoring y la relación entre los elementos del modelo origen y los del modelo resultante;
- **ejemplo,** muestra una aplicación del refactoring.

Las reglas de transformación son expresadas como contratos OCL con una notación sencilla, existiendo una brecha sintáctica con el estándar QVT de OMG para definir transformaciones. La propuesta está alineada con este estándar ya que QVT está basado en el package CORE que depende de Essential MOF (MOF, 2006) y Essential OCL (OCL, 2006).

Cada regla de refactoring comienza con la palabra clave **Transformation** seguida del nombre de la regla. A continuación se detalla el cuerpo de la regla delimitado por llaves.

Los parámetros de la regla se denotan mediante una lista de declaración de variables seguida a la palabra clave **parameter**. Los parámetros serán del tipo de alguna metaclase del metamodelo UML.

Después de las palabras claves **local operations** se definirán un conjunto de operaciones que serán usadas en el cuerpo de dicha regla.

Las expresiones booleanas OCL que aparecen después de la palabra clave **preconditions**, se refieren a las restricciones que debe mantener el modelo fuente para que la regla pueda ser aplicada.

Las expresiones booleanas OCL que aparecen después de la palabra clave **postconditions**, se refieren al resultado de aplicar la regla, relacionando elementos del modelo fuente con elementos del modelo destino.

Las operaciones locales, precondiciones y postcondiciones del cuerpo de la regla escritas en letra cursiva reflejan operaciones y/o restricciones sobre elementos que pueden no estar presentes en un modelo. Por ejemplo, en la especificación de un refactoring a nivel PIM cuyo modelo puede no tener asociado aún el código correspondiente, en la especificación de la regla cualquier referencia a atributos o invocaciones a métodos en el cuerpo de un método se escriben en cursiva reflejando que pueden no estar presente en dicho nivel de abstracción, pero si están presentes tiene que valer lo especificado en la regla.

La forma general para la notación de la regla es la siguiente:

Transformation *nombre-transformación* **{**

parameter

<*lista-de-parámetros*>

local operations

<lista-de-*Expresiones-OCL*>

preconditions

< lista-de-*Expresiones-OCL*>

postconditions

< lista-de-*Expresiones-OCL*>

}

3.4 Clasificación de refactorings

En los capítulos siguientes se presenta una clasificación de refactorings basada en los distintos niveles de abstracción de la arquitectura MDA. Se proponen refactorings aplicables a modelos independientes de la plataforma, a modelos dependientes de la plataforma y a modelos dependientes de la implementación.

En el capítulo 4 se muestra un refactoring a nivel PIM, *Extraer Composite*, donde las reestructuraciones más importantes que provoca la aplicación de este refactoring se dan sobre elementos que están presentes en este nivel de abstracción, como lo son las

agregaciones y jerarquías de herencia. Si bien este refactoring analiza y reestructura los elementos mencionados anteriormente, se especifica el impacto que estas reestructuraciones causan en los demás niveles de abstracción, por ejemplo, sobre el nivel de código. La regla de refactoring contempla estos cambios en las postcondiciones, las cuales se muestran en letra cursiva indicando que hacen referencia a elementos que pueden no estar presentes en ese nivel de abstracción. Esto permite mantener los modelos consistentes y coherentes.

En el capítulo 5 se especifican refactorings aplicables a modelos que dependen de una plataforma específica. Se muestra un refactoring para modelos específicos de la plataforma C++ que trata de evitar los problemas que ocasiona la herencia múltiple permitida en esta plataforma. Para esta reestructuración se analizan las jerarquías de herencia y el tipo de herencia entre las clases que intervienen, elementos presentes en este nivel de abstracción. Además, se muestra un refactoring para modelos específicos de la plataforma Java, que permite mover a una interfaz el comportamiento común de varias clases. Para este refactoring se analizan elementos tales como clases y signaturas de operaciones presentes en este nivel de abstracción. Este refactoring refleja una práctica ventajosa y fomentada en la comunidad de diseñadores bajo la plataforma Java.

El capítulo 6 muestra refactorings para modelos vinculados a código. El refactoring dependiente de la implementación C++ trata de minimizar dependencias de compilación entre archivos. Este refactoring apunta a fomentar el uso de una estrategia simple para los desarrolladores C++. Esta consiste en utilizar referencias y punteros a objetos en vez de objetos, utilizar declaraciones de clases y proveer archivos *headers* separados para declaraciones y definiciones de clases. El refactoring a nivel de implementación Java sugiere que cuando una clase que implementa una interfaz provee código sólo para algunos de los métodos de la interfaz, se reestructure el modelo utilizando el patrón *Adapter*. Los refactorings para ambas plataformas tratan con elementos que están presentes en el nivel vinculado a código.

En (Pereira y Favre, 2006) se presenta una propuesta para el refactoring de modelos estáticos UML en el contexto de MDA. Se muestra una clasificación de refactorings a nivel PIM, PSM e ISM y se describe la especificación de los mismos como contratos OCL basados en metamodelos.

CAPÍTULO 4

REFACTORING A NIVEL PIM

Para la definición de reglas de transformación en los distintos niveles de MDA se necesita especificar metamodelos correspondientes a cada nivel, ya que estas reglas imponen relaciones entre un metamodelo fuente 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.

Las ideas centrales de este capítulo ilustradas con el ejemplo que se detalla a continuación fueron publicadas en (Favre y Pereira, 2007).

4.1 Metamodelos a nivel PIM

Para cada refactoring aplicable a nivel PIM se especifican los metamodelos origen y destino como especializaciones del metamodelo UML. Estos metamodelos son complementados con restricciones OCL.

La especificación de UML (versión 2.0) incluye el metamodelo correspondiente a varios de los diagramas que se pueden representar con este lenguaje. Los ejemplos de esta tesis se basan en el metamodelo de Diagramas de Clases.

En el Anexo A, los diagramas más relevantes del paquete *Kernel* muestran parte de la sintaxis abstracta del metamodelo UML. Este paquete representa los conceptos de modelado básicos de los diagramas de clases UML. El metamodelo del diagrama de clases 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: generalización, asociación, agregación y composición además de la definición de invariantes o restricciones sobre los modelos de clases.

4.2 Ejemplo: Extraer *Composite*

Este refactoring está basado en el autor Joshua Kerievsky, quien en su libro *Refactoring To Patterns* (Kerievsky, 2004) propone un refactoring llamado *Extract Composite* aplicable a código Java.

4.2.1 Descripción

Cuando en una jerarquía de herencia dos o más subclases almacenan hijos de la misma jerarquía y tienen métodos duplicados que operan sobre los hijos, la aplicación de la regla *Extraer Composite* permite crear una superclase *Composite* y mover campos y métodos duplicados desde las subclases a la superclase. La figura 4.1 ejemplifica este refactoring.

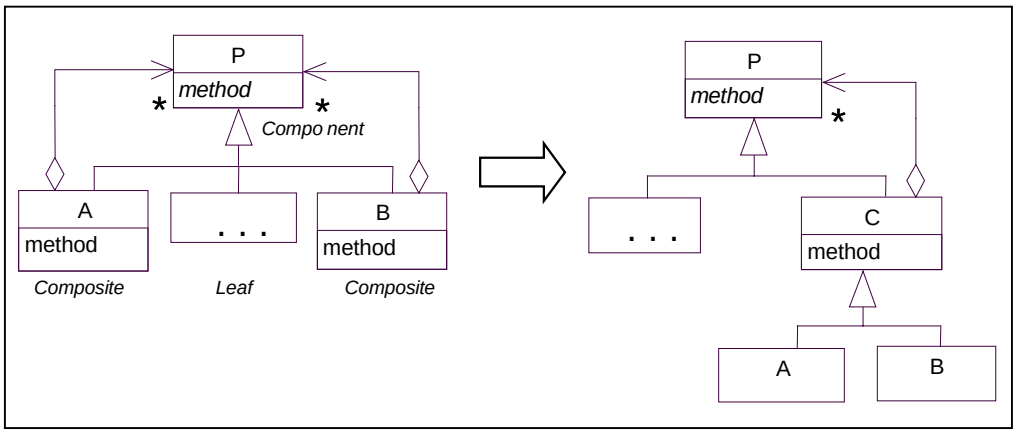


Figura 4.1 – Refactoring Extraer *Composite*

4.2.2 Motivación

En una jerarquía de herencia donde subclases almacenan colecciones de hijos que son clases en la misma jerarquía y poseen métodos para el tratamiento de dicha colección existe duplicación de métodos. La aplicación de esta regla permite simplificar las subclases eliminando la duplicación de métodos al factorizar en una superclase el comportamiento común.

4.2.3 Metamodelo origen

Sintaxis abstracta

La figura 4.2 del metamodelo origen muestra las metaclases relacionadas con los participantes esenciales en los modelos a los cuales puede aplicarse el refactoring *Extraer Composite*:

- *Composite*, *Component* y *Leaf*, que representan clases de un modelo con características particulares,
- *CompositeComponentAssoc*, *ComponentLeafGeneralization* y *ComponentCompositeGeneralization*, que representan relaciones particulares entre las clases anteriores,
- *AssEndComposite* y *AssEndComponent*, que representan propiedades de las clases,
- metaclases sombreadas en gris que corresponden al metamodelo UML.

El metamodelo origen 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*.

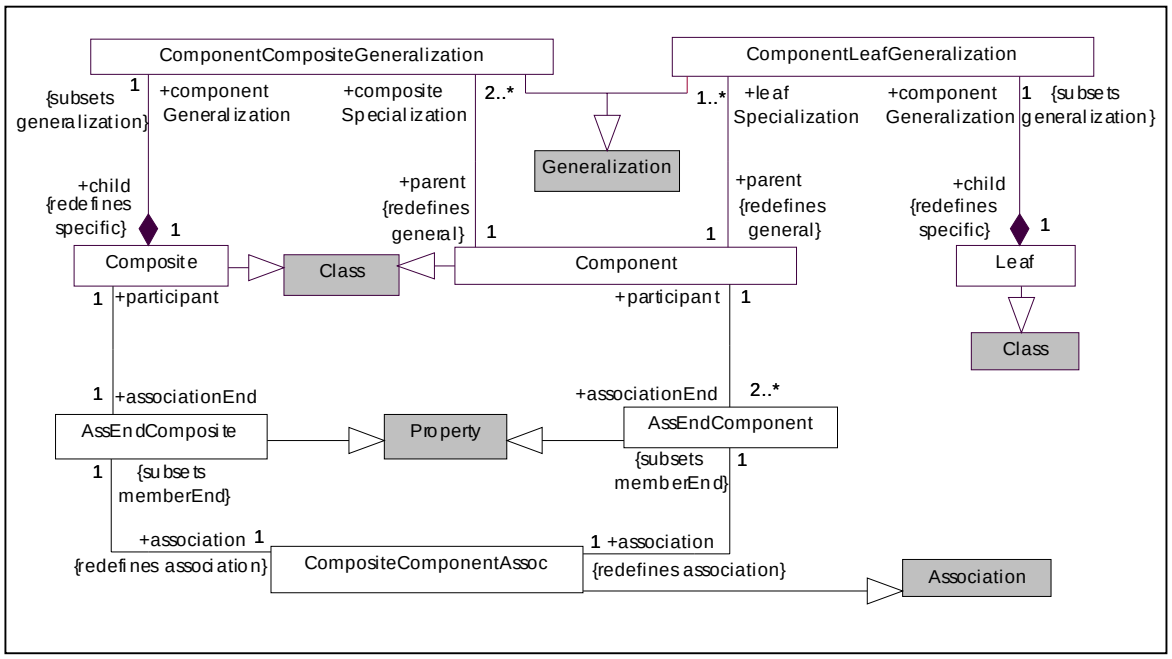


Figura 4.2 – Metamodelo origen del refactoring *Extraer Composite*

Descripciones de metaclases

AssEndComponent

Describe la conexión de la asociación *CompositeComponentAssoc* con la clase *Component*.

Generalizaciones:

- Property, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- association: CompositeComponentAssoc [1]

  Referencia a la asociación que conecta a las clases *Component* y *Composite*. Redefine *Property::association*.

- participant: Component [1]

  Designa a la clase que participa en la asociación en dicho extremo.

Restricciones

No posee restricciones adicionales.

AssEndComposite

Describe la conexión de la asociación *CompositeComponentAssoc* con la clase *Composite*.

Generalizaciones:

- Property, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- association: CompositeComponentAssoc [1]

  Referencia a la asociación que conecta a las clases *Component* y *Composite*. Redefine *Property::association*.

- participant: Composite [1]

  Designa a la clase que participa en la asociación en dicho extremo.

Restricciones

[1] El extremo de asociación es una agregación o una composición.

```
self.aggregation = #shared  or  self.aggregation = #composite
```

Component

Representa una clase que posee las características de una clase *Component* en una jerarquía de herencia.

Generalizaciones:

- Class, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- associationEnd: AssEndComponent [2..*]

  Designa al conjunto de extremos de asociaciones, en los cuales la clase *Component* participa, pertenecientes a asociaciones que conectan con la clase *Composite*.

- compositeSpecialization: ComponentCompositeGeneralization [2..*]

  Especifica el conjunto de generalizaciones en las cuales *Component* es la superclase y una clase *Composite* es la subclase.

- leafSpecialization: ComponentLeafGeneralization [1..*]

  Especifica el conjunto de generalizaciones en las cuales *Component* es la superclase y una clase *Leaf* es la subclase.

Restricciones

[1] Las asociaciones entre *Composite* y *Component* son equivalentes.

```
self.associationEnd → collect (assEnd | assEnd.association) → forAll ( a1, a2 |
            a1 = a2  or
            a1.isEquivalentTo(a2) )
```

[2] En cada clase *Composite*, subclase de *Component*, existen operaciones funcionalmente equivalentes a operaciones de otras clases *Composite*. La operación *isEquivalentOperationTo()* verifica si la operación pasada como argumento es equivalente a la operación a la cual se aplica la operación, es decir, son equivalentes si sus signaturas *matchean* y semánticamente coinciden a través de un *matching* exacto o plug-in. Estos matcheos han sido tomados y adaptados del trabajo de Zaremski y Wing (1997). La operación *isEquivalentOperationTo()* está especificada como operación adicional de la clase Operation del metamodelo UML (ver Anexo A).

```
-- En cada clase Composite, subclase de Component
self.compositeSpecialization.child  →  forAll ( class |
   -- existen operaciones
   class.ownedOperation → exists ( op |
      self.compositeSpecialization.child → excluding (class) →  forAll ( c |
            c.ownedOperation → exists ( o |
                  -- funcionalmente equivalentes a operaciones de otras clases
                  -- Composite
                   op.isEquivalentOperationTo(o) ))))
```

Operaciones locales

CompositeComponentAssoc::isEquivalentTo(a: CompositeComponentAssoc): Boolean;

-- Verifica si la asociación pasada como argumento es equivalente a la asociación a la
-- cual se aplica la operación, es decir, son equivalentes si, salvo el nombre, las demás
-- características se conservan.

```
isEquivalentTo (a) =

   (self.name = a.name or self.name <> a.name) and
   self.isDerived = a.isDerived and
   self.visibility = a.visibility  and
   self.memberEnd → forAll (a1End |
      a.memberEnd → exists (a2End |
         (a1End.name = a2End.name or a1End.name <> a2End.name) and
         a1End.visibility = a2End.visibility  and
         a1End.isLeaf = a2End.isLeaf  and
         a1End.isStatic = a2End.isStatic  and
         a1End.isDerived = a2End.isDerived  and
         a1End.isReadOnly = a2End.isReadOnly  and
         a1End.isDerivedUnion = a2End.isDerivedUnion  and
         a1End.aggregation = a2End.aggregation  and
         a1End.upper = a2End.upper and
         a1End.lower = a2End.lower  and
         a1End.subsettedProperty = a2End.subsettedProperty and
         a1End.redefinedProperty = a2End.redefinedProperty  ))
```

ComponentCompositeGeneralization

Representa a la relación de generalización existente entre las clases *Component* y *Composite*.

Generalizaciones:

- Generalization, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- parent: Component [1]

  Referencia a la clase que cumple el rol de padre en la relación de generalización. Redefine *Generalization::general*.

- child: Composite [1]

  Referencia a la clase que cumple el rol de hija en la relación de generalización. Redefine *Generalization::specific*.

Restricciones

No posee restricciones adicionales.

ComponentLeafGeneralization

Representa a la relación de generalización existente entre las clases *Component* y *Leaf*.

Generalizaciones:

- Generalization, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- parent: Component [1]

  Referencia a la clase que cumple el rol de padre en la relación de generalización. Redefine *Generalization::general*.

- child: Leaf [1]

  Referencia a la clase que cumple el rol de hija en la relación de generalización. Redefine *Generalization::specific*.

Restricciones

No posee restricciones adicionales.

Composite

Representa una clase que posee las características de una clase *Composite* en una jerarquía de herencia.

Generalizaciones:

- Class, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- associationEnd: AssEndComposite [1]

  Designa al conjunto de extremos de asociaciones, en los cuales la clase *Composite* participa, pertenecientes a asociaciones que conectan con la clase *Component*.

- componentGeneralization: ComponentCompositeGeneralization [1]

  Especifica el conjunto de generalizaciones en las cuales *Composite* es la subclase y una clase *Component* es la superclase. Subconjunto de *Classifier::generalization.*

Restricciones

No posee restricciones adicionales.

CompositeComponentAssoc

Describe una asociación binaria que relaciona las metaclases *Composite* y *Component*.

Generalizaciones:

- Association, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- assEndComposite: AssEndComposite [1]

  Referencia al extremo de asociación vinculado a *Composite* de la asociación *CompositeComponentAssoc*. Subconjunto de *Association::memberEnd*.

- assEndComponent: AssEndComponent [1]

  Referencia al extremo de asociación vinculado a *Component* de la asociación *CompositeComponentAssoc*. Subconjunto de *Association::memberEnd*.

Restricciones

[1] *CompositeComponentAssoc* es una asociación binaria, es decir, los únicos extremos miembros de la asociación son *assEndComposite* y *assEndComponent*.

self.memberEnd → size () = 2

Leaf

Representa una clase que es una clase hoja en la jerarquía de herencia.

Generalizaciones:

- Class, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- componentGeneralization: ComponentLeafGeneralization [1]

  Referencia a la relación de generalización donde *Leaf* es subclase de una clase *Component*. Subconjunto de *Classifier::generalization*.

Restricciones

No posee restricciones adicionales.

4.2.4 Metamodelo destino

Sintaxis abstracta

La figura 4.3 del metamodelo destino muestra las metaclases relacionadas con los participantes esenciales en los modelos generados al aplicarse el refactoring *extraer composite*. Las diferencias entre los metamodelo origen y destino sugieren que:

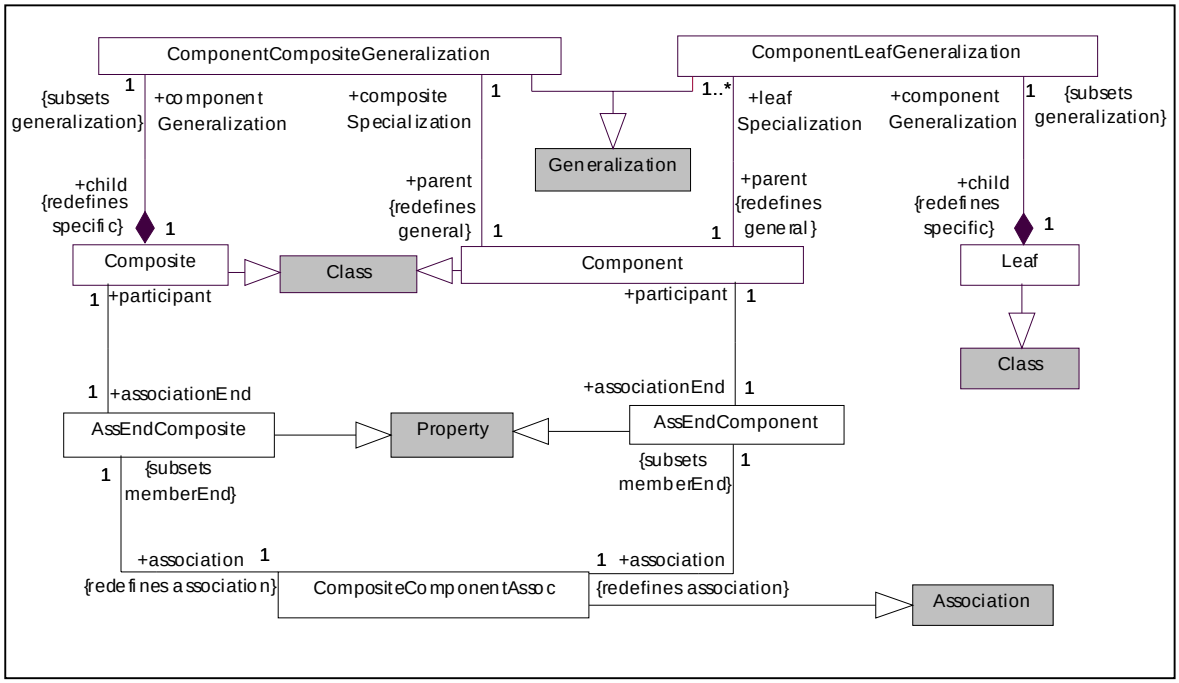


Figura 4.3 – Metamodelo destino del refactoring *Extraer Composite*

- en un modelo origen, una instancia de *Component* tiene dos o más instancias de *ComponentCompositeGeneralization* (*compositeSpecialization*) y dos o más extremos de asociación (*associationEnd*);
- en un modelo destino, una instancia de *Component* tiene exactamente una instancia de *ComponentCompositeGeneralization* y un único extremo de asociación de tipo *AssEndComponent*.

Descripciones de metaclases

AssEndComponent

Describe la conexión de la asociación *CompositeComponentAssoc* con la clase *Component*.

Generalizaciones:

- Property, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- association: CompositeComponentAssoc [1]

  Referencia a la asociación que conecta a las clases *Component* y *Composite*. Redefine *Property::association*.

- participant: Component [1]

  Designa a la clase que participa en la asociación en dicho extremo.

Restricciones

No posee restricciones adicionales.

AssEndComposite

Describe la conexión de la asociación *CompositeComponentAssoc* con la clase *Composite*.

Generalizaciones:

- Property, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- association: CompositeComponentAssoc [1]

  Referencia a la asociación que conecta a las clases *Component* y *Composite*. Redefine *Property::association*.

- participant: Composite [1]

  Designa a la clase que participa en la asociación en dicho extremo.

Restricciones

[1] El extremo de asociación es una agregación o una composición.

```
self.aggregation = #shared  or  self.aggregation = #composite
```

Component

Representa una clase que posee las características de una clase *Component* en una jerarquía de herencia.

Generalizaciones:

- Class, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- associationEnd: AssEndComponent [1]

  Designa al extremo de asociación, en el cual la clase *Component* participa, perteneciente a la asociación que conecta con la clase *Composite*.

- compositeSpecialization: ComponentCompositeGeneralization [1]

  Especifica la generalización en la cual *Component* es la superclase y una clase *Composite* es la subclase.

- leafSpecialization: ComponentLeafGeneralization [1..*]

  Especifica el conjunto de generalizaciones en las cuales *Component* es la superclase y una clase *Leaf* es la subclase.

Restricciones

No posee restricciones adicionales.

ComponentCompositeGeneralization

Representa a la relación de generalización existente entre las clases *Component* y *Composite*.

Generalizaciones:

- Generalization, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- parent: Component [1]

  Referencia a la clase que cumple el rol de padre en la relación de generalización. Redefine *Generalization::general*.

- child: Composite [1]

  Referencia a la clase que cumple el rol de hija en la relación de generalización. Redefine *Generalization::specific*.

Restricciones

No posee restricciones adicionales.

ComponentLeafGeneralization

Representa a la relación de generalización existente entre las clases *Component* y *Leaf*.

Generalizaciones:

- Generalization, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- parent: Component [1]

  Referencia a la clase que cumple el rol de padre en la relación de generalización. Redefine *Generalization::general*.

- child: Leaf [1]

  Referencia a la clase que cumple el rol de hija en la relación de generalización. Redefine *Generalization::specific*.

Restricciones

No posee restricciones adicionales.

Composite

Representa una clase que posee las características de una clase *Composite* en una jerarquía de herencia.

Generalizaciones:

- Class, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- associationEnd: AssEndComposite [1]

  Designa al conjunto de extremos de asociaciones, en los cuales la clase *Composite* participa, pertenecientes a asociaciones que conectan con la clase *Component*.

- componentGeneralization: ComponentCompositeGeneralization [1]

Especifica el conjunto de generalizaciones en las cuales *Composite* es la subclase y una clase *Component* es la superclase. Subconjunto de *Classifier::generalization.*

Restricciones

No posee restricciones adicionales.

CompositeComponentAssoc

Describe una asociación binaria que relaciona las metaclases *Composite* y *Component*.

Generalizaciones:

- Association, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- assEndComposite: AssEndComposite [1]

  Referencia al extremo de asociación vinculado a *Composite* de la asociación *CompositeComponentAssoc*. Subconjunto de *Association::memberEnd*.

- assEndComponent: AssEndComponent [1]

  Referencia al extremo de asociación vinculado a *Component* de la asociación *CompositeComponentAssoc*. Subconjunto de *Association::memberEnd*.

Restricciones

[1] *CompositeComponentAssoc* es una asociación binaria, es decir, los únicos extremos miembros de la asociación son *assEndComposite* y *assEndComponent*.

self.memberEnd → size () = 2

Leaf

Representa una clase que es una clase hoja en la jerarquía de herencia.

Generalizaciones:

- Class, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- componentGeneralization: ComponentLeafGeneralization [1]

  Referencia a la relación de generalización donde *Leaf* es subclase de una clase *Component*. Subconjunto de *Classifier::generalization*.

Restricciones

No posee restricciones adicionales.

4.2.5 Regla de transformación

La regla *Extraer Composite* se aplica sobre una jerarquía de herencia con dos o más subclases (*Composite*) que almacenan hijos de la misma jerarquía y tienen operaciones duplicadas que operan sobre los hijos. La aplicación de esta regla crea una superclase (*Composite*) y mueve los atributos y operaciones equivalentes de las subclases a la superclase. La jerarquía de herencia se simplifica al factorizar en una superclase el comportamiento común.

Los pasos para aplicar esta regla se enumeran a continuación.

1. Crear una superclase abstracta *Composite*.
2. Reflejar como subclase de esta nueva clase, a cada clase de la jerarquía original que contiene atributos y operaciones equivalentes para administrar sus hijos (subclases de la misma jerarquía).
3. Detectar las operaciones equivalentes de las clases de la jerarquía original y moverlas a la superclase *Composite*.
4. Detectar los atributos equivalentes y moverlos a la superclase *Composite*. Renombrar los atributos si fuera necesario para que tengan sentido en la nueva jerarquía y actualizar cada referencia al mismo en las clases.
5. Chequear las clases clientes de las clases *Composite* de la jerarquía original para que puedan comunicarse a través de la nueva interfaz.

A continuación, se especifican estos cambios a través de la regla de transformación.

Transformation Extraer Composite **{**

parameter
source: Metamodelo Origen Extraer *Composite*:: Package
target: Metamodelo Destino Extraer *Composite*:: Package

local operations

postconditions
post
-- Para toda clase *sourceClass,* instancia de *Component,* en el paquete *source,*
source.ownedMember -> select(oclIsTypeOf(Component)) ->forAll (sourceClass |

-- existe una clase *targetClass*, instancia de *Component* en el paquete *target* tal que,
target.ownedMember -> select(oclIsTypeOf(Component)) -> exists (targetClass |

-- *targetClass* tiene sólo una relación de generalización del tipo
-- *ComponentCompositeGeneralization*, donde *targetClass* es padre de
-- una clase del tipo *Composite*,
targetClass.oclAsType(Component).compositeSpecialization -> size() =1 and

-- *targetClass* tiene un sólo extremo de asociación que la relaciona con una
-- clase del tipo *Composite*,
targetClass.oclAsType(Component).associationEnd -> size() =1 and

```
-- targetClass tiene como subclases el mismo conjunto de clases de tipo Leaf
-- que sourceClass,
targetClass.oclAsType(Component).leafSpecialization.child =
    sourceClass. oclAsType(Component).leafSpecialization.child and

-- atributos heredados de NamedElement
targetClass.name = sourceClass.name and
targetClass.visibility = sourceClass.visibility and

-- targetClass  tiene los siguientes valores para:
-- atributo heredado de RedefinableElement
targetClass.oclAsType(Class).isLeaf = sourceClass.isLeaf and

-- attributo heredado de Classifier
targetClass.oclAsType(Class).isAbstract = sourceClass.isAbstract and

-- asociaciones heredadas de Class
-- targetClass  tiene el mismo conjunto de clasificadores anidados que
-- sourceClass
targetClass.oclAsType(Class).nestedClassifier =
    sourceClass.oclAsType(Class).nestedClassifier   and

-- targetClass  tienen el mismo conjunto de operaciones propias que
-- sourceClass
targetClass.oclAsType(Class).ownedOperation =
    sourceClass.oclAsType(Class).ownedOperation   and

-- el conjunto de atributos propios de targetClass es igual al conjunto de
-- atributos propios de sourceClass,
-- menos los atributos que son extremos de asociación que la relacionan
-- con las diversas clases de tipo Composite del paquete source,
-- más el extremo de asociación que la vincula con una única clase
-- Composite del paquete target,
targetClass.oclAsType(Class).ownedAttribute =
sourceClass.oclAsType(Class).ownedAttribute
-> excluding(sourceClass.oclAsType(Component).
    associationEnd.association.assEndComposite)
-> including (targetClass. oclAsType(Component).
    associationEnd.association.assEndComposite)  and

-- asociaciones heredadas de Classifier
-- ambas clases tienen el mismo conjunto de generalizaciones
targetClass.oclAsType(Class).generalization =
    sourceClass.oclAsType(Class).generalization   and

-- ambas clases pertenecen al mismo package
targetClass.oclAsType(Class).package =
    sourceClass.oclAsType(Class).package   and
```

```
        -- ambas clases tienen el mismo conjunto de clasificadores redefinidos
        targetClass.oclAsType(Class).redefinedClassifier =
            sourceClass.oclAsType(Class).redefinedClassifier  and

        -- asociaciones heredadas de Namespace
        -- ambas clases tienen las mismas restricciones
        targetClass.oclAsType(Class).ownedRule =
            sourceClass.oclAsType(Class).ownedRule   and

        -- ambas clases tienen el mismo conjunto de elementos importados
        targetClass.oclAsType(Class).elementImport =
            sourceClass.oclAsType(Class).elementImport  ))
```

post

```
    -- Para toda clase sourceClass, instancia de Composite, en el paquete source,
    source.ownedMember -> select(oclIsTypeOf(Composite)) -> forAll ( sourceClass |

        -- existe una clase targetClass, instancia de Class, en el paquete target tal que,
        target.ownedMember ->  select(oclIsTypeOf(Class)) -> exists ( targetClass |

            -- la clase del paquete target (targetClass) tiene como padre a una
            -- clase de tipo Composite,
            targetClass.oclAsType(Class).superClass.oclIsTypeOf(Composite)) and

            -- targetClass  tiene los siguientes valores para:
            -- atributo heredado de RedefinableElement
            targetClass.oclAsType(Class).isLeaf =
                sourceClass.oclAsType(Composite).isLeaf and

            -- atributos heredados de NamedElement
            targetClass.name = sourceClass.name and
            targetClass.visibility = sourceClass.visibility and

            -- attributo heredado de Classifier
            targetClass.oclAsType(Class).isAbstract =
                sourceClass.oclAsType(Composite).isAbstract and

            -- Por cada operación de sourceClass

            sourceClass.oclAsType(Composite).ownedOperation -> forAll( op  |
            -- equivalente a operaciones de las demás clases de tipo Composite
            -- en el paquete source

                (source.ownedMember ->  select(oclIsTypeOf(Composite))
                -> excluding(sourceClass)) -> collect
                    (oclAsType(Composite).ownedOperation) -> forAll ( o |
```

```
if o.isEquivalentOperationTo(op)
-- la operación isEquivalentOperationTo() está especificada como
-- operación adicional de la clase Operation del metamodelo UML
-- (ver Anexo A).
then
-- si las operaciones o y op son equivalentes entonces,
-- una operación equivalente existirá en la clase
-- padre de targetClass en el paquete target
targetClass.oclAsType(Class).superClass.ownedOperation->
    exists (targetOp |
        op.isEquivalentOperationTo (targetOp)) and
-- y dicha operación será excluida de targetClass
targetClass.oclAsType(Class).ownedOperation -> excludes(op) and
-- las operaciones referenciadas directa o indirectamente por
-- op que tengan una operación equivalente referenciada  por
-- o se moverán a la clase padre de targetClass
op.referencedOperation -> forAll (refOp |
    o.referencedOperation -> forAll (refO |
    -- la operación referencedOperation está especificada como
    -- operación adicional de la clase Operation del metamodelo UML
    -- (ver Anexo A).
        if refOp.isEquivalentOperationTo(refO)
            then
            targetClass.oclAsType(Class).superClass.
            oclAsType(Class).ownedOperation->
                exists (targetOp |
                refOp.isEquivalentOperationTo (targetOp)) and
            -- y dicha operación será excluida de targetClass
            targetClass.oclAsType(Class).ownedOperation ->
                excludes(refOp)
            else
            -- si no son equivalentes pasarán a la clase padre de
            -- targetClass como operación abstracta.
            targetClass.oclAsType(Class).superClass.
            ownedOperation -> exists (targetOp |
            refOp.isEquivalentOperationTo (targetOp)  and
            targetOp.ownedMember-> select(oclIsKindOf(Action))
            ->size() = 0) ) )
else
-- si la operación de sourceClass (op) no tiene operaciones
-- equivalentes en las demás clases, será una operación de
-- targetClass en el paquete target
targetClass.oclAsType(Class).ownedOperation -> includes(op)
endif ))
```

```
-- Por cada atributo de sourceClass
sourceClass.oclAsType(Composite).ownedAttribute -> forAll( sa  |

-- equivalente a atributos de las demás clases de tipo Composite
-- en el paquete source
(source.ownedMember ->  select(oclIsTypeOf(Composite))
    ->excluding(sourceClass)).oclAsType(Composite).ownedAttribute
        -> forAll ( a |

            if a.isEquivalentPropertyTo(sa)
            -- la operación isEquivalentPropertyTo() está especificada como
            -- operación adicional de la clase Property del metamodelo UML
            -- (ver Anexo A).
            then
            -- si los atributos a y sa son equivalentes entonces,
            -- un atributo equivalente existirá en la clase
            -- padre de targetClass en el paquete target
            targetClass.oclAsType(Class).superClass.ownedAttribute ->
                exists (targetAt |
                    sa.isEquivalentPropertyTo (targetAt))  and
            -- y dicho atributo será excluido de targetClass
            targetClass.oclAsType(Class).ownedAttribute ->
                excludes(sa)  and

            -- en las implementaciones (si existen) de las operaciones de
            -- sourceClass y de su descendencia, toda referencia a la
            -- propiedad sa será reemplazada por la propiedad targetAt en
            -- la clase padre de targetClass y en su descendencia.
            sourceClass.oclAsType(Composite).referencedProperty
                -> includes (sa)

                implies
                targetClass.oclAsType(Class).superClass.
                    referencedProperty -> excludes(sa) and
                targetClass.oclAsType(Class).superClass.
                    referencedProperty -> includes(targetAt) and

            -- en las expresiones OCL, relacionadas con las invariantes de
            -- clase, postcondiciones, precondiciones y cuerpo de
            -- operaciones, de sourceClass o de una clase descendiente,
            -- toda referencia a la propiedad sa será
            -- reemplazada por la propiedad targetAt en
            -- la clase padre de targetClass y en su descendencia.
            sourceClass.oclAsType(Composite).referencedPropertyInOcl
            -> includes (sa)
                implies
                targetClass.oclAsType(Class).superClass.
                    referencedPropertyInOcl -> excludes(sa) and
                targetClass.oclAsType(Class).superClass.
                    referencedPropertyInOcl -> includes(targetAt)

            else
```

```
                    -- si el atributo de sourceClass (sa) no tiene atributos
                    -- equivalentes en las demás clases, será un atributo de
                    -- targetClass en el paquete target
                    targetClass.oclAsType(Class).ownedAttribute -> includes(sa)
                    endif ))   and

                -- ambas clases pertenecen al mismo package
                targetClass.oclAsType(Class).package =
                    sourceClass.oclAsType(Composite).package    and

                -- ambas clases tienen el mismo conjunto de clasificadores redefinidos
                targetClass.oclAsType(Class).redefinedClassifier =
                    sourceClass.oclAsType(Composite).redefinedClassifier  and

                -- ambas clases tienen el mismo conjunto de elementos importados
                targetClass.oclAsType(Class).elementImport =
                    sourceClass.oclAsType(Composite).elementImport  ))

post
    -- Para toda clase sourceClass en el paquete source,
    source.ownedMember -> select(oclIsTypeOf(Class)) -> forAll ( sourceClass |

        -- existe una clase targetClass en el paquete target tal que,
        target.ownedMember ->  select(oclIsTypeOf(Class)) -> exists ( targetClass |

            -- si sourceClass es cliente de alguna clase de tipo Composite
            if (sourceClass.oclAsType(Class).clientDependency -> exists ( c |
                    c.supplier.oclIsTypeOf(Composite)))
            then
            -- targetClass  tiene los siguientes valores para:
            -- atributo heredado de RedefinableElement
            targetClass.oclAsType(Class).isLeaf =
                sourceClass.oclAsType(Class).isLeaf and

            -- atributos heredados de NamedElement
            targetClass.name = sourceClass.name and
            targetClass.visibility = sourceClass.visibility and

            -- attributo heredado de Classifier
            targetClass.oclAsType(Class).isAbstract =
                sourceClass.oclAsType(Class).isAbstract and

            -- asociaciones heredadas de Class
            -- targetClass  tiene el mismo conjunto de clasificadores anidados
            -- que sourceClass
            targetClass.oclAsType(Class).nestedClassifier =
                sourceClass.oclAsType(Class).nestedClassifier   and

            -- operaciones propias de targetClass.
```

```
-- Para toda operación propia de sourceClass, del paquete source,
sourceClass.oclAsType(Class).ownedOperation-> forAll (sOp /

    -- existirá en targetClass o en su descendencia una operación con las
    -- siguientes propiedades:
    targetClass.oclAsType(Class).allDescendant.ownedOperation ->exists (tOp /

        -- si el tipo de retorno y/o los parámetros de la operación sOp son
        -- del tipo Composite (del metamodelo origen) en tOp serán del tipo
        -- Composite (del metamodelo destino), es decir,
        -- para todo parámetro de la operación sOp de sourceClass
        sOp.ownedParameter -> forAll (sP /

            -- existirá un parámetro en la operación (tOp) de targetClass
            -- tal que,
            tOp.ownedParameter -> exists (tP /

                -- si el tipo del parámetro es Composite (del metamodelo origen), en
                -- targetClass serán del tipo Composite (del metamodelo destino)
                if (sP.type.oclIsTypeOf(Composite))
                then
                    tP.type = Composite
                else
                    tP.type = sP.type
                endif  and

                -- el resto de las propiedades de los parámetros se mantienen
                tP.direction = sP.direction  and
                tP.defaultValue = sP.defaultValue  and
                tP.isOrdered = sP.isOrdered  and
                tP.isUnique = sP.isUnique  and
                tP.upperValue = sP.upperValue  and
                tP.lowerValue = sP.lowerValue   ))  and

            -- el resto de las propiedades de la operación se mantienen
            tOp.name = sOp.name and
            tOp.class = sOp.class and
            tOp.isQuery = sOp.isQuery and
            tOp.precondition = sOp.precondition and
            tOp.postcondition = sOp.postcondition and
            tOp.bodyCondition = sOp.bodyCondition and
            tOp.raisedException = sOp. raisedException and
            tOp.redefinedOperation = sOp.redefinedOperation  )) and

-- Atributos propios de targetClass (atributos y extremos de asociación).
-- Para todo atributo propio de sourceClass, del paquete source,
sourceClass.oclAsType(Class).ownedAttribute -> forAll (sAt /

    -- existirá en targetClass o en su descendencia un atributo con las
    -- siguientes propiedades
    targetClass.oclAsType(Class).allDescendant.ownedAttribute  -> exists (tAt /

        -- si el atributo de sourceClass es de tipo Composite (del metamodelo origen)
```

```
            if (sAt.type.oclIsTypeOf(Composite))
            then

                --el tipo del atributo de targetClass es de tipo Composite (del
                -- metamodelo destino);
                tAt.type.oclIsTypeOf(Compostie)   and

                -- y el resto de las propiedades del atributo se mantienen
                tAt.visibility = sAt.visibility and
                tAt.isLeaf = sAt.isLeaf and
                tAt.isStatic = sAt.isStatic and
                tAt.isDerived = sAt.isDerived and
                tAt.isReadOnly = sAt.isReadOnly and
                tAt.isDerivedUnion = sAt.isDerivedUnion and
                tAt.aggregation = sAt.aggregation and
                tAt.upper = sAt.upper and
                tAt.lower = sAt.lower and
                tAt.association = sAt.association and
                tAt.owningAssociation = sAt.owningAssociation and
                tAt.redefinedProperty = sAt.redefinedProperty and
                tAt.subsettedProperty = sAt.subsettedProperty
            else
                -- en caso contrario, los atributos de targetClass serán iguales
                -- a los de sourceClass
                tAt = sAt
            endif  ))   and

        -- asociaciones heredadas de Classifier
        -- ambas clases tienen el mismo conjunto de generalizaciones
        targetClass.oclAsType(Class).generalization =
            sourceClass.oclAsType(Class).generalization  and

        -- ambas clases pertenecen al mismo package
        targetClass.oclAsType(Class).package =
            sourceClass.oclAsType(Class).package and

        -- ambas clases tienen el mismo conjunto de clasificadores redefinidos
        targetClass.oclAsType(Class).redefinedClassifier =
            sourceClass.oclAsType(Class).redefinedClassifier and

        -- ambas clases tienen el mismo conjunto de elementos importados
        targetClass.oclAsType(Class).elementImport =
             sourceClass.oclAsType(Class).elementImport and

        else
        -- en caso contrario, targetClass será igual a sourceClass.
        targetClass = sourceClass
        endif  ))
}
```

4.2.6 Ejemplo

Este refactoring se ejemplifica sobre un parser del HyperText Markup Language (HTML) (Kerievsky, 2004). Cuando el parser HTML analiza código HTML, identifica y crea objetos para representar tags y piezas de texto HTML. Por ejemplo, dado el siguiente fragmento:

```
<HTML>
    <BODY>
        Hello, and welcome to my Web page! I work for
        <A REF= “http://industriallogic.com”>
          < IMG SRC= “http://industriallogic.com/images/logo141x145.gif”>
       < /A>
   < /BODY >
< /HTML>
```

el parser creará los siguientes tipos de objetos:

- *HTMLTag* (para el tag <BODY>)
- *HTMLStringNode* (para el string “Hello, and welcome...”)
- *HTMLLinkTag* (para el tag <A HREF= “….”>)

El tag link (<A HREF= “….”>) contiene un tag imagen (<IMG SRC “…”>) que el parser trata como un hijo de *HTMLLinkTag*. Cuando el parser detecta que el tag link contiene un tag image, construye un objeto *HTMLImageTag* como hijo del objeto *HTMLLinkTag*. Otros tags en el parser (tales como: *HTMLFormTag*, *HTMLTitleTag*, entre otros) también son considerados hijos que almacenan colecciones de hijos.

La figura 4.4.a. muestra una jerarquía simplificada de los tags de HTML. Este modelo UML/OCL es una instancia del metamodelo origen del refactoring Extraer Composite. Los tags HTML pueden ser: form, link e image. Los tags form y link son contenedores de hijos, es decir, un tag link puede contener un tag image.

Las clases *HTMLLinkTag* y *HTMLFormTag* son instancias de la metaclase *Composite* del metamodelo origen del refactoring Extraer Composite, *HTMLTag* es una instancia de la metaclase *Component* y *HTMLImageTag* es una instancia de la metaclase *Leaf*.

La aplicación de la regla Extraer Composite conduce a:

- crear una nueva clase,
- nombrar a la nueva clase como *CompositeHTMLTag* (nombre dado por el diseñador),
- hacer que *CompositeHTMLTag* tenga como clase padre a *HTMLTag* y como hijas a *HTMLFormTag* y *HTMLLinkTag*,
- mover las operaciones equivalentes desde las clases *HTMLFormTag* y *HTMLLinkTag* hacia la nueva clase,
- remover las agregaciones que cada subclase tiene con la clase *HTMLTag* (las agregaciones *HTMLFormTag-HTMLTag* y *HTMLLinkTag-HTMLTag)* y reemplazarlas por una única agregación entre *CompositeHTMLTag* y *HTMLTag*.

Para identificar las operaciones duplicadas de las clases *HTMLLinkTag* y *HTMLFormTag* se realiza un matching entre pares de operaciones de las clases. Estos matchings de signaturas y de especificaciones, adaptados del trabajo de Zaremski, se basan en las precondiciones y postcondiciones OCL de las operaciones. El *matcheo* de signaturas de funciones está basado en un *matcheo* de tipos y se reduce al *matcheo* de los tipos del dominio y del tipo de la imagen de la función. Dos funciones pueden tener la misma signatura y los comportamientos pueden ser completamente opuesto. El *matcheo* de especificaciones tiene en cuenta el comportamiento de las funciones permitiendo aumentar la precisión con que se determina el *matcheo* de funciones. Dadas las especificaciones de signaturas de funciones S y S', los *matcheos* de especificaciones que se adecuan a este proceso de identificación son los siguientes:

Matcheo Exacto Pre/Post: establece si dos componentes son equivalentes y por lo tanto completamente intercambiables. Dos especificaciones satisfacen el *matcheo* exacto pre/post si sus precondiciones son equivalentes y sus postcondiciones son equivalentes.

$$match_{E\text{-}pre/post}(S,S') = (S'_{pre} <=> S_{pre}) \wedge (S_{post} <=> S'_{post})$$

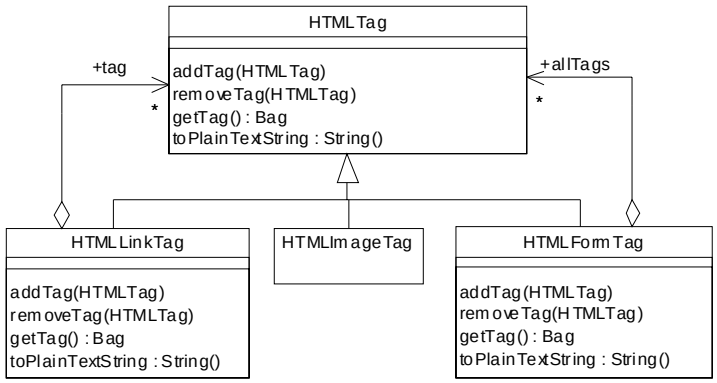
Matcheo Plug-In: bajo este *matcheo*, S' es *matcheada* por cualquier especificación S cuya precondición sea más débil (para permitir al menos todas las condiciones que S' permite) y cuya postcondición sea más fuerte (para proveer una garantía tan fuerte como S').

$$match_{plug\text{-}in}(S, S') = (S'_{pre} => S_{pre}) \wedge (S_{post} => S'_{post})$$

S es funcionalmente equivalente a S', ya que se puede reemplazar S por S' y tener el mismo comportamiento observable, pero la relación simétrica no se cumple.

Aplicando estos *matcheos* sobre el presente ejemplo se detecta a través del *matching* de signatura de la operación *addTag* de la clase *HTMLLinkTag matchea* con las operaciones *addTag* y *removeTag* de la clase *HTMLFormTag*. El matching de especificación determina que la operación *addTag* de la clase *HTMLLinkTag* sólo *matchea* con la operación addTag de la clase *HTMLFormTag*. El matching de especificación utilizado para este ejemplo en particular es el Plug-In Match, donde S es la especificación de la operación *addTag* de la clase *HTMLLinkTag* y S' la especificación de la operación *addTag* de la clase *HTMLFormTag* con el renombre de *tag* por *allTag*. El match $_{plug\text{-}in}$ (S, S') se reduce a probar (S_{post} => S'_{post}) ya que los requerimientos de la precondición (S'_{pre} => S_{pre}) se cumplen. Expresado en OCL sería:

```
( self.tag = self.tag@pre->including (t)  )
   implies
      ( self.tag→size() = self.tag@pre->size()+1   and
      self.tag->includes (t)  )
```



context HTMLLinkTag::addTag(t: HTMLTag)
post: self.tag = self.tag@pre ->including(t)

context HTMLLinkTag::removeTag (t: HTMLTag)
pre: self.tag-> includes(t)
post: self.tag = self.tag@pre → excluding(t)

context HTMLLinkTag::getTag (): Bag
post: result= self.tag

context HTMLLinkTag::toPlainTextTag():String
post: result = tags → iterate(
n:HTMLTag;
acc: String= '' | acc.concat(n.toPlainTextString())

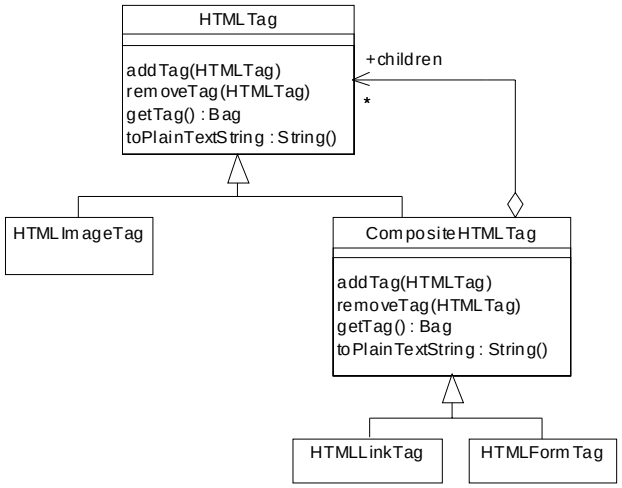
context HTMLFormTag::addTag(t: HTMLTag)
post: self.allTags→ size() =
self.allTags@pre -> size() + 1
post: self.allTags->includes (t)

context HTMLFormTag::removeTag (t: HTMLTag)
pre: self.allTags-> includes(t)
post: self.allTags->size() = self.allTags@pre -> size() - 1
post: not self.allTags->includes (t)

context HTMLFormTag::getTag (): Bag
post: result= self.allTags

context HTMLFormTag::toPlainTextTag():String
post: result=allTags → iterate(n:HTMLTag; acc:String= ''
| acc.concat(n.toPlainTextString())

a. Modelo Origen



context CompositeHTMLTag::addTag(t: HTMLTag)
post: self.tag = self.tag@pre ->including(t)

context CompositeHTMLTag::removeTag (t: HTMLTag)
pre: self.tag-> includes(t)
post: self.tag = self.tag@pre → excluding(t)

context CompositeHTMLTag::getTag (): Bag
post: result= self.tag

context CompositeHTMLTag::toPlainTextTag():String
post: result = tags → iterate(n:HTMLTag; acc: String= ''
| acc.concat(n.toPlainTextString())

b. Modelo Final

Figura 4.4 - Una instancia del refactoring *Extraer Composite*

Por lo tanto, S es equivalente funcionalmente a S', ya que se puede reemplazar S por S' y tener el mismo comportamiento observable, pero no se puede reemplazar S' por S y tener las mismas garantías. La operación *addTag* de la clase *HTMLLinkTag* es movida a la nueva clase Composite generada en el modelo destino. Similarmente se procede con las operaciones restantes. La tabla 4.1 muestra los matcheos resultantes y el modelo final se muestra en la figura 4.4.b.

Especificación de Operación S	Especificación de Operación S'	Matching de Especificación
HTMLLinkTag::removeTag	HTMLFormTag::removeTag	match $_{\text{plug-in}}$ (S, S')
HTMLLinkTag::getTag	HTMLFormTag::getTag	match $_{\text{E-pre/post}}$ (S, S')
HTMLLinkTag::toPlainTextTag	HTMLFormTag:: toPlainTextTag	match $_{\text{E-pre/post}}$ (S, S')

Tabla 4.1 - Matching de especificaciones

CAPÍTULO 5

REFACTORING A NIVEL PSM

En este capítulo se exhiben ejemplos de refactorings aplicables a modelos específicos de la plataforma C++ y Java. Se definen los metamodelos correspondientes a ambas plataformas teniendo en cuenta las características particulares de cada una.

El metamodelo C++, (ver Anexo B), refleja que una clase C++ tiene atributos, funciones y extremos de asociación, puede contener funciones y clases amigas como así también puede tener varias relaciones de generalización con otras clases. En cambio, el metamodelo Java (ver Anexo C), refleja la presencia de clases e interfaces Java en los modelos de esta plataforma. Una clase contiene campos, extremos de asociación y operaciones, puede heredar de una única clase base pero puede implementar varias interfaces. Una interfaz puede contener campos estáticos y finales, métodos sin implementación y extremos de asociación y además puede heredar de varias interfaces.

5.1 Refactoring de PSMs C++

En el nivel de modelos dependientes de la plataforma, en particular de la plataforma C++, se especializa el metamodelo UML para reflejar las características de la misma. Este metamodelo C++, a su vez, es especializado para cada transformación en particular, generando un metamodelo origen y un metamodelo destino por cada refactoring. Estos metamodelos son complementados con restricciones OCL.

5.1.1 Ejemplo: Herencia Múltiple, evitar réplicas de clases bases

Este refactoring está basado en las propuestas extraídas de:

- *Thinking in C++* de Bruce Eckel (Eckel, 2004), capítulo 10 de herencia múltiple y
- *Effective C++* de Scott Meyers (Meyers, 2005), donde propone mejoras a programas y diseños en C++ . El ejemplo se extrae del capítulo 6: Herencia y Diseño Orientado a objetos, ítem 40: *Use multiple inheritance judiciously*.

5.1.1.1 Descripción

Una clase tiene herencia múltiple cuando posee más de una clase base directa. Esto conduce a la posibilidad de tener una misma clase base repetida. En el ejemplo que se muestra a continuación, *Task* y *Displayed* son clases derivadas de la clase *Link* y la clase *Satellite* se deriva de *Task* y *Displayed*, por lo tanto *Satellite* tiene dos *Links*.

```
struct Link { ... };

class Task : public Link {
    / / the Link is used to maintain a list of all Tasks (the scheduler list)
    ... };

class Displayed : public Link {
    / / the Link is used to maintain a list of all Displayed objects (the display list)
    ... };

class Satellite : public Task, public Displayed {
    ... };
```

La clase *Satellite* usa dos objetos Links para representar los vínculos pero las dos listas no se interfieren. Naturalmente, debe referirse a los miembros de la clase Link salvando la ambigüedad que esto provoca.

```
void mess_ with_ links (Satellite* p)
{
    p->next = 0;                    / / referencia ambigua
    p->Link: :next = 0;             / / referencia ambigua
    p->Task: :Link: :next = 0;      / / ok
    p->Displayed: :Link: :next =0;  / / ok
    / / ...
}
```

En otros casos, sin embargo, no se quiere mantener la clase base común representada por varios objetos separados, por ejemplo:

```
class File {
    string fileName;
    ... };

class InputFile : public File { ... };

class OutputFile : public File { ... };

class IOFile : public InputFile, public OutputFile { ... };
```

La clase *IOFile* hereda una copia del dato miembro *fileName* (de la clase *File*) por cada una de sus clases bases, es decir, un objeto *IOFile* tendrá dos datos miembros *fileName*. Sin embargo, la lógica indica que un archivo de entrada/salida debería tener solamente un nombre de archivo. Por lo tanto, el campo *fileName* que hereda a través de sus clases bases no debería duplicarse. Para evitar la duplicación se debe tomar la clase *File* como clase base virtual. Es decir, todas las clases que heredan directamente de ella deberán usar herencia virtual tal como se muestra en el siguiente ejemplo.

```
class File {
    string fileName;
    ... };
```

```
class InputFile : virtual public File { ... };

class OutputFile : virtual public File { ... };

class IOFile : public InputFile, public OutputFile { ... };
```

La figura 5.1 ejemplifica este refactoring.

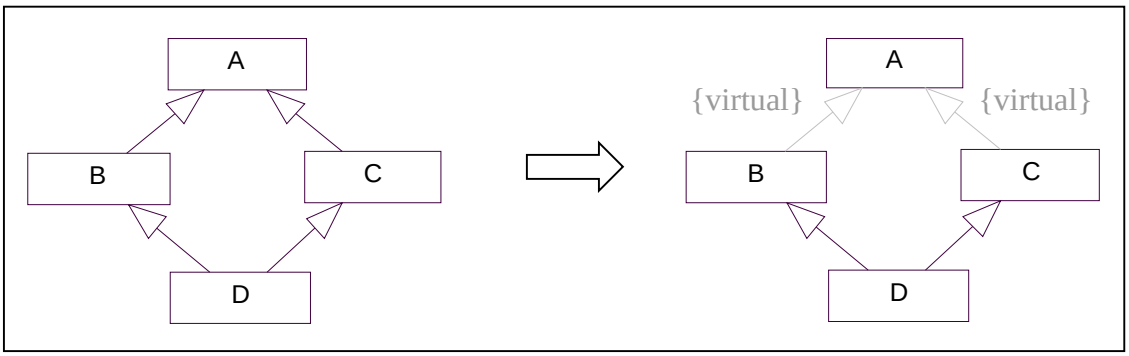


Figura 5.1 – Refactoring *Evitar réplicas de clases bases*

5.1.1.2 Motivación

El lenguaje C++, al soportar herencia múltiple, permite generar jerarquías de herencia donde se comparte una misma clase base. C++ realiza por defecto la replicación de la clase base. Si esto no es lo que se desea, el diseñador de la jerarquía deberá tomar la clase base como virtual. La decisión de si una clase base debe ser virtual o no, está basada en la estructura de la jerarquía de herencia completa. En el momento en que se están definiendo las clases B y C (figura 5.1), no se sabe si alguna otra clase va a heredar de ellas. Si no se declara A como clase base virtual de B y C, el diseñador de D necesitará modificar las definiciones de B y C para poder usarlas efectivamente. Es responsabilidad del diseñador aplicar este refactoring dependiendo de lo que se pretende de la jerarquía según lo detallado en los ejemplos de la introducción. Cabe aclarar que, al declarar A como clase base virtual de B y C se impone un costo adicional en tiempo y espacio a los clientes de dichas clases.

Cuando se definen funciones para una clase con una clase base virtual, por lo general, el programador no sabe si la base será compartida con otras clases derivadas. Esto puede ser un problema cuando se implementa un servicio que requiere que funciones de la clase base sean llamadas exactamente una vez. El lenguaje C++ asegura que el constructor de una base virtual sea llamado exactamente una vez. El constructor de una clase base virtual es invocado (implícita o explícitamente) desde el constructor de la clase más derivada. Esto significa, que la clase más derivada es responsable de inicializarla, pasando todos los parámetros requeridos al constructor de la clase base virtual. De esta manera siempre se está forzando a la clase más derivada a inicializar la clase base virtual lo cual es una tarea tediosa y confusa para el usuario de estas clases. Para desligar de esta responsabilidad a la clase más derivada, se crea un constructor por defecto en la clase base virtual.

5.1.1.3 Metamodelo origen

Sintaxis abstracta

El metamodelo origen (figura 5.2) es una especialización del metamodelo PSM-C++.

El metamodelo muestra las metaclases relacionadas a los participantes esenciales en los modelos a los cuales puede aplicarse este refactoring:

- *MostDerivedClass* representa la clase C++ más específica en una jerarquía de herencia múltiple,
- *MiddleClass* representa una clase C++ intermedia en una jerarquía de herencia múltiple,
- *SharedClass* representa una clase C++ compartida en una jerarquía de herencia múltiple, es decir, con más de un camino entre esta clase y la clase más derivada,
- *InheritanceRelation*, hereda de *C++Generalization*, es una relación de herencia que puede ser *FinalInheritance* o *MiddleInheritance*,
- *FinalInheritance* es una relación de herencia entre una clase y una clase base compartida en una jerarquía de herencia múltiple,
- *MiddleInheritance* es una relación de herencia entre clases intermedias en una jerarquía de herencia múltiple,
- metaclases sombreadas en gris claro corresponden al metamodelo PSM-C++.

El metamodelo origen sugiere que una instancia de *MostDerivedClass* tiene dos o más relaciones de herencia con clases que pueden ser instancias de *MiddleClass* o de *SharedClass*. Entre la instancia de *MostDerivedClass* y la instancia de *SharedClass* existen tantos caminos como relaciones de herencia tiene la instancia de *MostDerivedClass*.

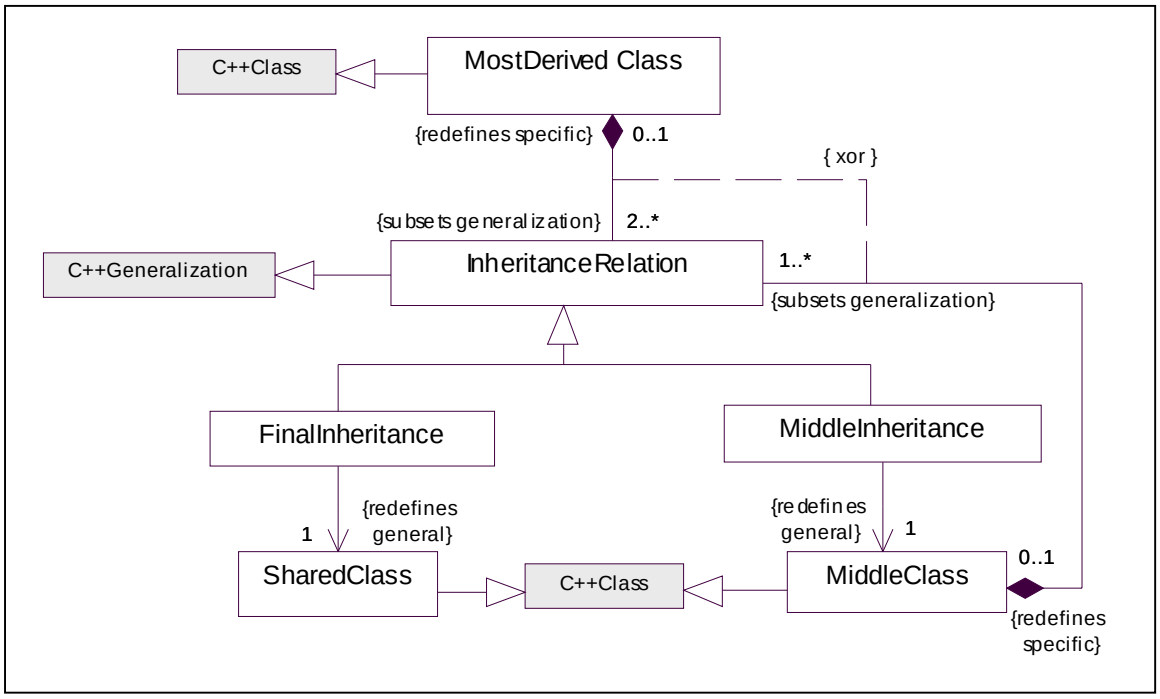


Figura 5.2 – Metamodelo origen del refactoring *Evitar réplicas de clases bases*

Descripciones de Metaclases

FinalInheritance

Representa una relación de herencia en la cual la clase más general es de tipo *SharedClass*.

Generalizaciones:

- InheritanceRelation.

Atributos

No posee atributos adicionales.

Asociaciones

- sharedClass: SharedClass [1]

  Referencia a la clase más general en la relación de herencia. Redefine *C++Generalization::general*.

Restricciones

[1] El especificador de acceso es público.

self.access-specifier = #public

InheritanceRelation

Representa una relación de herencia C++.

Generalizaciones:

- C++Generalization, del metamodelo PSM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- mostDerivedClass: MostDerivedClass [0..1]

  Referencia a la clase especializada en la relación de herencia. Redefine *C++Generalization::specific*.

- middleClass: MiddleClass [0..1]

  Referencia a la clase especializada en la relación de herencia. Redefine *C++Generalization::specific*.

Restricciones

No posee restricciones adicionales.

MiddleClass

Representa una clase C++ intermedia en una jerarquía de herencia.

Generalizaciones:

- C++Class, del metamodelo PSM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- inheritanceRelation: InheritanceRelation [1..*]

  Referencia al conjunto de relaciones de generalización que posee la clase. Subconjunto de *C++Class::generalization*.

Restricciones

No posee restricciones adicionales.

MiddleInheritance

Representa una relación de herencia en la cual la clase más general es de tipo *MiddleClass*.

Generalizaciones

- InheritanceRelation.

Atributos

No posee atributos adicionales.

Asociaciones

- middleClass: MiddleClass [1]

  Referencia a la clase más general en la relación de herencia. Redefine *C++Generalization::general*.

Restricciones

No posee restricciones adicionales.

MostDerivedClass

Representa a la clase C++ más derivada en una jerarquía de herencia.

Generalizaciones

- C++Class, del metamodelo PSM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- inheritanceRelation: InheritanceRelation [2..*]

  Referencia al conjunto de relaciones de generalización que posee la clase. Subconjunto de *C++Class::generalization*.

Restricciones

[1] La clase más derivada en todas sus relaciones de herencia de tipo *InheritanceRelation* tiene una clase en común en las cadenas de clases antecesoras.

```
self.allSuperClass -> select (oclIsTypeOf(SharedClass)) -> size() = 1
```

SharedClass

Representa una clase C++ compartida en una jerarquía de herencia.

Generalizaciones:

- C++Class, del metamodelo PSM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

Restricciones

No posee restricciones adicionales.

5.1.1.4 Metamodelo destino

Sintaxis abstracta

El metamodelo destino, especialización del metamodelo PSM-C++, muestra las metaclases relacionadas a los participantes esenciales en los modelos generados al aplicarse este refactoring (figura 5.3).

Las diferencias entre el metamodelo origen y destino sugieren que:

- en el modelo origen, una clase C++ (instancia de *MostDerivedClass* o de *MiddleClass*) tiene una relación de herencia con otra clase (instancia de *SharedClass*), en el metamodelo destino dicha relación de herencia es virtual,
- en el modelo destino, la instancia de *SharedClass* debe tener un constructor por defecto mientras que el metamodelo origen indica que puede estar presente o no.

Descripciones de metaclases

DefaultConstructor

Representa un constructor por defecto de la clase.

Generalizaciones:

- Constructor, del metamodelo PSM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- sharedClass: SharedClass [1]

  Referencia a la clase que posee el constructor. Redefine *C++MemberFunction::class*.

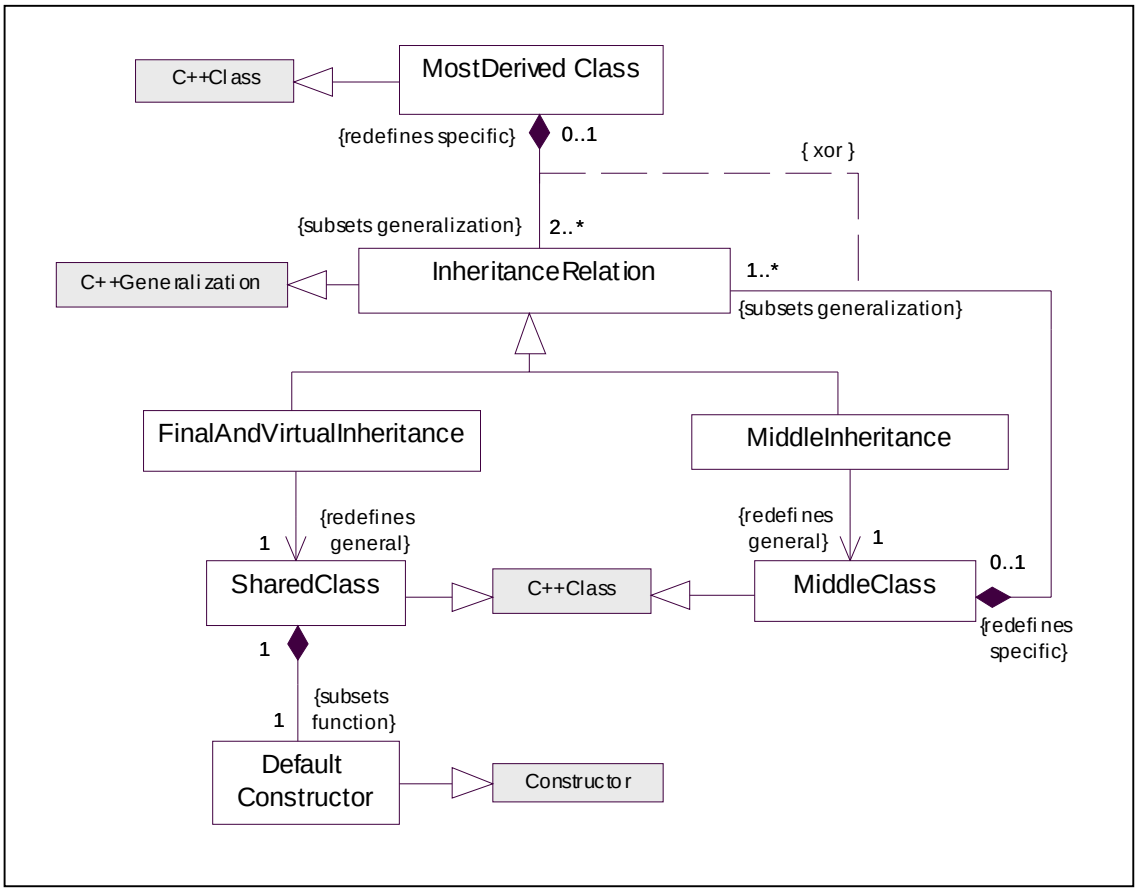


Figura 5.3 – Metamodelo destino del refactoring *Evitar réplicas de clases bases*

Restricciones

No posee restricciones adicionales.

FinalAndVirtualInheritance

Representa una relación de herencia virtual en la cual la clase más general es de tipo *SharedClass*.

Generalizaciones:

- InheritanceRelation.

Atributos

No posee atributos adicionales.

Asociaciones

- sharedClass: SharedClass [1]

  Referencia a la clase más general en la relación de herencia. Redefine *C++Generalization::general*.

Restricciones

[1] El especificador de acceso es público.

self.access-specifier = #public

[2] El tipo de herencia es virtual.

```
self.isVirtual = true
```

InheritanceRelation

Representa una relación de herencia C++.

Generalizaciones:

- C++Generalization, del metamodelo PSM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- mostDerivedClass: MostDerivedClass [0..1]

  Referencia a la clase especializada en la relación de herencia. Redefine *C++Generalization::specific*.

- middleDerivedClass: MiddleDerivedClass [0..1]

  Referencia a la clase especializada en la relación de herencia. Redefine *C++Generalization::specific*.

Restricciones

No posee restricciones adicionales.

MiddleClass

Representa una clase C++ intermedia en una jerarquía de herencia.

Generalizaciones:

- C++Class, del metamodelo PSM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- inheritanceRelation: InheritanceRelation [1..*]

  Referencia al conjunto de relaciones de generalización que posee la clase. Subconjunto de *C++Class::generalization*.

Restricciones

No posee restricciones adicionales.

MiddleInheritance

Representa una relación de herencia en la cual la clase más general es de tipo *MiddleClass*.

Generalizaciones

- InheritanceRelation.

Atributos

No posee atributos adicionales.

Asociaciones

- middleClass: MiddleClass [1]

  Referencia a la clase más general en la relación de herencia. Redefine *C++Generalization::general*.

Restricciones

No posee restricciones adicionales.

MostDerivedClass

Representa a la clase C++ más derivada en una jerarquía de herencia.

Generalizaciones

- C++Class, del metamodelo PSM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- inheritanceRelation: InheritanceRelation [2..*]

  Referencia al conjunto de relaciones de generalización que posee la clase. Subconjunto de *C++Class::generalization*.

Restricciones

[2] La clase más derivada en todas sus relaciones de herencia de tipo *InheritanceRelation* tiene una clase en común en las cadenas de clases antecesoras.

```
self.allSuperClass -> select (oclIsTypeOf(SharedClass)) -> size() = 1
```

SharedClass

Representa una clase C++ compartida en una jerarquía de herencia.

Generalizaciones:

- C++Class, del metamodelo PSM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- defaultConstructor: DefaultConstructor [1]

Referencia al constructor por defecto que posee la clase. Subconjunto de *C++Class::function.*

Restricciones

No posee restricciones adicionales.

5.1.1.5 Regla de transformación

Esta regla de transformación se aplica sobre jerarquías de herencia múltiple donde se comparte una misma clase base pero no se quiere tener réplicas de la misma. Este refactoring transformará las relaciones de herencia pública entre la clase base y sus clases derivadas en relaciones de herencia virtual. Además proveerá a la clase base compartida con un constructor por defecto, desligando a la clase más derivada de la responsabilidad de inicializarla.

A continuación, se especifican estos cambios a través de la regla de transformación.

Transformation Evitar réplicas de clases bases **{**
parameter
source: Metamodelo Origen Evitar réplicas de clases bases:: Package
target: Metamodelo Destino Evitar réplicas de clases bases:: Package

local operations

postconditions
post
-- Para toda clase *sourceClass*, de tipo *MostDerivedClass*, en el paquete *source*,
source.ownedMember -> select(oclIsTypeOf(MostDerivedClass)) ->
forAll (sourceClass |

-- existe una clase *targetClass*, de tipo *MostDerivedClass* en el paquete *target* tal que,
target.ownedMember -> select(oclIsTypeOf(MostDerivedClass)) ->
exists (targetClass |

-- atributos heredados de *NamedElement*
targetClass.name = sourceClass.name and
targetClass.visibility = sourceClass.visibility and

-- asociaciones heredadas de *Namespace*
-- ambas clases tienen las mismas restricciones
targetClass.oclAsType(C++Class).ownedRule =
sourceClass.oclAsType(C++Class).ownedRule and

-- ambas clases tienen el mismo conjunto de elementos importados
targetClass.oclAsType(C++Class).elementImport =
sourceClass.oclAsType(C++Class).elementImport

-- *targetClass* tiene los siguientes valores para:

```
-- attributo heredado de Classifier
targetClass.oclAsType(C++Class).isAbstract =
    sourceClass.oclAsType(C++Class).isAbstract and

-- asociaciones heredadas de Classifier

-- ambas clases pertenecen al mismo package
targetClass.oclAsType(C++Class).package =
    sourceClass.oclAsType(C++Class).package   and

-- ambas clases tienen el mismo conjunto de clasificadores redefinidos
targetClass.oclAsType(C++Class).redefinedClassifier =
    sourceClass.oclAsType(C++Class).redefinedClassifier  and

-- atributos heredados de C++Class
targetClass.oclAsType(C++Class).isFinal =
    sourceClass.oclAsType(C++Class).isFinal and

-- asociaciones heredadas de C++Class

-- targetClass  tiene el mismo conjunto de clases anidadas que
-- sourceClass
targetClass.oclAsType(C++Class).nestedClass =
    sourceClass.oclAsType(C++Class).nestedClass   and

-- targetClass tiene el mismo conjunto de funciones amigas que
-- sourceClass
targetClass.oclAsType(C++Class).friendClass =
    sourceClass.oclAsType(C++Class).friendClass and

-- targetClass  tiene el mismo conjunto de extremos de asociación que
-- sourceClass
targetClass.oclAsType(C++Class).associationEnd =
    sourceClass.oclAsType(C++Class).associationEnd and

-- targetClass  tiene el mismo conjunto de atributos que
-- sourceClass
targetClass.oclAsType(C++Class).atribute =
    sourceClass.oclAsType(C++Class).atribute and

-- targetClass  tiene el mismo conjunto de funciones que
-- sourceClass
targetClass.oclAsType(C++Class).function =
    sourceClass.oclAsType(C++Class).function and

-- targetClass  tiene el mismo conjunto de funciones amigas que
-- sourceClass
targetClass.oclAsType(C++Class).friendFunction =
    sourceClass.oclAsType(C++Class).friendFunction and
```

```
            -- targetClass  tiene el mismo conjunto de parámetros que
            -- sourceClass
            targetClass.oclAsType(C++Class).parameter =
                sourceClass.oclAsType(C++Class).parameter and

            -- por cada generalización que posee sourceClass del tipo
            -- FinalInheritance, en targetClass, existirá una generalización del tipo
            -- FinalAndVirtualInheritance,caso contrario, se mantienen como están,
            targetClass.oclAsType(C++Class).generalization -> forAll ( gt |
                sourceClass.oclAsType(C++Class).generalization -> exists ( gs|
                    if (gs.oclIsTypeOf(FinalInheritance) )
                    then gt.oclIsTypeOf(FinalAndVirtualInheritance)
                    else  gt = gs
                    endif  ) )

post
    -- Para toda clase sourceClass, del tipo SharedClass, en el paquete source,
    source.ownedMember -> select(oclIsTypeOf(SharedClass)) -> forAll ( sourceClass |

        -- existe una clase targetClass, de tipo SharedClass, en el paquete target tal que,
        target.ownedMember ->  select(oclIsTypeOf(SharedClass)) -> exists (targetClass |

            -- atributos heredados de NamedElement
            targetClass.name = sourceClass.name and
            targetClass.visibility = sourceClass.visibility and

            -- asociaciones heredadas de Namespace

            -- ambas clases tienen las mismas restricciones
            targetClass.oclAsType(C++Class).ownedRule =
                sourceClass.oclAsType(C++Class).ownedRule   and

            -- ambas clases tienen el mismo conjunto de elementos importados
            targetClass.oclAsType(C++Class).elementImport =
                sourceClass.oclAsType(C++Class).elementImport

            -- targetClass  tiene los siguientes valores para:

            -- attributo heredado de Classifier
            targetClass.oclAsType(C++Class).isAbstract =
                sourceClass.oclAsType(C++Class).isAbstract and

            -- asociaciones heredadas de Classifier
            -- ambas clases pertenecen al mismo package
            targetClass.oclAsType(C++Class).package =
                sourceClass.oclAsType(C++Class).package   and

            -- ambas clases tienen el mismo conjunto de clasificadores redefinidos
            targetClass.oclAsType(C++Class).redefinedClassifier =
                sourceClass.oclAsType(C++Class).redefinedClassifier  and
```

```
-- atributos heredados de C++Class
targetClass.oclAsType(C++Class).isFinal =
    sourceClass.oclAsType(C++Class).isFinal and

-- asociaciones heredadas de C++Class

-- targetClass  tiene el mismo conjunto de clases anidadas que
-- sourceClass
targetClass.oclAsType(C++Class).nestedClass =
    sourceClass.oclAsType(C++Class).nestedClass   and

-- targetClass tiene el mismo conjunto de funciones amigas que
-- sourceClass
targetClass.oclAsType(C++Class).friendClass =
    sourceClass.oclAsType(C++Class).friendClass and

-- targetClass  tiene el mismo conjunto de extremos de asociación que
-- sourceClass
targetClass.oclAsType(C++Class).associationEnd =
    sourceClass.oclAsType(C++Class).associationEnd and

-- targetClass  tiene el mismo conjunto de atributos que sourceClass
targetClass.oclAsType(C++Class).atribute =
    sourceClass.oclAsType(C++Class).atribute  and

-- targetClass  incluye el mismo conjunto de funciones que sourceClass
targetClass.oclAsType(C++Class).function -> includesAll
    sourceClass.oclAsType(C++Class).function and
-- targetClass posee un constructor por defecto
targetClass.oclAsType(C++Class).function -> exists (f |
    f.oclIsTypeOf(DefaultConstructor))  and

-- targetClass  tiene el mismo conjunto de funciones amigas que
-- sourceClass
targetClass.oclAsType(C++Class).friendFunction =
    sourceClass.oclAsType(C++Class).friendFunction and

-- targetClass  tiene el mismo conjunto de parámetros que sourceClass
targetClass.oclAsType(C++Class).parameter =
    sourceClass.oclAsType(C++Class).parameter and

-- targetClass  tiene el mismo conjunto de generalizaciones que
-- sourceClass
targetClass.oclAsType(C++Class).generalization =
    sourceClass.oclAsType(C++Class).generalization and
```

post

```
    -- Para toda clase sourceClass, del tipo MiddleClass, en el paquete source,
    source.ownedMember -> select(oclIsTypeOf(MiddleClass)) -> forAll ( sourceClass |
```

```
        -- existe una clase targetClass, de tipo MiddleClass, en el paquete target tal que,
        target.ownedMember -> select(oclIsTypeOf(MiddleClass)) ->exists ( targetClass |

            -- ambas clases tienen las mismas características.
            targetClass  =  sourceClass))


post
    -- Para toda clase sourceClass, de tipo C++Class, en el paquete source,
        source.ownedMember -> select(oclIsTypeOf(C++Class)) ->
            forAll ( sourceClass |

        -- existe una clase targetClass, de tipo C++Class, en el paquete target tal que,
        target.ownedMember ->  select(oclIsTypeOf(C++Class)) ->
            exists ( targetClass |

            -- ambas clases tienen las mismas características.
            targetClass  =  sourceClass))
}
```

5.1.1.6 Ejemplo

La figura 5.4.a muestra una jerarquía de herencia múltiple. Un objeto *Window* (instancia de la clase *Window*) registra las coordenadas x e y del vértice superior izquierdo, el ancho y la altura de la ventana. Se puede conocer de este objeto sus atributos, cambiar la ubicación del vértice, cambiar el color del fondo de la ventana y dibujarla.

Una ventana con borde (instancia de *WindowWithBorder*) hereda todos los atributos y comportamiento de ventana y además posee un tamaño y estilo del borde. Este último se puede definir a través de una operación.

Una ventana con menú (instancia de *WindowWithMenu*) hereda todos los atributos y comportamiento de ventana y además puede insertar opciones de menúes.

Una instancia de *MyWindow* hereda de *WindowWithBorder* y de *WindowWithMenu*.

Este modelo es una instancia del metamodelo origen del refactoring *Evitar réplicas de clases bases* (figura 5.2) donde *Window* es una instancia de la metaclase *SharedClass*, *WindowWithBorder* y *WindowWithMenu* son instancias de *MiddleClass* y *MyWindow* es una instancia de *MostDerivedClass*. Tanto entre *MyWindow* y *WindowWithBorder* como entre *MyWindow* y *WindowWithMenu* existe una relación de herencia (instancia de *MiddleInheritance*). Entre *WindowWithBorder* y *Window* como entre *WindowWithMenu* y *Window* existe una relación de herencia final (instancia de *FinalInheritance*).

La aplicación de esta regla conduce a:

- cambiar la relación de herencia existente entre *WindowWithBorder* y *Window* reflejando que esta última clase es una clase base virtual, es decir, heredar de *Window* virtualmente;

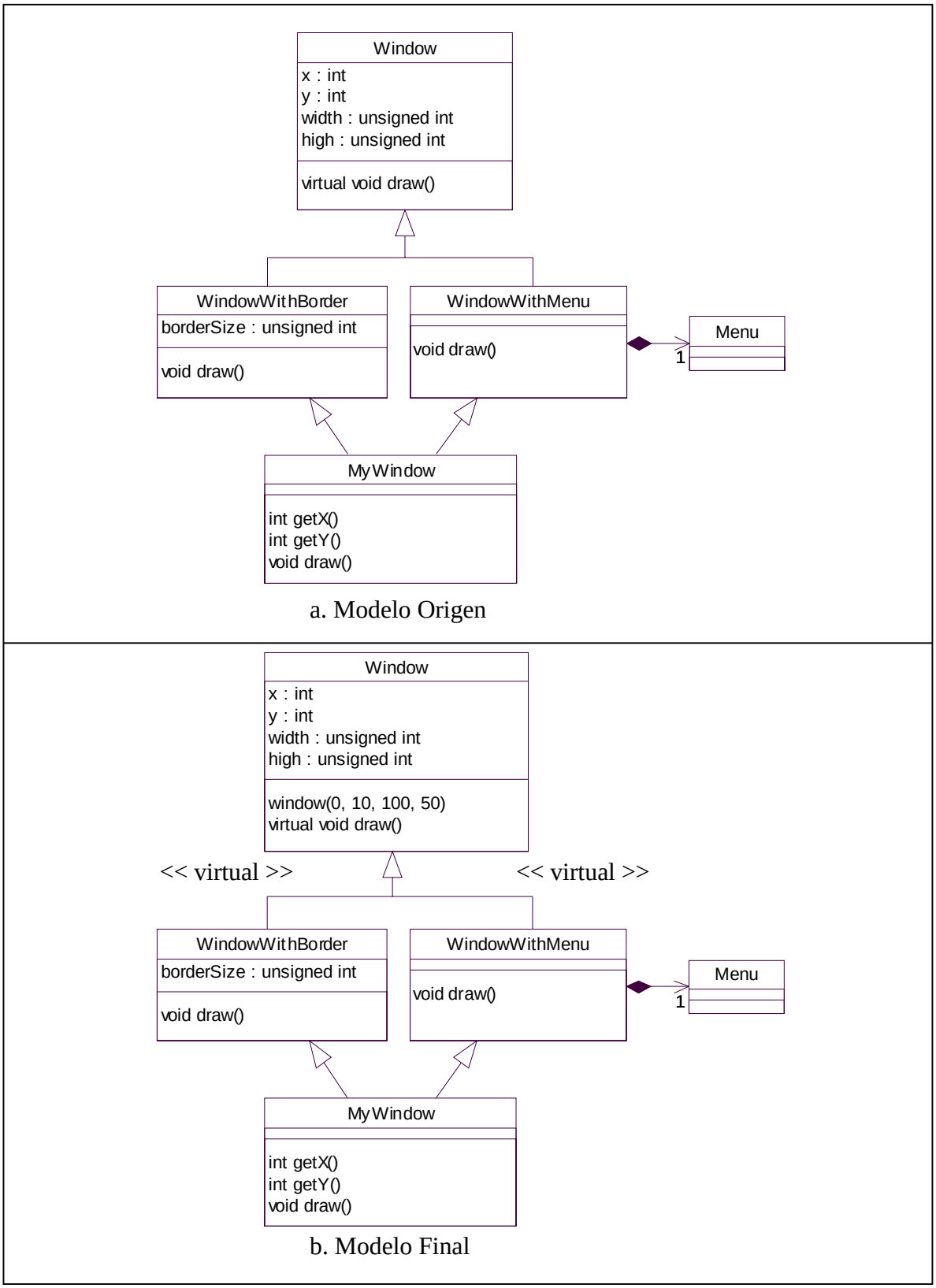


Figura 5.4 – Una instancia del refactoring *Evitar réplicas de clases bases*

- cambiar la relación de herencia existente entre *WindowWithMenu* y *Window* reflejando que esta última clase es una clase base virtual, es decir, heredar de *Window* virtualmente;
- agregar a la clase *Window* un constructor por defecto, en este caso, un constructor con parámetros que tengan valores por defecto;
- El modelo resultante, figura 5.4.b, es una instancia del metamodelo destino del refactoring *Evitar réplicas de clases bases* (figura 5.3).

5.2 Refactoring de PSMs Java

En el nivel de modelos dependientes de la plataforma Java, se especializa el metamodelo UML para reflejar las características de esta plataforma. Este metamodelo Java, a su vez, es especializado para cada transformación en particular, generando un metamodelo origen y un metamodelo destino por cada refactoring, complementados con restricciones OCL.

5.2.1 Ejemplo: Extraer Interfaz

Este refactoring está basado en el refactoring propuesto en el libro *Refactoring: Improving the Design of Existing Code*, capítulo 11, página 277 (Fowler, 1999).

5.2.1.1 Descripción

Este refactoring se aplica cuando dos clases tienen un subconjunto de operaciones comunes. Si las operaciones tienen la misma signatura el refactoring creará una interfaz que contendrá las operaciones comunes.

La figura 5.5 ejemplifica este refactoring.

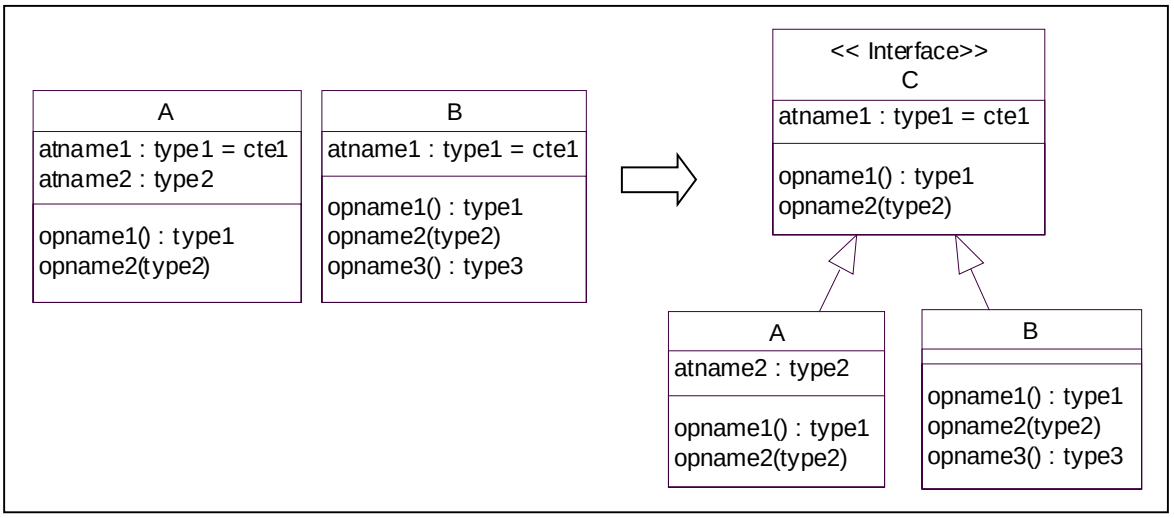


Figura 5.5 – Refactoring *Extraer Interfaz*

5.2.1.2 Motivación

Cuando en dos clases 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.

5.2.1.3 Metamodelo origen

Sintaxis abstracta

El metamodelo origen (figura 5.6) es una especialización del metamodelo PSM-Java.

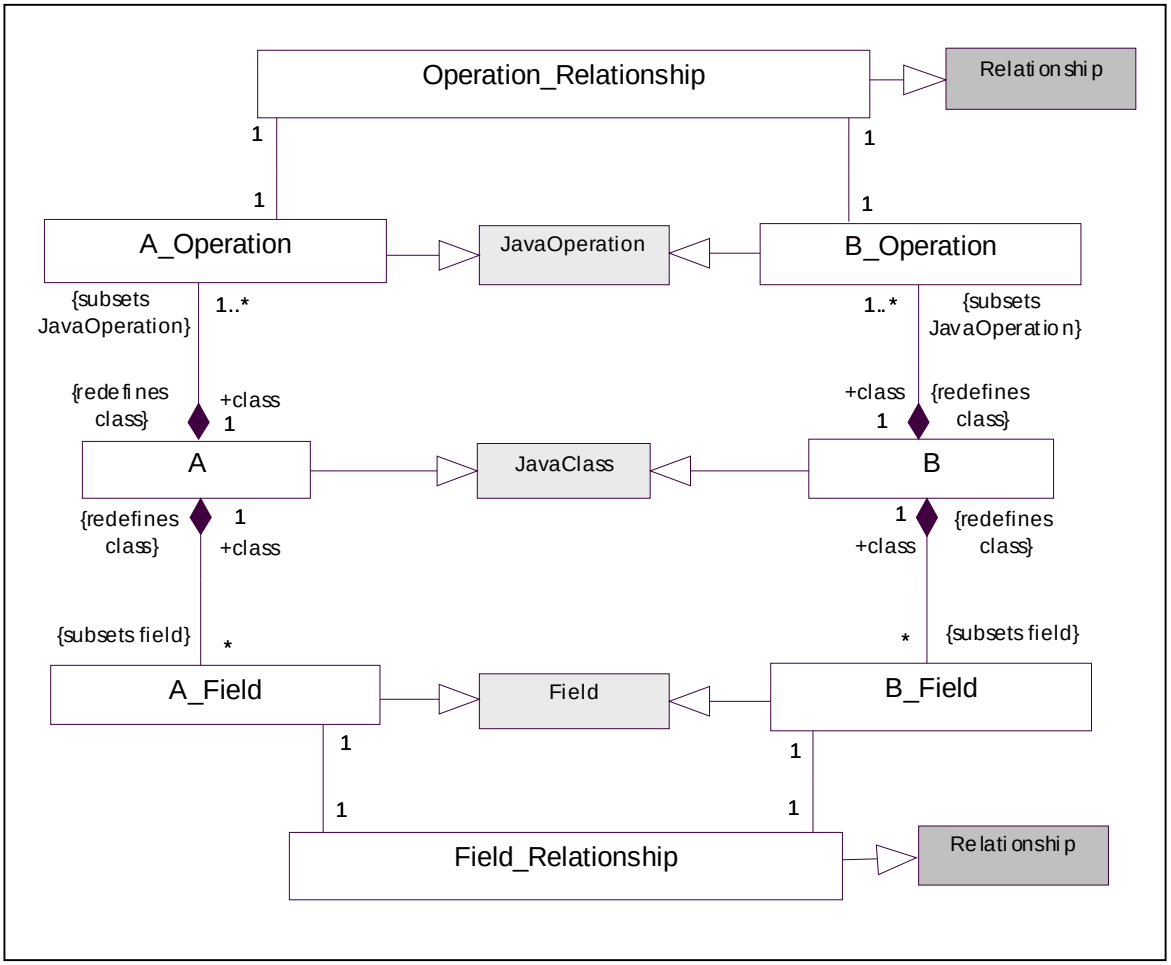


Figura 5.6 – Metamodelo origen del refactoring *Extraer Interfaz*

El metamodelo muestra las metaclases relacionadas a los participantes esenciales en los modelos a los cuales puede aplicarse este refactoring:

- *A* y *B* representan clases Java que poseen operaciones comunes,
- *A_Operation* representa una operación Java de la clase *A* que es sintácticamente equivalente a una operación de la clase B,
- *B_Operation* representa una operación Java de la clase *B* que es sintácticamente equivalente a una operación de la clase A,
- *Operation_Relationship*, hereda de *Relationship*, es una relación que vincula dos operaciones de A y B respectivamente que son sintácticamente equivalentes,
- *A_Field* representa un campo Java de la clase *A* que es equivalente a un campo de la clase B,
- *B_Field* representa un campo Java de la clase *B* que es equivalente a un campo de la clase A,
- *Field_Relationship*, hereda de *Relationship*, es una relación que vincula dos campos de A y B respectivamente que son equivalentes,
- metaclases sombreadas en gris claro corresponden al metamodelo PSM-Java,
- metaclases sombreadas en gris oscuro corresponden al metamodelo UML.

El metamodelo origen sugiere que una instancia de *A* y una instancia 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 campos, instancias de *A_Field* y *B_Field* respectivamente, entre las cuales existe una relación de equivalencia (instancia de *Field_Relationship*).

Descripciones de metaclases

A

Representa una clase Java.

Generalizaciones

- JavaClass, del metamodelo PSM-Java.

Atributos

No posee atributos adicionales.

Asociaciones

- a_Operation: A_Operation [1..*]

  Referencia un subconjunto de operaciones de la clase. Subconjunto de *JavaClass::JavaOperation*.

- a_Field: A_Field [*]

  Referencia un subconjunto de campos de la clase. Subconjunto de *JavaClass::Field*.

Restricciones

[1] La clase posee un subconjunto de operaciones, *A_Operation*, que son operaciones sintácticamente equivalentes a un subconjunto de operaciones de una clase *B*.

```
self.a_Operation -> forAll (Aop |
    self.package.ownedMember -> exists (m | m.oclIsTypeOf(B) and
        m.oclAsType(B).b_Operation -> exists (Bop |
                Bop.name = Aop.name and
                Bop.ownedParameter = Aop.ownedParameter  and
                Bop.type = Aop.type  ) ) )
```

[2] La clase posee un subconjunto de campos, *A_Field*, que son equivalentes a un subconjunto de campos de una clase *B*.

```
self.a_Field -> forAll (Afield |
    self.package.ownedMember -> exists (m | m.oclIsTypeOf(B) and
        m.oclAsType(B).b_Field -> exists (Bfield |
                Bfield.name = Afield.name and
                Bfield.visibility = Afield.visibility = public  and
                Bfield.isStatic = Afield.isStatic = true  and
                Bfield.isFinal = Afield.isFinal = true  and
                Bfield.isVolatile = Afield. isVolatile and
                Bfield.isTransient = Afield.isTransient   ) ) )
```

A_Field

Representa un campo Java.

Generalizaciones

- Field, del metamodelo PSM-Java.

Atributos

No posee atributos adicionales.

Asociaciones

- field_Relationship: Field_Relationship [1]

  Referencia a la relación de equivalencia que posee con otro campo de la clase B.

- class: A [1]

  Referencia la clase que posee el campo. Redefine *Field::class*.

Restricciones

No posee restricciones adicionales.

A_Operation

Representa una operación Java.

Generalizaciones

- JavaOperation, del metamodelo PSM-Java.

Atributos

No posee atributos adicionales.

Asociaciones

- operation_Relationship: Operation_Relationship [1]

  Referencia a la relación de equivalencia que posee con otra operación de la clase B.

- class: A [1]

  Referencia la clase que posee la operación. Redefine *JavaOperation::class*.

Restricciones

No posee restricciones adicionales.

B

Representa una clase Java.

Generalizaciones

- JavaClass, del metamodelo PSM-Java.

Atributos

No posee atributos adicionales.

Asociaciones

- b_Operation: B_Operation [1..*]

  Referencia un subconjunto de operaciones de la clase. Subconjunto de *JavaClass::JavaOperation*.

- b_Field: B_Field [*]

  Referencia un subconjunto de campos de la clase. Subconjunto de *JavaClass::Field*.

Restricciones

[1] La clase posee un subconjunto de operaciones, *B_Operation*, que son operaciones sintácticamente equivalentes a un subconjunto de operaciones de una clase A.

```
self.b_Operation -> forAll (Bop |
   self.package.ownedMember -> exists (m | m.oclIsTypeOf(A) and
      m.oclAsType(A).a_Operation -> exists (Aop |
                  Aop.name = Bop.name and
                  Aop.ownedParameter = Bop.ownedParameter  and
                  Aop.type = Bop.type  ) ) )
```

[2] La clase posee un subconjunto de campos, B_Field, que son equivalentes a un subconjunto de campos de una clase A.

```
self.b_Field -> forAll (Bfield |
    self.package.ownedMember -> exists (m | m.oclIsTypeOf(A) and
        m.oclAsType(A).a_Field -> exists (Afield |
                Afield.name = Bfield.name and
                Afield.visibility = Bfield.visibility = public  and
                Afield.isStatic = Bfield.isStatic = true  and
                Afield.isFinal = Bfield.isFinal = true  and
                Afield.isVolatile = Bfield. isVolatile and
                Afield.isTransient = Bfield.isTransient   ) ) )
```

B_Field

Representa un campo Java.

Generalizaciones

- Field, del metamodelo PSM-Java.

Atributos

No posee atributos adicionales.

Asociaciones

- field_Relationship: Field_Relationship [1]

  Referencia a la relación de equivalencia que posee con otro campo de la clase A.

- class: B [1]

  Referencia la clase que posee la operación. Redefine *Field::class*.

Restricciones

No posee restricciones adicionales.

B_Operation

Representa una operación Java.

Generalizaciones

- JavaOperation, del metamodelo PSM-Java.

Atributos

No posee atributos adicionales.

Asociaciones

- operation_Relationship: Operation_Relationship [1]

  Referencia a la relación de equivalencia que posee con otra operación de la clase A.

- class: B [1]

Referencia la clase que posee la operación. Redefine *JavaOperation::class*.

Restricciones

No posee restricciones adicionales.

Field_Relationship

Representa una relación de equivalencia entre dos campos.

Generalizaciones

- Relationship, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- a_Field: A_Field [1]

  Referencia un campo de la clase A.

- b_Field: B_Field [1]

  Referencia un campo de la clase B.

Restricciones

No posee restricciones adicionales.

Operation_Relationship

Representa una relación de equivalencia sintáctica entre dos operaciones.

Generalizaciones:

- Relationship, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- a_Operation: A_Operation [1]

  Referencia una operación de la clase A.

- b_Operation: B_Operation [1]

  Referencia una operación de la clase B.

Restricciones

No posee restricciones adicionales.

5.2.1.4 Metamodelo destino

Sintaxis abstracta

El metamodelo destino, especialización del metamodelo PSM-Java, muestra las metaclases relacionadas a los participantes esenciales en los modelos generados al aplicarse este refactoring (figura 5.7).

Las diferencias entre el metamodelo origen y destino son las siguientes:

- en el metamodelo origen, dos clases Java *A* y *B*, poseen operaciones comunes y campos equivalentes,
- en el metamodelo destino, las operaciones comunes y los campos equivalentes pasan a ser miembros de una interfaz (instancia de *AnInterface*) que las clases *A* y *B* implementan.

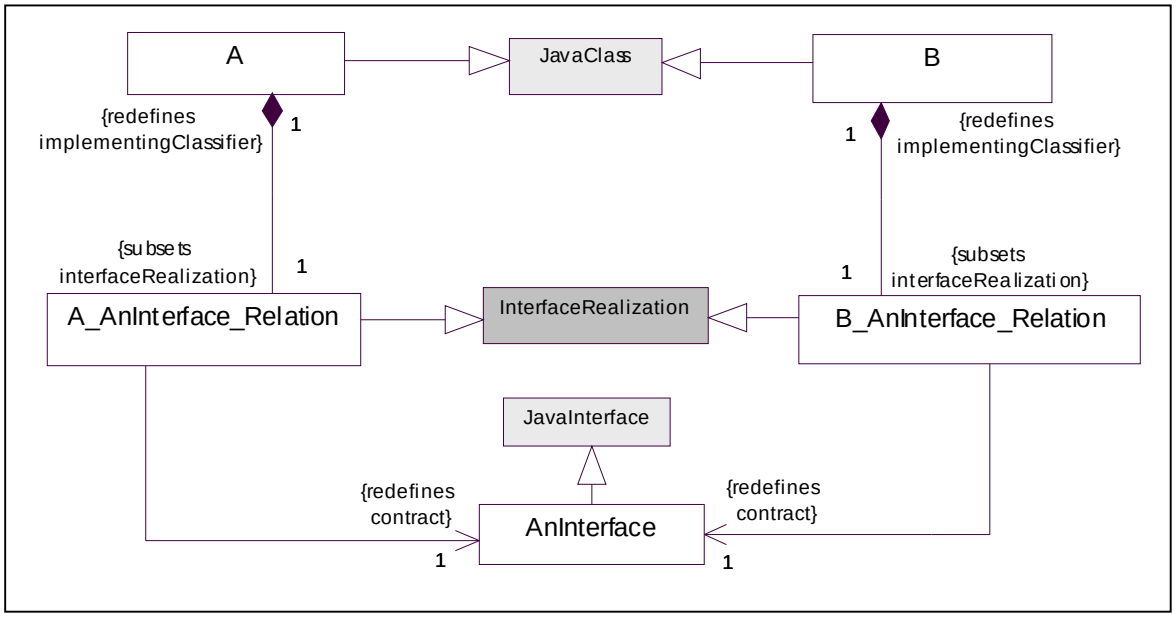


Figura 5.7 – Metamodelo destino del refactoring *Extraer Interfaz*

Descripciones de Metaclases

A

Representa una clase Java.

Generalizaciones

- JavaClass, del metamodelo PSM-Java.

Atributos

No posee atributos adicionales.

Asociaciones

- a_AnInterface_Relation: A_AnInterface_Relation [1]

Referencia a la relación de realización de interfaz de la clase. Subconjunto de *BehavioredClassifier::interfaceRealization*.

Restricciones

No posee restricciones adicionales.

AnInterface

Representa una interfaz Java.

Generalizaciones:

- JavaInterface, del metamodelo PSM-Java.

Atributos

No posee atributos adicionales.

Asociaciones

No posee asociaciones adicionales.

Restricciones

No posee restricciones adicionales.

A_AnInterface_Relation

Representa una relación de realización de interfaz entre la clase *A* y la interfaz *AnInterface*.

Generalizaciones

- InterfaceRealization, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- anInterface: AnInterface [1]

  Referencia a la interfaz que participa en la relación de realización. Redefine *InterfaceRealization::contract*.

- a: A [1]

  Referencia a la clase que participa en la relación. Redefine *InterfaceRealization::implementingClassifier.*.

Restricciones

No posee restricciones adicionales.

B

Representa una clase Java.

Generalizaciones

- JavaClass, del metamodelo PSM-Java.

Atributos

No posee atributos adicionales.

Asociaciones

- b_AnInterface_Relation: B_AnInterface_Relation [1]

  Referencia a la relación de realización de interfaz de la clase. Subconjunto de *BehavioredClassifier::interfaceRealization*.

Restricciones

No posee restricciones adicionales.

B_AnInterface_Relation

Representa una relación de realización de interfaz entre la clase *B* y la interfaz *AnInterface*.

Generalizaciones

- InterfaceRealization, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- anInterface: AnInterface [1]

  Referencia a la interfaz que participa en la relación de realización. Redefine *InterfaceRealization::contract*.

- b: B [1]

  Referencia a la clase que participa en la relación. Redefine *InterfaceRealization::implementingClassifier*..

Restricciones

No posee restricciones adicionales.

5.2.1.5 Regla de transformación

La regla *Extraer Interfaz* se aplica cuando dos clases tienen un subconjunto de operaciones comunes. Este refactoring genera una interfaz Java que contendrá las operaciones de ambas clases que comparten la interfaz y eventualmente campos equivalentes.

A continuación se detallan los pasos para aplicar esta regla.

1. Generar una interfaz Java.
2. Declarar en la interfaz las operaciones comunes.
3. Declarar en la interfaz los campos equivalentes.
4. Declarar que las clases originales implementan dicha interfaz.
5. Adaptar las declaraciones de tipo de los clientes para usar la interfaz.

Se especifican estos cambios a través de la siguiente regla de transformación.

Transformation Extraer Interfaz **{**

parameter
source: Metamodelo Origen Extraer Interfaz:: JavaPackage
target: Metamodelo Destino Extraer Interfaz:: JavaPackage

local operations

Metamodelo Origen Extraer Interfaz::JavaOperation::onlyUseASubsetOf (
Metamodelo Origen Extraer Interfaz:: class: JavaClass): Boolean;
-- Verifica si en el cuerpo de la operación se invoca sólo el subconjunto de
-- operaciones comunes de las operaciones de la clase pasada como parámetro.
onlyUseASubsetOf *=*
self.body.invokedMethod -> exists (m | class.a_Operation -> includes (m)) and
self.body.invokedMethod -> excludesAll (
class.javaOperation – class.a_Operation)

postconditions

post
-- Para todo par de clases, *sourceA* y *sourceB* del tipo *A y B* respectivamente, en el
-- paquete *source*,
source.ownedMember -> select(oclIsKindOf(JavaClass)) -> forAll (
sourceA, sourceB |
sourceA.oclIsTypeOf(A) and
sourceB.oclIsTypeOf(B) and

-- existe una clase, *targetA* del tipo A, en el paquete *target* tal que,
target.ownedMember -> select(oclIsTypeOf(JavaClass)) -> exists (targetA |
targetA.oclIsTypeOf(A) and

-- y existe una clase, *targetB* del tipo B, en el paquete *target* tal que,
target.ownedMember -> select(oclIsKindOf(JavaClass)) -> exists (targetB |
targetB.oclIsTypeOf(B) and

-- atributos heredados de *JavaClass*

-- las clases del paquete *target* tienen los mismos valores que las clases
-- del paquete *source* para los siguientes atributos:
targetA.oclAsType(A).isFinal = sourceA.oclAsType(A).isFinal and
targetB.oclAsType(B).isFinal = sourceB.oclAsType(B).isFinal and

targetA.oclAsType(A).isStatic = sourceA.oclAsType(A).isStatic and
targetB.oclAsType(B).isStatic = sourceB.oclAsType(B).isStatic and

-- asociaciones heredadas de *JavaClass*

-- las clases del paquete *target* tienen los mismos valores que las clases
-- del paquete *source* para las siguientes asociaciones:
targetA.oclAsType(A).javaPackage =
sourceA.oclAsType(A).javaPackage and

```
targetB.oclAsType(B).javaPackage =
    sourceB.oclAsType(B).javaPackage  and

targetA.oclAsType(A).parameters =
    sourceA.oclAsType(A).parameters and
targetB.oclAsType(B).parameters =
    sourceB.oclAsType(B).parameters and

targetA.oclAsType(A).associationEnd =
    sourceA.oclAsType(A).associationEnd   and
targetB.oclAsType(B).associationEnd =
    sourceB.oclAsType(B).associationEnd    and

targetA.oclAsType(A).superClass =
    sourceA.oclAsType(A).superClass    and
targetB.oclAsType(B).superClass =
    sourceB.oclAsType(B).superClass and

targetA.oclAsType(A).nestedClass =
    sourceA.oclAsType(A).nestedClass   and
targetB.oclAsType(B).nestedClass =
    sourceB.oclAsType(B).nestedClass    and

targetA.oclAsType(A).javaOperation =
    sourceA.oclAsType(A).javaOperation   and
targetB.oclAsType(B).javaOperation =
    sourceB.oclAsType(B).javaOperation    and

-- las clases del paquete target tienen los campos de las clases del
-- paquete source menos aquellos campos equivalentes que son movidos
-- a la interfaz
targetA.oclAsType(A).field =
    sourceA.oclAsType(A).field – sourceA.oclAsType(A).a_field      and
targetB.oclAsType(B).field =
    sourceB.oclAsType(B).field – sourceB.oclAsType(B).b_field       and

-- en el paquete target las clases targetA y targetB incluyen el conjunto
-- de interfaces que implementan las clases del paquete source,
targetA.oclAsType(A).implement -> includes
    (sourceA.oclAsType(A).implement)  and

targetB.oclAsType(B).implement -> includes
    (sourceB.oclAsType(B).implement)  and

-- y además, las clases del paquete target implementan una instancia de
-- AnInterface
targetA.oclAsType(A).implement -> includes (
    targetA.oclAsType(A).a_AnInterface_Relation.anInterface)  and
```

```
targetB.oclAsType(B).implement -> includes (
    targetB.oclAsType(B).b_AnInterface_Relation.anInterface)  and

-- la instancia de AnInterface es la misma para ambas clases del paquete
-- target
targetB.oclAsType(B).b_AnInterface_Relation.anInterface  =
    targetA.oclAsType(A).a_AnInterface_Relation.anInterface     and

-- se define theInterface como la interfaz común a ambas clases del
-- paquete target
let theInterface: JavaInterface =
    targetB.oclAsType(B).b_AnInterface_Relation.anInterface  in

    -- dicha interfaz contendrá las operaciones comunes de las clases del
    -- paquete source
    theInterface.method = sourceA.oclAsType(A).a_Operation    and
    theInterface.method = sourceB.oclAsType(B).b_Operation    and

    -- dicha interfaz contendrá los campos equivalentes de las clases del
    -- paquete source
    theInterface.field = sourceA.oclAsType(A).a_field    and
    theInterface.field = sourceB.oclAsType(B).b_field    and

-- atributos heredados de NamedElement
targetA.name = sourceA.name and
targetB.name = sourceB.name and

targetA.visibility = sourceA.visibility and
targetB.visibility = sourceB.visibility and

-- attributo heredado de Classifier
targetA.oclAsType(A).isAbstract =
    sourceA.oclAsType(A).isAbstract and
targetB.oclAsType(B).isAbstract =
    sourceB.oclAsType(B).isAbstract and

-- asociaciones heredadas de Namespace
-- ambas clases tienen las mismas restricciones
targetA.oclAsType(A).ownedRule =
    sourceA.oclAsType(A).ownedRule   and
targetB.oclAsType(B).ownedRule =
    sourceB.oclAsType(B).ownedRule   and

-- las clases tienen el mismo conjunto de elementos importados
targetA.oclAsType(A).elementImport =
    sourceA.oclAsType(A).elementImport  and
targetB.oclAsType(B).elementImport =
    sourceB.oclAsType(B).elementImport and
```

```
-- si alguna clase del paquete source es cliente sólo del subconjunto de
-- operaciones que son movidas a la interfaz, se ajustan las declaraciones
-- de tipo en dicha clase para denotar que sólo es usado ese
-- comportamiento, en caso contrario mantiene las propiedades en el
-- paquete target

-- para toda clase classS en el paquete source,
source.ownedMember ->  select(oclIsTypeOf(JavaClass)) -> forAll (classS |

-- existe una clase classT en el paquete target tal que,
target.ownedMember ->  select(oclIsTypeOf(JavaClass)) -> exists ( classT |

    -- la clase del paquete target  tiene los mismos valores que la clase
    -- del paquete source para los siguientes atributos:
    -- atributos heredados de JavaClass
    classT.oclAsType(JavaClass).isFinal =
        classS.oclAsType(JavaClass).isFinal    and

    classT.oclAsType(JavaClass).isStatic =
        classS.oclAsType(JavaClass).isStatic and

    -- la clase del paquete target  tiene los mismos valores que la clase
    -- del paquete source para las siguientes asociaciones:
    -- asociaciones heredadas de JavaClass
    classT.oclAsType(JavaClass).javaPackage =
        classS.oclAsType(JavaClass).javaPackage  and

    classT.oclAsType(JavaClass).parameters =
        classS.oclAsType(JavaClass).parameters and

    classT.oclAsType(JavaClass).associationEnd =
        classS.oclAsType(JavaClass).associationEnd   and

    classT.oclAsType(JavaClass).superClass =
        classS.oclAsType(A).superClass and

    classT.oclAsType(JavaClass).nestedClass =
        classS.oclAsType(JavaClass).nestedClass   and

    classT.oclAsType(JavaClass).field =
        classS.oclAsType(JavaClass).field   and

    classT.oclAsType(JavaClass).implement =
        classS.oclAsType(JavaClass).implement   and

    -- en las operaciones de la clase del paquete target se deberán ajustar
    -- las declaraciones de tipo si la operación de la clase del paquete
    -- source sólo usa el subconjunto de operaciones que son movidas a la
```

```
-- interfaz
classS.oclAsType(JavaClass).javaOperation -> forAll (sOp |
    classT.oclAsType(JavaClass).javaOperation  -> exists (tOp |

    if (sOp.onlyUseASubsetOf (sourceA))
    then
    -- si el tipo de retorno y/o los parámetros de la operación sOp
    -- son del tipo A o B, en tOp serán del tipo AnInterface, es decir,

    -- para todo parámetro de la operación sOp de ClassS
    sOp.ownedParameter ->  forAll (sP /

        -- existirá un parámetro en la operación (tOp)de ClassT tal que,
        tOp.ownedParameter ->  exists (tP /

            -- si el tipo del parámetro es A o B, en classT serán
            -- del tipo AnInterface
            if (   sP.type.oclIsTypeOf(sourceA) or
                   sP.type.oclIsTypeOf(sourceB) )
            then tP.type.oclIsTypeOf(AnInterface)
            else  tP.type = sP.type
            endif     and

            -- el resto de las propiedades de los parámetros se mantienen
            tP.direction = sP.direction  and
            tP.defaultValue = defaultValue  and
            tP.isOrdered = sP.isOrdered  and
            tP.isUnique = sP.isUnique  and
            tP.upperValue = sP.upperValue  and
            tP.lowerValue = sP.lowerValue   ))  and

    -- el resto de las propiedades de la operación se mantienen
    tOp.name = sOp.name and
    tOp.class = sOp.class and
    tOp.isQuery = sOp.isQuery and
    tOp.precondition = sOp.precondition and
    tOp.postcondition = sOp.postcondition and
    tOp.bodyCondition = sOp.bodyCondition and
    tOp.raisedException = sOp. raisedException and
    tOp.redefinedOperation = sOp.redefinedOperation
    else

    -- la operación de la clase del paquete target se mantiene igual que
    -- en el paquete source
    tOp = sOp
    endif  ))


-- atributos heredados de NamedElement
classT.name = classS.name and

classT.visibility = classS.visibility and
```

```
                -- attributo heredado de Classifier
                classT.oclAsType(JavaClass).isAbstract =
                    classS.oclAsType(JavaClass).isAbstract and

                -- asociaciones heredadas de Namespace
                -- ambas clases tienen las mismas restricciones
                classT.oclAsType(JavaClass).ownedRule =
                    classS.oclAsType(JavaClass).ownedRule   and

                -- las clases tienen el mismo conjunto de elementos importados
                classT.oclAsType(JavaClass).elementImport =
                    classS.oclAsType(JavaClass).elementImport
                ))
        )))
}
```

5.2.1.6 Ejemplo

La figura 5.8.a muestra un modelo que contiene una clase *Employee* con atributos como el nombre, el departamento al que pertenece, el sueldo, si tiene algún tipo de experiencia y operaciones para obtener los valores de dichos atributos. La clase *Computer* hereda las características de *Electronic*, marca y garantía, y además tiene características propias como cpu, memoria, precio y capacidades. Posee además, operaciones para recuperar los valores de los atributos. Este modelo es una instancia del metamodelo origen del refactoring *Extraer Interfaz* (figura 5.6). La clase *Employee* del ejemplo es una instancia de la metaclase *A* del metamodelo origen. Las operaciones *getRate*() y *hasSpecialSkill*() de la clase *Employee* son instancias de la metaclase *A_Operation* ya que son operaciones sintácticamente equivalentes a las operaciones *getRate*() y *hasSpecialSkill*() (instancias de la metaclase *B_Operation*) de la clase *Computer* (instancia de la metaclase *B*)

La aplicación de esta regla conduce a:

- crear una interfaz Java *Billable*;
- declarar en la interfaz *Billable* las operaciones sintácticamente equivalentes, *getRate*() y *hasSpecialSkill*();
- reflejar que las clases *Employee* y *Computer* implementan la interfaz *Billable*;
- adaptar las declaraciones de tipo de los clientes (si hubiera en el modelo) para usar la interfaz.

El modelo resultante, figura 5.8.b, es una instancia del metamodelo destino del refactoring *Extraer Interfaz* (figura 5.7). La clase *Employee* es una instancia de la metaclase *A*, la clase *Computer* es una instancia de la metaclase *B* y la interfaz *Billable* es una instancia de la metaclase *AnInterface*.

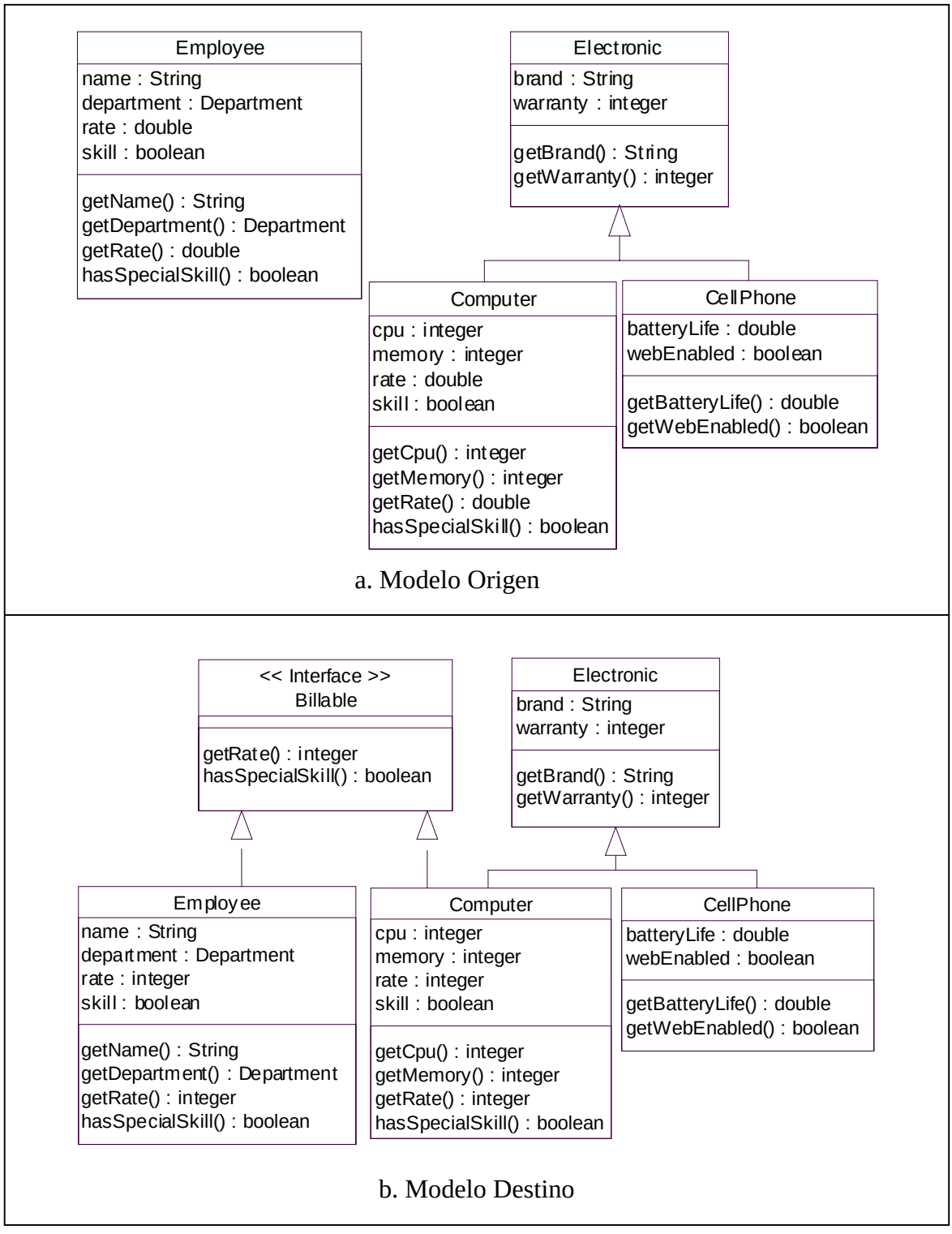


Figura 5.8 – Una instancia del refactoring *Extraer Interfaz*

CAPÍTULO 6

REFACTORING A NIVEL ISM

En este capítulo se exhiben ejemplos de refactorings aplicables a modelos específicos de la implementación en los lenguajes de programación C++ y Java. Se definen los metamodelos correspondientes a ambos lenguajes teniendo en cuenta las características particulares de cada uno.

El metamodelo ISM C++, ver Anexo D, refleja que un proyecto C++ consta de uno o varios archivo C++. Los archivos pueden ser de implementación o header y contienen declaraciones de variables y constantes globales, directivas al precompilador, declaraciones y definiciones de clasificadores y declaraciones y definiciones de funciones. Un clasificador puede ser un tipo de dato C++ o una clase C++. Una clase tiene variables y funciones miembros, puede contener además clases amigas y varias relaciones de generalización con otras clases. La implementación de las funciones consta de un cuerpo de una o varias sentencias.

El metamodelo ISM Java, ver Anexo E, refleja que un paquete Java contiene archivos, clases e interfaces. Una clase contiene campos 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 campos estáticos y finales, métodos sin implementación y puede heredar de varias interfaces.

6.1 Refactoring de ISMs C++

En el nivel de modelos dependientes de la implementación se especializa el metamodelo UML para reflejar las características de esta plataforma. Este metamodelo C++, a su vez, es especializado para cada transformación en particular, generando un metamodelo origen y un metamodelo destino por cada refactoring. Estos metamodelos son complementados con restricciones OCL.

6.1.1 Ejemplo: Minimizar dependencias de compilación entre archivos

Este refactoring está basado en una de las propuestas de Scott Meyers de mejorar los programas y diseños de C++ en su libro *Effective C++* (Meyers, 2005). El ejemplo se extrae del ítem 31 del capítulo 5 de Implementaciones.

6.1.1.1 Descripción

Un archivo que contiene la definición de una clase *A*, con directivas al precompilador para incluir archivos que contienen las definiciones de clases necesarias para poder compilar la clase A, produce una dependencia de compilación entre dichos archivos.

Estos archivos pueden desacoplarse generando archivos de interfaces y de implementación. La idea de esta separación es reemplazar las definiciones de clases con declaraciones de clases. Esto se consigue utilizando una simple estrategia:

- **utilizar referencias y punteros a objetos**: la definición de objetos de un tipo necesita para su compilación la definición del tipo, en cambio, la definición de referencias y punteros a un tipo necesita sólo la declaración del tipo;
- **utilizar declaraciones de clases**: la definición de una clase no se necesita para declarar una función que usa dicha clase, aún si la función pasa como parámetro o retorna el tipo de la clase por valor;
- **proveer archivos headers separados para declaraciones y definiciones**: los autores de librerías deberían proveer ambos archivos headers y los clientes de dicha librería deberían incluir el archivo de declaración.

Una alternativa a esta propuesta es que la clase *A* se divida en dos clases, comúnmente llamadas *Handle* y *Body*, donde la clase *Handle* ofrece sólo la interfaz y la clase *Body* implementa dicha interfaz. Estas clases soportan el *C++ Pimpl idiom* ("Private Implementation" idiom), método que oculta detalles de implementación moviendo miembros privados de la clase *A* a la clase *Body*, la clase *Handle* tendrá un puntero a dicha implementación. La figura 6.1 muestra esta propuesta.

6.1.1.2 Motivación

Una clase *A* no puede ser compilada a menos que el compilador tenga acceso a las definiciones de las clases en términos de las cuales está implementada. Tales definiciones son provistas a través de las directivas al precompilador para incluir los archivos headers que contienen las definiciones de clases. Esto genera una dependencia de compilación entre el archivo de la clase que se está definiendo y los archivos incluidos. Como consecuencia de esto, si una de estas clases auxiliares cambia su implementación, el archivo que contiene la clase *A* debe ser recompilado, como así también cualquier archivo que use la clase *A*. Esto es porque C++ permite poner detalles de implementación en las definiciones de clase. Para evitar este inconveniente, se deberían especificar los detalles de implementación en una clase separada, es decir, separando interfaces de implementaciones, esto hacen los lenguajes de programación como Smalltalk, Eiffel y Java.

Una alternativa para desacoplar interfaces de implementación es usar las clases *Handle-Body*. De esta manera los clientes de estas clases sólo deberían recompilar si cambian las interfaces de las mismas. Debido a que las interfaces tienden a estabilizarse antes que las implementaciones, esta técnica permite ahorrar tiempo de compilación y vinculación de archivos en el transcurso del desarrollo de grandes sistemas.

Cabe aclarar que esta técnica tiene un costo asociado. Las funciones miembros deberán navegar a través de un puntero a la implementación agregando un nivel de indirección por cada acceso. Se agrega a la cantidad de memoria requerida para almacenar cada objeto el tamaño del puntero a la implementación. Por último, el puntero a la implementación debe

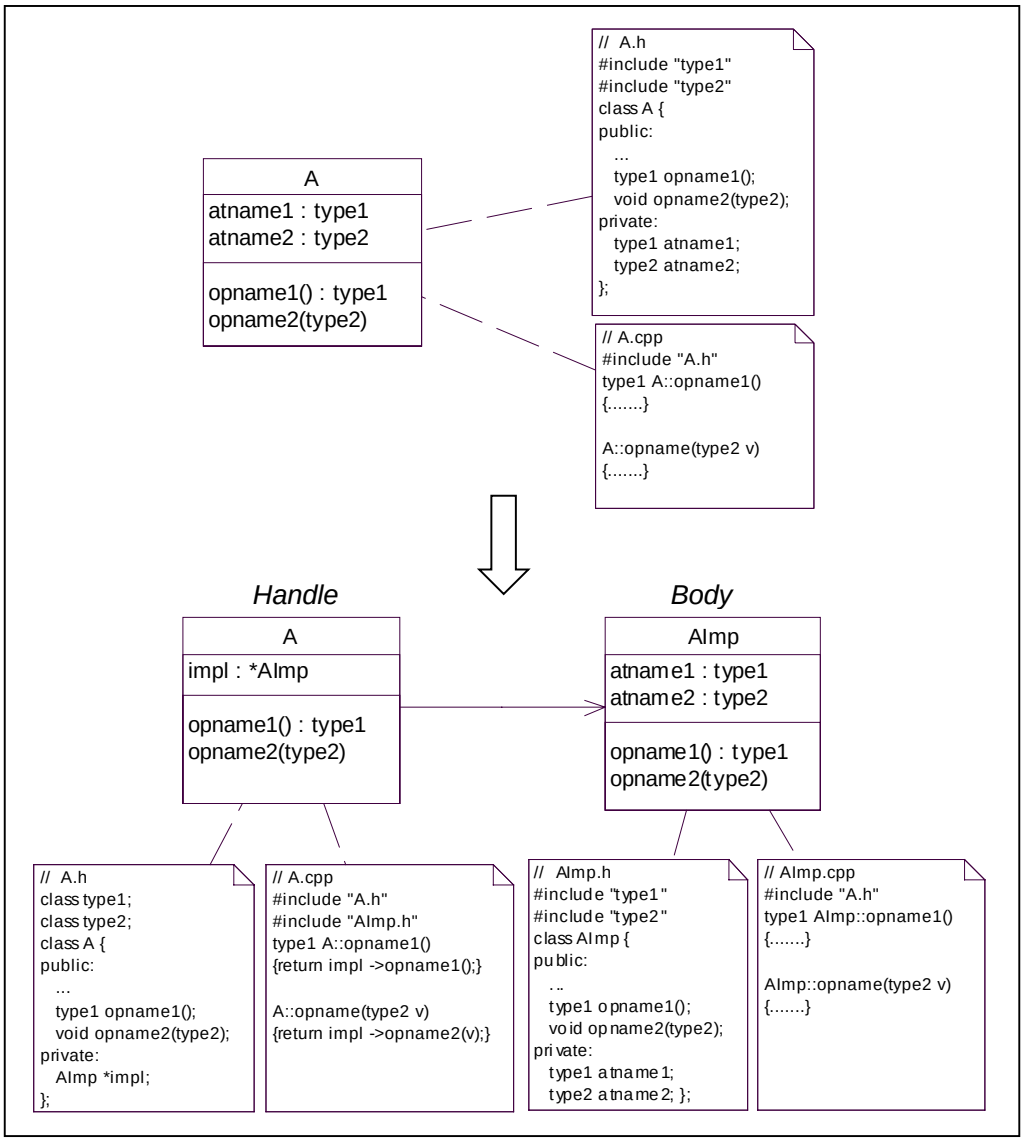


Figura 6.1 – Refactoring *Minimizar dependencias de compilación*

ser inicializado (en el constructor de la clase *Handle*) para que apunte a un objeto de implementación asignado dinámicamente, sobrecargando la administración de memoria dinámica.

Podrían desacoplarse las interfaces de la implementación durante la fase del desarrollo del sistema para minimizar el impacto sobre los clientes de dichas clases cuando éstas cambian su implementación. En la fase de producción podrían cambiarse por clases concretas cuando son significativas las diferencias de velocidad y/o tamaño como para justificar el incremento en las dependencias entre clases. Las herramientas deberían ser capaces de realizar este tipo de transformación automáticamente.

6.1.1.3 Metamodelo origen

Sintaxis abstracta

El metamodelo origen (figura 6.2) es una especialización del metamodelo ISM-C++.

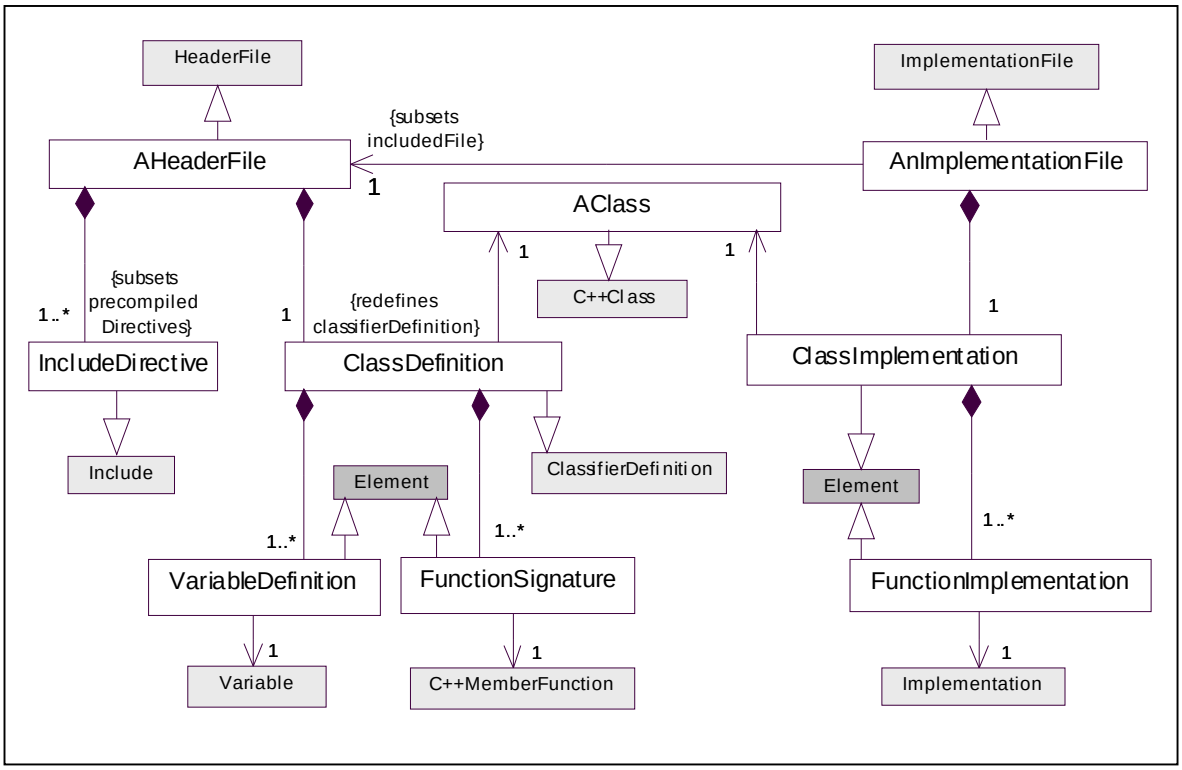


Figura 6.2 – Metamodelo origen del refactoring *Minimizar dependencias de compilación*

El metamodelo muestra las metaclases relacionadas a los participantes esenciales en los modelos a los cuales puede aplicarse este refactoring:

- *AClass* representa una clase C++,
- *AHeaderFile* representa un archivo header C++ que contiene la definición de la clase *AClass*, *ClassDefinition*, la cual consta de definiciones de variables miembros, *VariableDefinition*, y de signaturas de funciones, *FuncionSignature*,
- *IncludeDirective* representa a un subconjunto de directivas al precompilador asociadas a los tipos que intervienen en las definiciones de variables y funciones miembros de la clase,
- *AnImplementationFile* representa un archivo de implementación C++ que contiene la implementación de la clase *AClass*, *ClassImplementation*, la cual contiene la implementación de las funciones de la clase, *FunctionImplementation*,
- metaclases sombreadas en gris claro corresponden al metamodelo ISM-C++ y las metaclases sombreadas en gris oscuro corresponden al metamodelo UML.

El metamodelo origen sugiere que una instancia de *AHeaderFile* contiene la definición de una instancia de *AClass* la cual consta de la definición de por lo menos una variable y de una o más signaturas de funciones. Existe una instancia de *AnImplementationFile* que implementa el conjunto de funciones de la instancia de *AClass*. La instancia de *AnImplementationFile* contiene por lo menos una directiva al precompilador para incluir la instancia de *AHeaderFile* que contiene la definición de la clase que implementa (asociación entre *AnImplementationFile* y *AHeaderFile*).

Descripciones de metaclases

AClass

Representa una clase C++ para la cual existe un archivo header y un archivo de implementación que contienen la definición y la implementación de la clase respectivamente.

Generalizaciones

- C++Class, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

No posee asociaciones adicionales.

Restricciones

[1] Es una clase concreta.

```
not self.isAbstract
```

[2] No participa en jerarquías de herencia.

```
self.superClass --> size() = 0
```

AHeaderFile

Representa un archivo header C++ que contiene la definición de una clase.

Generalizaciones

- HeaderFile, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- includeDirective: IncludeDirective [1..*]

  Referencia el conjunto de directivas al precompilador asociadas con los tipos vinculados a las definiciones de variables y funciones. Subconjunto de *C++File::precompilerDirective.*

- classDefinition: ClassDefinition [1]

  Referencia a la definición de la clase. Redefine *C++File::classifierDefinition.*

Restricciones

[1] Por cada tipo User-Defined-Type usado en la definición de la clase, ya sea en la definición de variables o en las signaturas de funciones, existirá una directiva al precompilador para incluir el archivo header donde está definido dicho tipo.

```
let usedTypes: Set (C++Class) =
self.classDefinition.variableDefinition.variable.type -> union (
self.classDefinition.functionSignature.c++MemberFunction.returnType) -> union(
self.classDefinition.functionSignature.c++MemberFunction.parameter.type)
```

```
-> select (t| t.oclIsKindOf(User-DefinedType))
in
usedTypes -> forAll (t|
  self.includeDirective.headerFile -> exists (f|
      f.classifierDefinition.c++Class = t ))
```

[2] El archivo header, en la definición de clase, define un conjunto de variables que corresponden a las variables de la clase que define.

```
self.classDefinition.variableDefinition.variable = self.classDefinition.aclass.variable
```

[3] El archivo header, en la definición de clase, define un conjunto de signaturas de funciones que corresponden a las funciones miembros de la clase que define.

```
self.classDefinition.functionSignature.c++MemberFunction -> forAll( f |
  self.classDefinition.aClass.function -> exists (fc|
      f.hasSameSignature(fc) and f.body -> size() = 0 ))
```

[4] El archivo header, no tiene definiciones de funciones

```
self.functionDefinition -> isEmpty ()
```

AnImplementationFile

Representa un archivo de implementación C++ que contiene la implementación de las funciones de una clase.

Generalizaciones

- ImplementationFile, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- classImplementation: ClassImplementation [1]

  Referencia a la implementación de la clase.

- aHeaderFile: AHeaderFile [1]

  Referencia un archivo incluido mediante una directiva al precompilador. Subconjunto de *C++File::includedFile*.

Restricciones

[1] El archivo de implementación incluye el archivo header que contiene la definición de la clase que implementa.

```
self.aHeaderFile.classDefinition.aClass = self.classImplementation.aClass
```

[2] El archivo de implementación, en la implementación de la clase, implementa un conjunto de funciones que corresponden a las funciones miembros de la clase que implementa.

```
self.classImplementation.functionImplementation.implementation.function -> forAll( f |
self.classImplementation.aclass.function -> exists (fc|
      f.hasSameSignature(fc) and f.body -> size() = 1 ))
```

[3] El archivo de implementación no tiene declaraciones de variables y constantes globales.

```
self.globalVariableandConstantDeclaration -> isEmpty()
```

[4] El archivo de implementación no tiene declaraciones ni definiciones de clases.

```
self.classifierDeclaration -> isEmpty()
self.classifierDefinition -> isEmpty()
```

[5] En el archivo de implementación, las definiciones de funciones corresponden a las funciones de la clase que implementa.

```
self.functionDefinition.c++Function = self.classImplementation.aClass.function
```

ClassDefinition

Representa la definición de una clase contenida en un archivo header.

Generalizaciones

- ClassifierDefinition, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- aHeaderFile: AHeaderFile [1]

  Referencia al archivo header que posee esta definición de clase.

- aClass: AClass [1]

  Referencia a la clase que define.

- variableDefinition: VariableDefinition [1..*]

  Referencia las definiciones de variables de la clase.

- functionSignature: FunctionSignature [1..*]

  Referencia las signaturas de las funciones de la clase.

Restricciones

No posee restricciones adicionales.

ClassImplementation

Representa la implementación de una clase contenida en un archivo de implementación.

Generalizaciones

- Element, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- aClass: AClass [1]

  Referencia a la clase que implementa.

- anImplementationFile: AnImplementationFile [1]

Referencia al archivo de implementación que posee esta implementación de clase.

- functionImplementation: FunctionImplementation [1..*]

  Referencia las implementaciones de las funciones de la clase.

Restricciones

No posee restricciones adicionales.

IncludeDirective

Representa un subconjunto de directivas al precompilador para incluir los archivos de los tipos utilizados en la definición de la clase.

Generalizaciones

- Include, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- aHeaderFile: AHeaderFile [1]

  Referencia al archivo que contiene la directiva.

Restricciones

No posee restricciones adicionales.

FunctionImplementation

Representa la implementación de las funciones de la clase.

Generalizaciones

- Element, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- classImplementation: ClassImplementation [1]

  Referencia a la implementación de la clase que contiene la función.

- implementation: Implementation [1]

  Referencia a la implementación de la función.

Restricciones

No posee restricciones adicionales.

FunctionSignature

Representa las signaturas de las funciones de la clase.

Generalizaciones

- Element, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- classDefinition: ClassDefinition [1]

  Referencia a la definición de la clase que contiene la signatura de función.

- c++MemberFunction: C++MemberFunction [1]

  Referencia a la función miembro que declara.

Restricciones

No posee restricciones adicionales.

VariableDefinition

Representa las definiciones de variables de la clase.

Generalizaciones

- Element, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- classDefinition: ClassDefinition [1]

  Referencia a la definición de la clase que contiene la variable.

- variable: Variable [1]

  Referencia a la variable que define.

Restricciones

No posee restricciones adicionales.

6.1.1.4 Metamodelo destino

Sintaxis abstracta

El metamodelo destino es una especialización del metamodelo ISM-C++ y se presenta en la figura 6.3 a través de los diagramas:

- de las metaclases *HandleHeaderFile* y *HandleImplementationFile*,
- de las metaclases *BodyHeaderFile* y *BodyImplementationFile*.

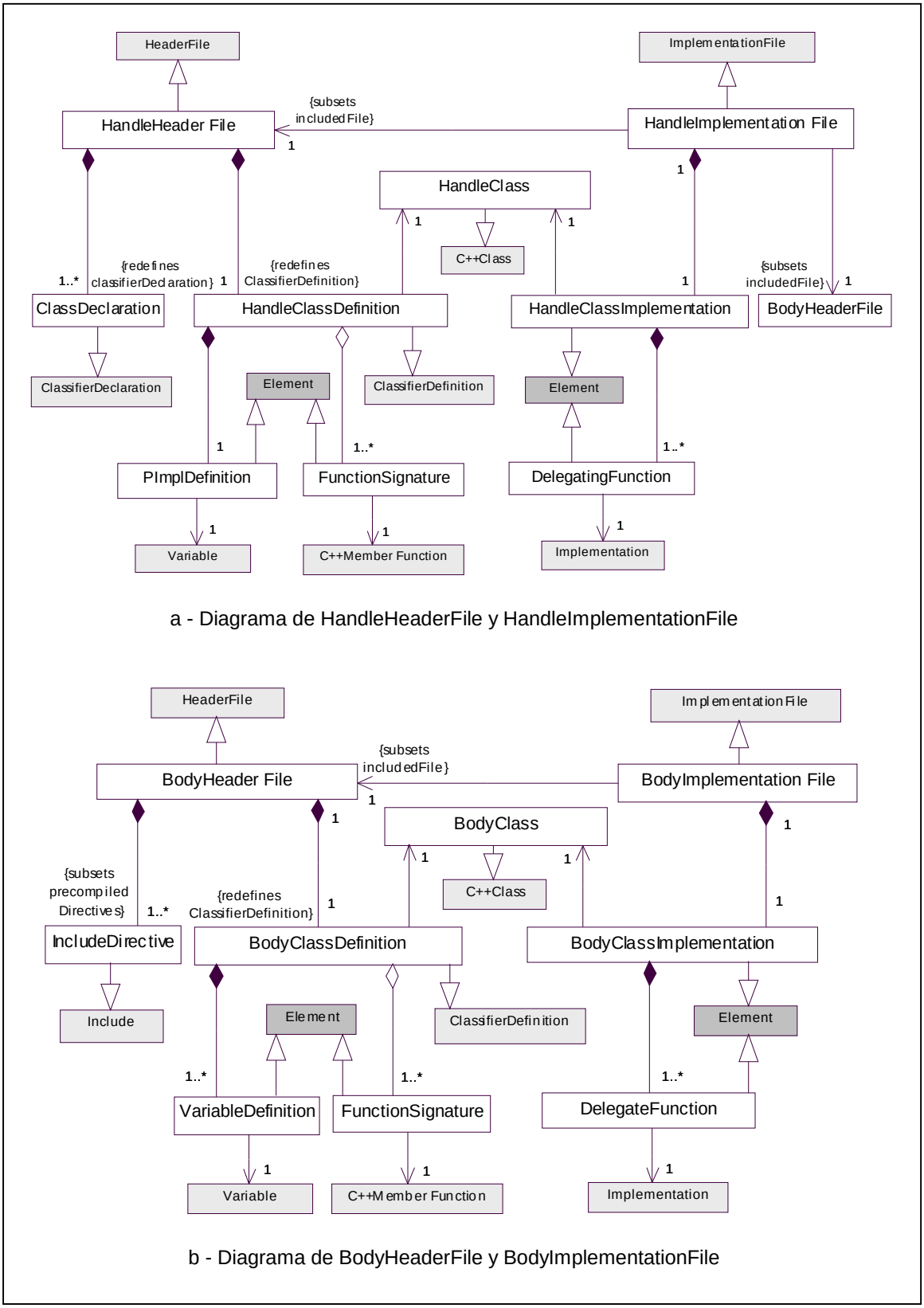


Figura 6.3 – Metamodelo destino del refactoring *Minimizar dependencias de compilación*

El metamodelo muestra las metaclases relacionadas a los participantes esenciales en los modelos generados al aplicarse este refactoring:

- *HandleHeaderFile* y *HandleImplementationFile* son los archivos header y de implementación que contienen la definición e implementación de *HandleClass*;
- *BodyHeaderFile* y *BodyImplementationFile* son los archivos header y de implementación que contienen la definición e implementación de *BodyClass*;

Las diferencias entre el metamodelo origen y destino sugieren lo siguiente:

- en un modelo origen, los dos archivos (*header* y de implementación) que definen una clase (*AClass*) contienen detalles de implementación;
- en un modelo destino, existirán dos clases (*HandleClass* y *BodyClass*) cada una con sus respectivos archivos *header* y de implementación, donde *HandleClass* cumple el rol de interfaz y *BodyClass* contendrá la implementación.

Descripciones de metaclases

BodyClass

Representa una clase C++ que implementa una interfaz.

Generalizaciones

- C++Class, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

No posee asociaciones adicionales.

Restricciones

No posee restricciones adicionales.

BodyClassDefinition

Representa la definición de una clase del tipo BodyClass contenida en un archivo header.

Generalizaciones

- ClassifierDefinition, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- bodyHeaderFile: BodyHeaderFile [1]

  Referencia al archivo header que posee esta definición de clase.

- bodyClass: BodyClass [1]

Referencia a la clase que define.

- variableDefinition: VariableDefinition [1..*]

  Referencia las definiciones de variables de la clase.

- functionSignature: FunctionSignature [1..*]

  Referencia las signaturas de las funciones de la clase.

Restricciones

No posee restricciones adicionales.

BodyClassImplementation

Representa la implementación de una clase del tipo BodyClass contenida en un archivo de implementación.

Generalizaciones

- Element, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- bodyClass: BodyClass [1]

  Referencia a la clase que implementa.

- bodyImplementationFile: BodyImplementationFile [1]

  Referencia al archivo de implementación que posee esta implementación de clase.

- delegateFunction: DelegateFunction [1..*]

  Referencia las implementaciones de las funciones de la clase.

Restricciones

No posee restricciones adicionales.

BodyHeaderFile

Representa un archivo header C++ que contiene la definición de una clase del tipo BodyClass.

Generalizaciones

- HeaderFile, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- includeDirective: IncludeDirective [1..*]

Referencia el conjunto de directivas al precompilador asociadas con los tipos vinculados a las definiciones de variables y funciones. Subconjunto de *C++File::precompilerDirective*.

- bodyClassDefinition: BodyClassDefinition [1]

Referencia a la definición de la clase. Redefine *C++File::classifierDefinition*.

Restricciones

[1] Por cada tipo C++Class usado en la definición de la clase, ya sea en la definición de variables o en las signaturas de funciones, existirá una directiva al precompilador para incluir el archivo header donde está definido dicho tipo.

```
let usedTypes: Set (C++Class) =
self.bodyClassDefinition.variableDefinition.variable.type -> union
(self.bodyClassDefinition.functionSignature.c++MemberFunction.returnType) -> union(
self.bodyClassDefinition.functionSignature.c++MemberFunction.parameter.type)
-> select (t| t.oclIsKindOf(C++Class))
in
usedTypes -> forAll (t|
  self.includeDirective.headerFile -> exists (f|
      f.classifierDefinition.c++Class = t ))
```

[2] El archivo header, en la definición de clase, define un conjunto de variables que corresponden a las variables de la clase que define.

```
self.bodyClassDefinition.variableDefinition.variable =
  self.bodyClassDefinition.bodyClass.variable
```

[3] El archivo header, en la definición de clase, define un conjunto de signaturas de funciones que corresponden a las funciones miembros de la clase que define.

```
self.bodyClassDefinition.functionSignature.c++MemberFunction -> forAll( f |
self.bodyClassDefinition.bodyClass.function -> exists (fc|
      f.hasSameSignature(fc) and f.body -> isEmpty() ))
```

BodyImplementationFile

Representa un archivo de implementación C++ que contiene la implementación de las funciones de una clase de tipo BodyClass.

Generalizaciones

- ImplementationFile, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- bodyClassImplementation: BodyClassImplementation [1]

  Referencia a la implementación de la clase.

- bodyHeaderFile: BodyHeaderFile [1]

Referencia al archivo incluido mediante una directiva al precompilador. Subconjunto de *C++File::includedFile*.

Restricciones

[1] El archivo de implementación, en la implementación de la clase, implementa un conjunto de funciones que corresponden a las funciones miembros de la clase que implementa.

```
self.bodyClassImplementation.delegateFunction.implementation.function ->
  forAll( f |  self.bodyClassImplementation.bodyClass.function -> exists (fc|
      f.hasSameSignature(fc) and f.body -> size() = 1 ))
```

ClassDeclaration

Representa un conjunto de declaraciones de clase de los tipos utilizados en la definición de la clase.

Generalizaciones

- ClassifierDeclaration, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- handleHeaderFile: HandleHeaderFile [1]

  Referencia al archivo que contiene las declaraciones. Redefine *C++File::classifierDeclaration*.

Restricciones

No posee restricciones adicionales.

DelegateFunction

Representa la implementación de las funciones de la clase.

Generalizaciones

- Element, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- bodyClassImplementation: BodyClassImplementation [1]

  Referencia a la implementación de la clase que contiene las funciones.

- implementation: Implementation [1]

  Referencia a la implementación de la función.

Restricciones

No posee restricciones adicionales.

DelegatingFunction

Representa las funciones de la clase que delegan el comportamiento a un objeto del tipo *BodyClass*.

Generalizaciones

- Element, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- handleClassImplementation: HandleClassImplementation [1]

  Referencia a la implementación de la clase que contiene las funciones.

- implementation: Implementation [1]

  Referencia a la implementación de la función.

Restricciones

[1] La función en su implementación contiene la invocación de un método en el cual delega el comportamiento.

```
self.implementation.invokedMethod -> size() = 1
```

[2] La función tiene la misma signatura que el método invocado en el cuerpo de la función.

```
self.implementation.function.hasSameSignature(self.body.invokedMethod)
```

[3] El método invocado pertenece a BodyClass.

```
self.implementation.invokedMethod.class = BodyClass
```

FunctionSignature

Representa las signaturas de las funciones de la clase.

Generalizaciones

- Element, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- bodyClassDefinition: BodyClassDefinition [1]

  Referencia a la definición de *BodyClass* que contiene las signaturas de funciones.

- handleClassDefinition: HandleClassDefinition [1]

  Referencia a la definición de *HandleClass* que contiene las signaturas de funciones.

- c++MemberFunction: C++MemberFunction [1]

  Referencia a la función miembro que declara.

Restricciones

No posee restricciones adicionales.

HandleClass

Representa una clase C++ que cumple el rol de interfaz.

Generalizaciones

- C++Class, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

No posee asociaciones adicionales.

Restricciones

No posee restricciones adicionales.

HandleClassDefinition

Representa la definición de una clase del tipo *HandleClass* contenida en un archivo header.

Generalizaciones

- ClassifierDefinition, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- handleHeaderFile: HandleHeaderFile [1]

  Referencia al archivo header que posee esta definición de clase.

- handleClass: HandleClass [1]

  Referencia a la clase que define.

- pImplDefinition: PImplDefinition [1]

  Referencia la definición de la variable puntero a la implementación.

- functionSignature: FunctionSignature [1..*]

  Referencia las signaturas de las funciones de la clase.

Restricciones

No posee restricciones adicionales.

HandleClassImplementation

Representa la implementación de una clase del tipo *HandleClass* contenida en un archivo de implementación.

Generalizaciones

- Element, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- handleClass: HandleClass [1]

  Referencia a la clase que implementa.

- handleImplementationFile: HandleImplementationFile [1]

  Referencia al archivo de implementación que posee esta implementación de clase.

- delegatingFunction: DelegatingFunction [1..*]

  Referencia a las funciones de delegación.

Restricciones

No posee restricciones adicionales.

HandleHeaderFile

Representa un archivo header C++ que contiene la definición de una clase del tipo HandleClass.

Generalizaciones

- HeaderFile, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- classDeclaration: ClassDeclaration [1..*]

  Referencia el conjunto de declaraciones de clases asociadas con los tipos vinculados a las definiciones de variables y funciones. Redefine *C++File::classifierDeclaration*.

- handleClassDefinition: HandleClassDefinition [1]

  Referencia a la definición de la clase. Redefine *C++File::classifierDefinition*.

Restricciones

[1] El archivo contendrá una declaración de clase por cada clase definida por el usuario usada en las signaturas de las funciones de la definición de la clase *Handle*.

```
let usedTypes: Set (C++Class) =
self.handleClassDefinition.functionSignature.c++MemberFunction.returnType) -> union(
self.handleClassDefinition.functionSignature.c++MemberFunction.parameter.type)
```

```
-> select (t| t.oclIsKindOf(User-DefinedType))
in
usedTypes -> forAll (t|
  self.classDeclaration.c++Classifier -> exists ( c| c = t )))
```

[2] El archivo contendrá una declaración de la clase *BodyClass*.

```
self.classDeclaration.c++Classifier -> includes (BodyClass)
```

[3] En la definición de la clase *Handle* existirá una variable que será un puntero a la clase implementación, es decir, a *BodyClass.*

```
self.handleClassDefinition.pImplDefinition.variable.var-type = #pointer and
self.handleClassDefinition.pImplDefinition.variable.type = BodyClass
```

HandleImplementationFile

Representa un archivo de implementación C++ que contiene la implementación de las funciones de una clase de tipo HandleClass.

Generalizaciones:

- ImplementationFile, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- handleClassImplementation: HandleClassImplementation [1]

  Referencia a la implementación de la clase.

- bodyHeaderFile: BodyHeaderFile [1]

  Referencia al archivo incluido mediante una directiva al precompilador. Subconjunto de *C++File::includedFile*.

- handleHeaderFile: HandleHeaderFile [1]

  Referencia al archivo incluido mediante una directiva al precompilador. Subconjunto de *C++File::includedFile*.

Restricciones

[1] El archivo de implementación, en la implementación de la clase Handle, implementa un conjunto de funciones que corresponden a las funciones miembros de la clase que implementa, las cuales son funciones de delegación.

```
self.handleClassImplementation.delegatingFunction.implementation.function -> forAll
  ( f |  self.handleClassImplementation.handleClass.function -> exists (fc|
       f.hasSameSignature(fc) and f.body -> size() = 1 and
       f.body.invokedMethod -> size() =1 and
       f.body.invokedMethod.hasSameSignature(f) and
       f.body.invokedMethod.class = BodyClass))
```

IncludeDirective

Representa un subconjunto de directivas al precompilador para incluir los archivos de los tipos utilizados en la definición de la clase.

Generalizaciones

- Include, del metamodelo ISM-C++.

Atributos

No posee atributos adicionales.

Asociaciones

- bodyHeaderFile: BodyHeaderFile [1]

  Referencia al archivo que contiene la directiva.

Restricciones

No posee restricciones adicionales.

PImplDefinition

Representa la definición de una variable de la clase que es puntero a la implementación.

Generalizaciones

- Element, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- handleClassDefinition: HandleClassDefinition [1]

  Referencia a la definición de la clase que contiene la variable.

- variable: Variable [1]

  Referencia a la variable que define.

Restricciones

[1] El tipo de la variable definida es *BodyClass*.

```
self.variable.type = BodyClass
```

VariableDefinition

Representa las definiciones de variables de la clase.

Generalizaciones

- Element, del metamodelo UML.

Atributos

No posee atributos adicionales.

Asociaciones

- bodyClassDefinition: BodyClassDefinition [1]

  Referencia a la definición de la clase que contiene la variable.

- variable: Variable [1]

Referencia a la variable que define.

Restricciones

No posee restricciones adicionales.

6.1.1.5 Regla de transformación

La regla *Minimizar dependencias de compilación entre archivos* se aplica sobre aquellos archivos headers que contienen detalles de implementación de una clase, provocando dependencias de compilación entre dicho archivo y los que contienen la definición de los tipos usados en la clase. Este refactoring genera dos clases, interfaz e implementación, por cada clase del modelo original, generando un archivo header por cada una de ellas. De esta manera, se obtiene un archivo header para declaraciones y otro para definiciones. Los clientes deberán incluir sólo el archivo header de declaraciones y de esta manera evitar la dependencia entre los archivos.

Los pasos para aplicar esta regla son los siguientes:

1. Generar dos clases *Handle* y *Body*, interfaz e implementación respectivamente;
2. Generar un archivo header para la clase *Handle* que contenga las declaraciones de las clases usadas, signaturas de funciones y un puntero a la implementación de la clase, (puntero a la clase *Body*);
3. Generar un archivo de implementación de la clase *Handle* que contenga funciones que delegan su comportamiento en funciones de la clase *Body*;
4. Generar un archivo header para la clase *Body* que contenga directivas *#include* para los tipos de datos utilizados, signaturas de funciones y los datos privados definidos en la clase original;
5. Generar un archivo de implementación de la clase *Body* que contengan la implementación de las funciones como estaban en la clase original.
6. Actualizar los clientes para que incluyan en sus archivos al archivo header de declaraciones, es decir, el archivo header de la clase *Handle*.

A continuación, se especifican parcialmente estos cambios a través de la regla de transformación.

Transformation Minimizar dependencias entre archivos **{**

parameter
source: Metamodelo Origen Minimizar dependencias:: C++Project
target: Metamodelo Destino Minimizar dependencias:: C++Project

local operations

postconditions
post
-- Por cada archivo *sourceFile*, de tipo *AHeaderFile*, del proyecto source,
source.C++File -> select(oclIsTypeOf(AHeaderFile)) -> forAll (sourceFile |

-- existe un archivo *targetFile*, de tipo *HandleHeaderFile*, en el proyecto target tal que,
target.C++File -> select(oclIsTypeOf(HandleHeaderFile)) -> exists (targetFile |

```
    -- tienen el mismo conjunto de declaraciones de variables y constantes globales
    targetFile.globalVariableandConstantDeclaration =
        sourceFile.globalVariableandConstantDeclaration   and

    -- el conjunto de directivas #include de sourceFile, para incluir tipos usados en
    -- la definición de clase, se corresponden con declaraciones de tipos en
    -- targetFile,
    sourceFile.includeDirective -> forAll (include |
        targetFile.classDeclaration -> exists ( declaration |
            include.headerFile.classifierDefinition.c++Class =
                declaration.c++Classifier)  )       and

    -- las directivas #include que no pertenecen al conjunto de IncludeDirective
    -- están incluidas en targetFile.
    targetFile.precompilerDirective -> includes (
        sourceFile.precompilerDirective -> excluding
            (sourceFile.includeDirective))

     -- tienen el mismo conjunto de directivas al precompilador (excluyendo las
     -- directivas #include)
    targetFile.precompilerDirective -> select (oclIsTypeOf(OtherDirective)) =
        sourceFile.precompilerDirective -> select (oclIsTypeOf(OtherDirective))  and

    -- tienen el mismo conjunto de declaraciones de clasificadores,
    targetFile.classifierDeclaration = sourceFile.classifierDeclaration  and

    -- las definiciones de variables en sourceFile son reemplazadas por una
    -- variable puntero a la implementación en targetFile,
    targetFile.handleClassDefinition.pImplDefinition.variable -> excludes(
        sourceFile.classDefinition.variableDefinition.variable )  and
    targetFile.handleClassDefinition.pImplDefinition.variable -> size() = 1

    -- tienen el mismo conjunto de signaturas de funciones,
    targetFile.handleClassDefinition.functionSignture.c++MemberFunction =
        sourceFile.classDefinition.functionSignature.c++MemberFunction )


-- y existe un archivo BodyHeaderFile en el proyecto target tal que,
target.C++File -> select(oclIsTypeOf(BodyHeaderFile)) -> exists ( targetFile |

    -- tienen el mismo conjunto de declaraciones de variables y constantes globales
    targetFile.globalVariableandConstantDeclaration =
        sourceFile.globalVariableandConstantDeclaration   and

    -- tienen el mismo conjunto de directivas al precompilador
    targetFile.precompilerDirective = sourceFile.precompilerDirective  and

    -- tienen el mismo conjunto de declaraciones de clases,
    targetFile.classifierDeclaration = sourceFile.classifierDeclaration  and
```

```
        -- tienen el mismo conjunto de definiciones de variables
        targetFile.bodyClassDefinition.variableDefinition.variable =
            sourceFile.classDefinition.variableDefinition.variable  and

        -- y tienen el mismo conjunto de signaturas de funciones.
        targetFile.bodyClassDefinition.functionSignature.c++MemberFunction =
            sourceFile.classDefinition.functionSignature.c++MemberFunction ) )
```

post

```
-- Por cada archivo AnImplementationFile del proyecto source,
source.c++File -> select(oclIsTypeOf(AnImplementationFile)) -> forAll (
sourceFile |

    -- existe un archivo HandleImplementationFile en el proyecto target tal que,
    target.c++File -> select(oclIsTypeOf(HandleImplementationFile)) ->
      exists ( targetFile |

        -- targetFile no tiene declaraciones de variables y constantes globales
        targetFile.globalVariableandConstantDeclaration -> isEmpty()  and

        -- targetFile no tiene declaraciones ni definiciones de clases,
        targetFile.classifierDeclaration -> isEmpty()   and
        targetFile.classifierDefinition -> isEmpty()  and

         -- targetFile incluye al archivo BodyHeaderFile y al archivo
         -- HandleHeaderFile que define a la clase que implementa
         targetFile.includedFile -> includes (BodyHeaderFile)  and
         targetFile.includedFile -> includes (HandleHeaderFile)  and
          targetFile.handleHeaderFile.handleClassDefinition.handleClass =
              targetFile.handleClassImplementation.handleClass    and

        -- tienen el mismo conjunto de directivas al precompilador (excluyendo las
        -- directivas #include)
        target.precompilerDirectives -> select (oclIsTypeOf(OtherDirective)) =
            source.precompilerDirectives -> select (oclIsTypeOf(OtherDirective))

        -- por cada función de sourceFile existirá en targetFile una función con la
        -- misma signatura que delega su comportamiento a la clase BodyClass
        sourceFile.classImplementation.functionImplementation
            -> forAll( fs |
              targetFile.handleClassImplementation.delegatingFunction
              -> exists (ft|
                    if fs.implementation.function.oclIsTypeOf(Constructor)
                    then
                        ft.implementation.function.oclIsTypeOf(Constructor)  and
                        ft.implementation.statement.oclIsTypeOf(Assignment) and
```

```
                    ft.implementation.statement.exp1.oclIsTypeOf(Identifier)   and

                    ft.implementation.statement.exp1 = ft.handleClassImplementation.
                    handleImplementationFile. handleHeaderFile.
                    handleClassDefinition.implDefinition.variable.name   and

                    ft.implementation.statement.assignmentOperator = #=   and
                    ft.implementation.statement.exp2.oclIsTypeOf(CreateObject)   and
                    ft.implementation.statement.exp1.classifier = BodyClass

                else
                    ft.implementation.function.hasSameSignature(fs.function)  and
                    ft.implementation.invokedMethod.class = BodyClass  and
                    ft.implementation.invokedMethod.hasSameSignature(fs.function)
                endif   ))  )   and

    -- y existe un archivo BodyImplementationFile en el proyecto target tal que,
    target.c++File -> select(oclIsTypeOf(BodyImplementationFile)) ->
      exists ( targetFile |

        -- targetFile no tiene declaraciones de variables y constantes globales
        targetFile.globalVariableandConstantDeclaration -> isEmpty()  and

        -- targetFile no tiene declaraciones ni definiciones de clases,
        targetFile.classifierDeclaration -> isEmpty()  and
        targetFile.classifierDefinition -> isEmpty()  and

        -- targetFile incluye al archivo BodyHeaderFile que define a la clase que
        -- implementa
        targetFile.includedFile -> includes (BodyHeaderFile)  and
         targetFile.bodyHeaderFile.bodyClassDefinition.bodyClass =
             targetFile.bodyClassImplementation.bodyClass    and

        -- tienen el mismo conjunto de directivas al precompilador (excluyendo las
        -- directivas #include)
        target.precompilerDirectives -> select (oclIsTypeOf(OtherDirective)) =
             source.precompilerDirectives -> select (oclIsTypeOf(OtherDirective))

        -- tienen el mismo conjunto de implementación de funciones
         targetFile.bodyClassImplementation.delegateFunction.implementation =
              sourceFile.classImplementation.functionImplementation.implementation )  )
...
}
```

6.1.1.6 Ejemplo

La figura 6.4.a muestra la clase *Person* con sus miembros públicos y privados y dos archivos, header y de implementación, que contienen la definición e implementación de *Person* respectivamente. El archivo header *Person.h* contiene varias directivas al

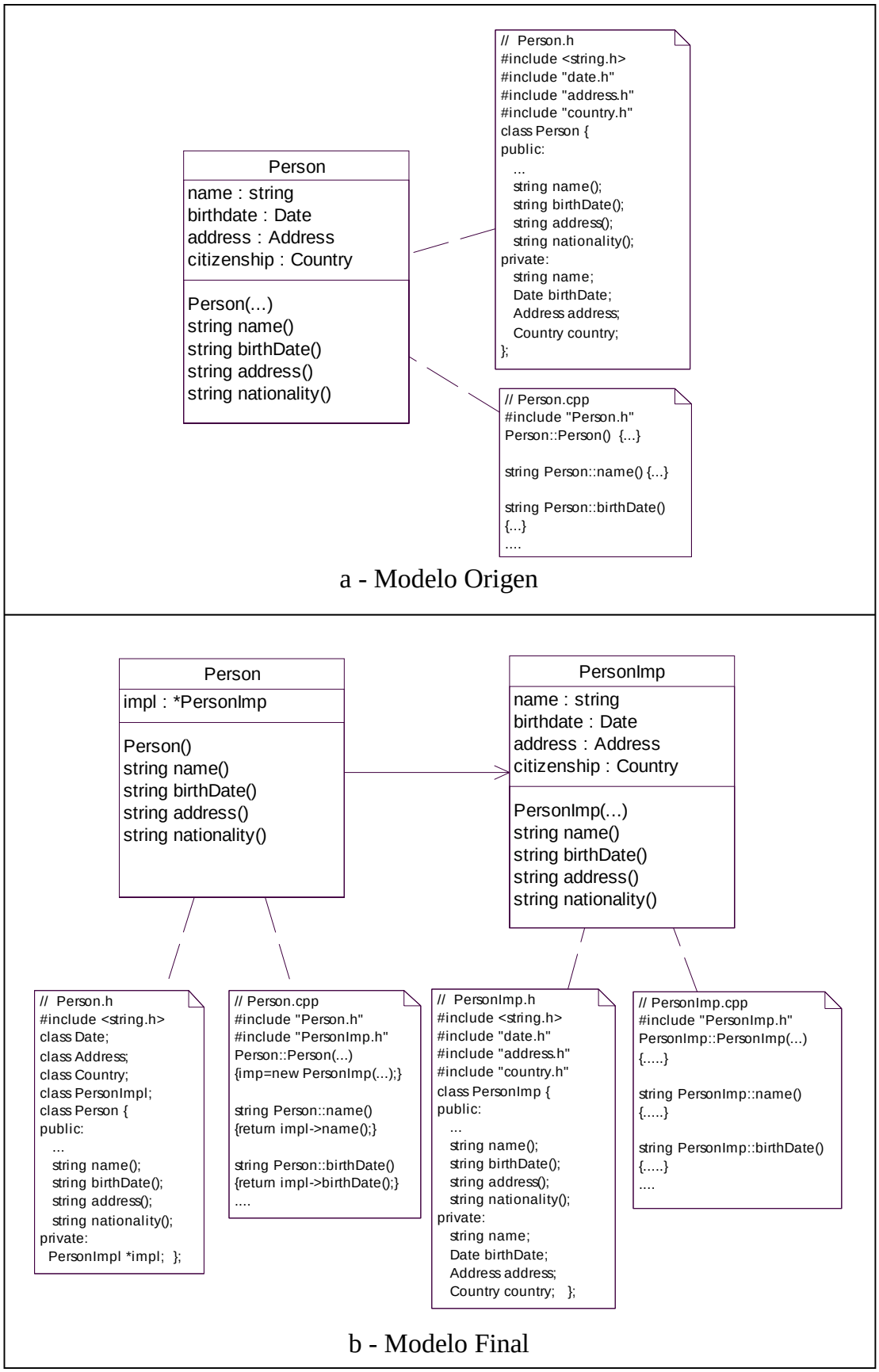


Figura 6.4 – Una instancia del refactoring Minimizar dependencias de compilación

precompilador para incluir las definiciones de los tipos usados, ya que al definir objetos de un tipo como miembros privados de la clase se necesita la definición de dicho tipo. El archivo *Person.cpp* contiene las implementaciones de las funciones de *Person*.

Este modelo es una instancia del metamodelo origen del refactoring *Minimizar dependencias de compilación* (figura 6.2). La clase *Person* del ejemplo es una instancia de la metaclase *AClass* del metamodelo origen. Los archivos *Person.h* y *Person.cpp* son instancias de *AHeaderFile* y de *AnImplementationFile* respectivamente. El archivo *Person.h* contiene directivas #include para incluir los archivos headers de *date*, *address* y *country* (subconjunto del total de directivas que tiene el archivo), las cuales son instancias de *IncludeDirective*. Además, contiene definiciones de variable como miembros privados de la clase que son instancias de *VariableDefinition*, y funciones públicas que son instancias de *FunctionSignature*. El archivo *Person.cpp* contiene una directiva #include para incluir *Person.h* (representada en el metamodelo a través de la asociación unidireccional *AnImplementationFile-AHeaderFile*), y las implementaciones de las funciones de la clase, cada una de ellas es una instancia de *FunctionImplementation*.

La aplicación de esta regla conduce a:

- crear dos clases *Person* y *PersonImp* que cumplirán el rol de interfaz e implementación de la clase *Person* del modelo original;
- crear dos archivos, header y de implementación, para la clase *Person*;
- crear dos archivos, header y de implementación, para la clase *PersonImp*;
- incluir en el archivo *Person.h* las declaraciones de tipos usados en las signaturas de funciones de la clase, las signaturas de funciones de la clase y la estructura privada de la clase que será una variable puntero a la implementación;
- incluir en el archivo *Person.cpp* funciones que delegan su comportamiento a las correspondientes funciones de *PersonImp*;
- incluir en el archivo *PersonImp.h* las directivas #include para incluir los tipos correspondientes a las definiciones de variables de la estructura privada de la clase, la definición de variables privadas y funciones públicas;
- incluir en el archivo *PersonImp.cpp* las funciones de la clase con la implementación que tiene en la clase *Person* del modelo original;

El modelo resultante se muestra en la figura 6.4.b. La clase *Person*, instancia de la metaclase *HandleClass*, cumple el rol de la clase interfaz. El archivo header *Person.h* y el archivo de implementación *Person.cpp* son instancias de *HandleHeaderFile* y *HandleImplementationFile* respectivamente. La clase *PersonImp*, instancia de la metaclase *BodyClass*, cumple el rol de la clase de implementación. El archivo header *PersonImp.h* y el archivo *PersonImp.cpp* son instancias de *BodyHeaderFile* y de *BodyImplementationFile* respectivamente.

6.2 Refactoring de ISMs Java

En el nivel de modelos dependientes de la implementación en lenguaje Java, se especializa el metamodelo UML para reflejar las características de este lenguaje. Este metamodelo Java, a su vez, es especializado para cada transformación en particular, generando un

metamodelo origen y un metamodelo destino por cada refactoring, complementados con restricciones OCL.

6.2.1 Ejemplo: Adaptar Interfaz

Este refactoring está basado en el autor Joshua Kerievsky, quien en su libro *Refactoring To Patterns* (Kerievsky, 2004) propone un refactoring llamado *Adapt Interface* aplicable a código Java.

6.2.1.1 Descripción

Cuando en un modelo una clase implementa una interfaz pero sólo provee código para algunos de los métodos de la interfaz, la aplicación de la regla *Adaptar Interfaz* permite reestructurar este tipo de código a través del patrón *Adapter* (figura 6.5). 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.

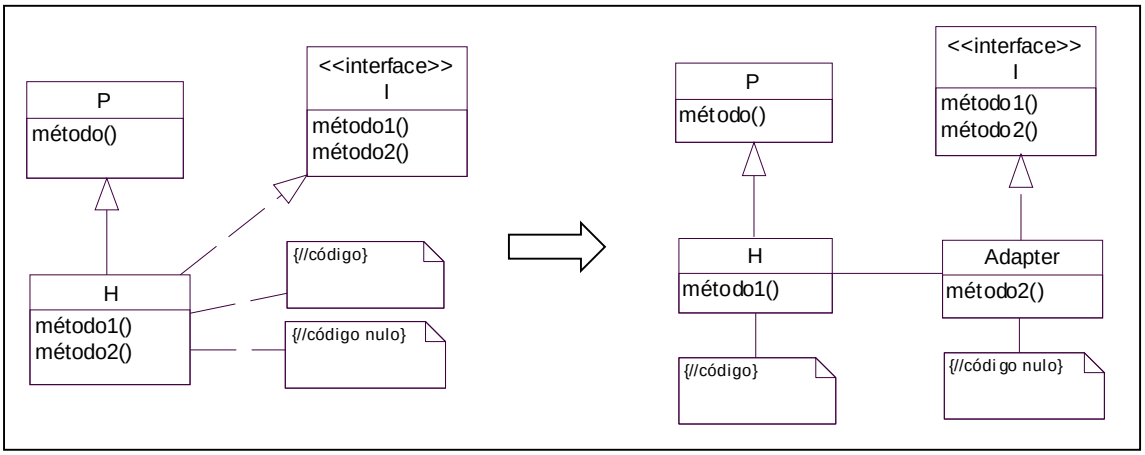


Figura 6.5 – Refactoring *Adaptar Interfaz*

6.2.1.2 Motivación

Por lo general, una clase concreta 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.

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.

6.2.1.3 Metamodelo origen

Sintaxis abstracta

El metamodelo origen es una especialización del metamodelo ISM-JAVA. La figura 6.6 del metamodelo origen muestra las metaclases relacionadas a los participantes esenciales en los modelos a los cuales puede aplicarse el refactoring *adaptar interfaz*:

- *AClass* representa una clase Java, *AnInterface* representa una interfaz Java,
- *Class-Interface-Relation* representa una relación de realización de interfaz entre *AClass* y *AnInterface*,
- *OperationWithCode* representa una operación Java de *AClass* cuya implementación no tiene ninguna sentencia y *OperationWithoutCode* representa una operación Java de *AClass* con implementación,
- *AMethod* representa un método de una interfaz,
- metaclases sombreadas en gris claro corresponden al metamodelo ISM-Java,
- metaclase sombreada en gris oscuro (InterfaceRealization) corresponde al metamodelo UML.

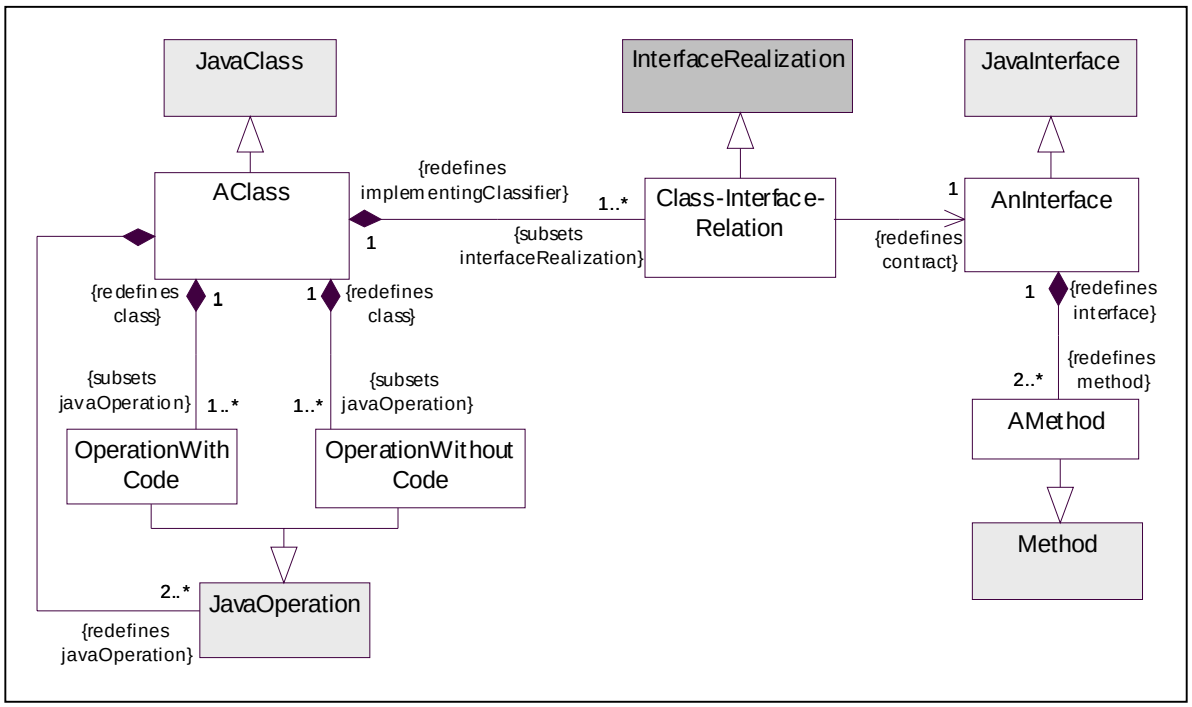


Figura 6.6 – Metamodelo origen del refactoring *Adaptar Interfaz*

El metamodelo origen 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.

Descripciones de metaclases

AClass

Representa una clase Java que implementa por lo menos una interfaz pero provee código sólo para un subconjunto de los métodos de la interfaz.

Generalizaciones

- JavaClass, del metamodelo ISM-JAVA.

Atributos

No posee atributos adicionales.

Asociaciones

- class-Interface-Relation: Class-Interface-Relation [1..*]

  Designa a un conjunto particular de relaciones de realizaciones de interfaces. Por cada relación de este tipo que la clase posee con una interfaz, la clase tendrá un subconjunto de métodos de la interfaz implementados con código nulo. Subconjunto de *BehavoiredClassifier::interfaceRealization.*

- javaOperation: JavaOperation [2..*]

  Especifica el conjunto de operaciones Java de la clase. Redefine *JavaClass::javaOperation.*

- operationWithCode: OperationWithCode [1..*]

  Especifica el conjunto de operaciones Java con código de la clase. Subconjunto de *JavaClass::javaOperation*.

- operationWithoutCode: OperationWithoutCode [1..*]

  Especifica el conjunto de operaciones Java con código nulo de la clase. Subconjunto de *JavaClass::javaOperation*.

Restricciones

[1] *AClass* es una clase concreta.

```
not self.isAbstract
```

[2] Por cada interfaz que la clase implementa a través de la relación *Class-Interface-Relation* existe un subconjunto de métodos de la interfaz que la clase implementa con código nulo.

```
self.class-Interface-Relation.anInterface -> forAll ( int|
    int.aMethod  -> exists (met|
        self.operationWithoutCode -> exists ( op|
            -- op y met tienen el mismo nombre,
            op.name = met.name and
            -- la misma visibilidad,
            op.visibility = met.visibility   and
            -- la misma cantidad de parámetros,
            op.parameter -> size() = met.parameter -> size()   and
            -- tienen los mismos tipos de parámetros
            op.parameter ->  forAll ( p |
                met.parameter ->  exists ( pp |
                    p.type = pp.type and p.direction = pp.direction  )))
```

AnInterface

Representa una interfaz Java del modelo que es implementada por una clase de tipo *AClass*.

Generalizaciones

- JavaInterface, del metamodelo ISM-JAVA.

Atributos

No posee atributos adicionales.

Asociaciones

- aMethod: AMethod [2..*]

  Referencia el conjunto de métodos de la interfaz. Redefine *JavaInterface::method*.

Restricciones

No posee restricciones adicionales.

AMethod

Representa un método de la interfaz.

Generalizaciones:

- Method, del metamodelo ISM-JAVA.

Atributos

No posee atributos adicionales.

Asociaciones

- anInterface: AnInterface [1]

  Referencia a la interfaz que posee el método. Redefine *Method::interface*.

Restricciones

No posee restricciones adicionales.

Class-Interface-Relation

Describe una relación de realización de interfaz entre una clase de tipo *AClass* y una interfaz de tipo *AnInterface*.

Generalizaciones

- InterfaceRealization, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- aClass: AClass [1]

Referencia a la clase que posee esta relación de realización. Redefine *InterfaceRealization::implementingClassifier*.

- anInterface: AnInterface [1]

  Referencia a la interfaz que especifica el contrato. Redefine *InterfaceRealization::contract*.

Restricciones

No posee restricciones adicionales.

OperationWithCode

Representa una operación Java cuya implementación contiene sentencias de código.

Generalizaciones

- JavaOperation, del metamodelo ISM-JAVA.

Atributos

No posee atributos adicionales.

Asociaciones

- aClass: AClass [1]

  Referencia a la clase Java que posee el método. Redefine *JavaOperation::class*.

Restricciones

[1] Es una operación que tiene implementación.

```
not self.body -> isEmpty()
```

[2] Es una operación que tiene una implementación con un bloque de código que posee un conjunto de sentencias.

```
self.body.block.blockStatement  -> size() > 0
```

OperationWithoutCode

Representa una operación Java cuya implementación tiene código nulo.

Generalizaciones

- JavaOperation, del metamodelo ISM-JAVA.

Atributos

No posee atributos adicionales.

Asociaciones

- aClass: AClass [1]

  Referencia a la clase Java que posee el método. Redefine *JavaOperation::class*.

Restricciones

[1] Es una operación que tiene implementación.

```
not self.body -> isEmpty()
```

[2] Es una operación que tiene una implementación con un bloque de código sin sentencias.

```
self.body.block.blockStatement  -> isEmpty()
```

6.2.1.4 Metamodelo destino

Sintaxis abstracta

El metamodelo destino es una especialización del metamodelo ISM-JAVA (figura 6.7).

El metamodelo muestra las metaclases relacionadas a los participantes esenciales en los modelos generados al aplicarse el refactoring *adaptar interfaz.*

Las diferencias entre el metamodelo origen y destino son las siguientes:

- en el metamodelo origen, *AClass* tiene una o más relaciones de realización de interfaz del tipo *Class-Interface-Relation*, donde para cada una de las interfaces que implementa existe subconjunto de métodos con código nulo;
- en el metamodelo destino, 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.

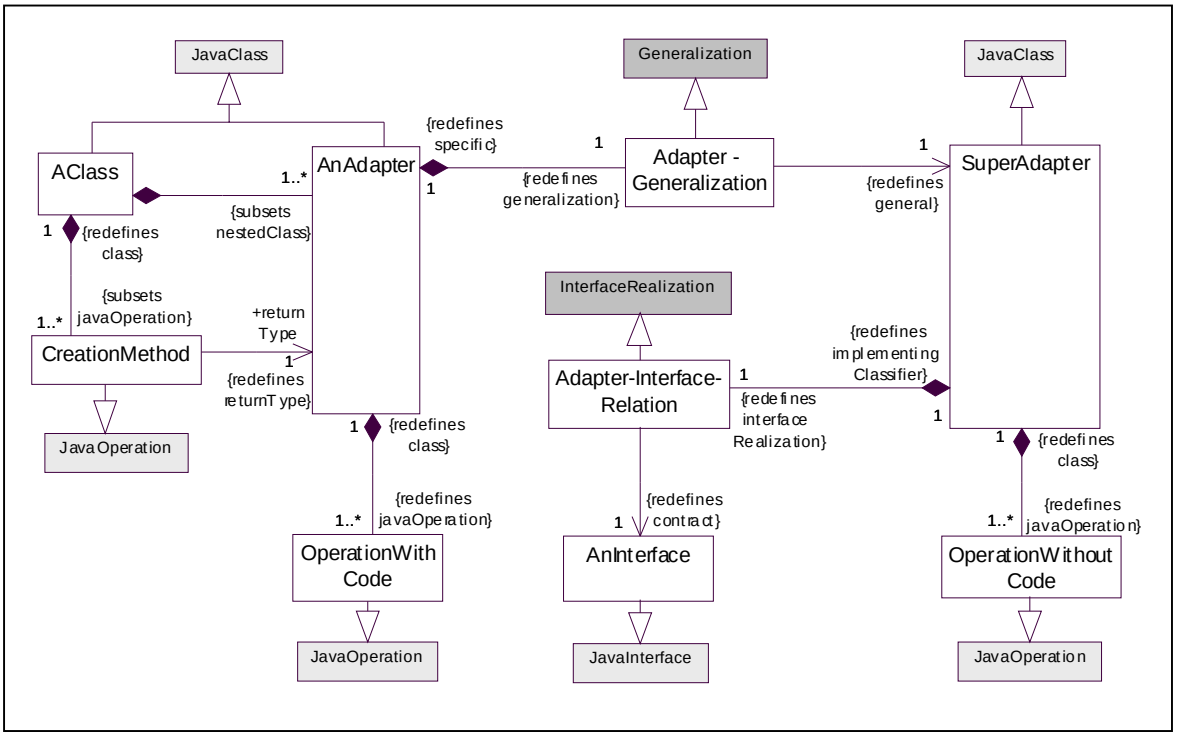


Figura 6.7 – Metamodelo destino del refactoring *Adaptar Interfaz*

Descripciones de metaclases

AClass

Representa una clase Java que implementa interfaces a través de clases *Adapter*.

Generalizaciones

- JavaClass, del metamodelo ISM-JAVA.

Atributos

No posee atributos adicionales.

Asociaciones

- anAdapter: AnAdapter [1..*]

  Designa al conjunto de clases anidadas anónimas de tipo Adapter. Subconjunto de *JavaClass::nestedClass*.

- creationMethod: CreationMethod [1..*]

  Especifica el conjunto de métodos de creación. Subconjunto de *JavaClass::javaOperation.*

Restricciones

[1] *AClass* es una clase concreta.

```
not self.isAbstract
```

[2] Por cada clase anidada anónima Adapter que implementa una interfaz, tendrá un método de creación que provee una instancia de un Adapter.

```
self.anAdapter  -> forAll ( adap|
    self.creationMethod -> exists ( met|
        met.returnType = adap  ) )
```

Adapter-Generalization

Describe una relación de generalización entre una clase *UnAdapter* y una clase *SuperAdapter*.

Generalizaciones

- Generalization, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- superAdapter: SuperAdapter [1]

  Referencia a la clase general en la relación de generalización. Redefine *Generalization::general.*.

- anInterface: AnInterface [1]

  Referencia a la clase especializada en la relación de generalización. Redefine *Generalization::specific*.

Restricciones

No posee restricciones adicionales.

AnAdapter

Representa una clase Java Adapter que extiende a una clase SuperAdapter para implementar una interfaz.

Generalizaciones

- JavaClass, del metamodelo ISM-JAVA.

Atributos

No posee atributos adicionales.

Asociaciones

- adapter-Generalization: Adapter-Generalization [1]

  Referencia a la relación de generalización. Redefine *Classifier::generalization.*

- aClass: AClass [1]

  Referencia a la clase que posee al adapter.

- operationWithCode: OperationWithCode [1..*]

  Referencia al conjunto de operaciones con código. Redefine *JavaClass::javaOperation.*

Restricciones

[1] AnAdapter es una clase anidada anónima, es decir, es una clase que no es abstracta y no es estática. Es una clase que es implícitamente final.

```
not self.isAbstract   and
not self.isStatic  and
self.isFinal
```

[2] AnAdapter por ser una clase anónima no tiene un constructor declarado explícitamente.

```
self.javaOperation -> select (oclIsTypeOf(Constructor)) -> isEmpty()
```

AnInterface

Representa una interfaz Java del modelo que es implementada por una clase Adapter.

Generalizaciones

- JavaInterface, del metamodelo ISM-JAVA.

Atributos

No posee atributos adicionales.

Asociaciones

No posee asociaciones adicionales.

Restricciones

[1] Una interfaz de tipo AnInterface posee dos o más métodos.

```
self.method -> size () >= 2
```

Adapter-Interface-Relation

Describe una relación de realización de interfaz entre una clase de tipo *SuperAdapter* y una interfaz de tipo *AnInterface*.

Generalizaciones

- InterfaceRealization, del paquete *Kernel* de UML.

Atributos

No posee atributos adicionales.

Asociaciones

- superAdapter: SuperAdapter [1]

  Referencia a la clase que posee esta relación de realización. Redefine *InterfaceRealization::implementingClassifier.*.

- anInterface: AnInterface [1]

  Referencia a la interfaz que especifica el contrato. Redefine *InterfaceRealization::contract*.

Restricciones

No posee restricciones adicionales.

CreationMethod

Representa una operación Java que es un método de creación.

Generalizaciones

- JavaOperation, del metamodelo ISM-JAVA.

Atributos

No posee atributos adicionales.

Asociaciones

- aClass: AClass [1]

  Referencia a la clase que posee la operación. Redefine *JavaOperation::class.*.

- returnType: AnAdapter [1]

  Especifica el tipo de retorno de la operación. Redefine *JavaOperation::returnType*.

Restricciones

[1] Es un método privado.

```
self.visibility = #private
```

[2] El método retorna una referencia a una instancia de *AnAdapter*, por lo tanto el cuerpo del método tendrá una sentencia *return* con una expresión de creación de instancia de clase anónima.

```
self.body.block.oclIsTypeOf(Return)  and
self.body.block.return.expression.oclIsTypeOf(ClassInstanceCreationExpression)
```

OperationWithCode

Representa una operación Java cuya implementación contiene sentencias de código.

Generalizaciones

- JavaOperation, del metamodelo ISM-JAVA.

Atributos

No posee atributos adicionales.

Asociaciones

- anAdapter: AnAdapter [1]

  Referencia a la clase que posee el método. Redefine *JavaOperation::class*.

Restricciones

[1] Es una operación que tiene implementación.

```
not self.body -> isEmpty()
```

[2] Es una operación que tiene una implementación con un bloque de código que posee un conjunto de sentencias.

```
self.body.block.blockStatement  -> size() > 0
```

OperationWithoutCode

Representa una operación Java cuya implementación tiene código nulo.

Generalizaciones

- JavaOperation, del metamodelo ISM-JAVA.

Atributos

No posee atributos adicionales.

Asociaciones

- superAdapter: SuperAdapter [1]

  Referencia a la clase que posee el método. Redefine *JavaOperation::class*.

Restricciones

[1] Es una operación que tiene implementación.

```
not self.body -> isEmpty()
```

[2] Es una operación que tiene una implementación con un bloque de código sin sentencias.

```
self.body.block.blockStatement  -> isEmpty()
```

SuperAdapter

Representa una clase Java que implementa una interfaz.

Generalizaciones

- JavaClass, del metamodelo ISM-JAVA.

Atributos

No posee atributos adicionales.

Asociaciones

- adapter-Interface-Relation: Adapter-Interface-Relation [1]

  Referencia la relación de realización de interfaz. Redefine *BehavoiredClassifier::interfaceRealization.*

- operationWithoutCode: OperationWithoutCode [1..*]

  Referencia al conjunto de operaciones con código nulo. Redefine *JavaClass::javaOperation*.

Restricciones

No posee restricciones adicionales.

6.2.1.5 Regla de transformación

La regla *Adaptar Interfaz* se aplica sobre aquellas clases de un modelo que implementan una interfaz pero sólo proveen código para algunos de los métodos de la interfaz. Este refactoring mueve los métodos implementados a un *Adapter* de la interfaz y la hace accesible a través de un método de creación (Creation Method).

Los pasos para aplicar esta regla son los siguientes:

1. Usar un *Adapter* que implemente la interfaz y provea el comportamiento nulo (métodos implementados con código nulo). Crear en la clase un método de creación (*Creation Method*) que retorne una referencia a una instancia que extiende al *Adapter*.
2. Borrar de la clase los métodos con comportamiento nulo.
3. Mover los métodos implementados (especificados en la interfaz) a la nueva instancia del *Adapter*.
4. Eliminar la relación de implementación de interfaz de la clase.
5. Proveer la instancia del *Adapter* a los clientes que lo necesiten.

A continuación, se especifican parcialmente estos cambios a través de la regla de transformación.

Transformation Adaptar Interfaz **{**

parameter
source: Metamodelo Origen Adaptar Interfaz:: JavaPackage
target: Metamodelo Destino Adaptar Interfaz:: JavaPackage

local operations

Metamodelo Origen Adaptar Interfaz:: JavaClass::
opWithCodeThatCorrespondWith
(Metamodelo Origen Adaptar Interfaz:: int: Interfaz):
Set (JavaOperation);

```
-- Retorna el conjunto de operaciones con código de la clase a la cual se aplica
-- la operación que se corresponde con métodos de la interfaz int.
opWithCodeThatCorrespondWith (int) =
-- Por cada operación con código de la clase, se selecciona la operación op tal que
self.operationWithCode -> select (op|
        -- existe en el conjunto de métodos de la interfaz int,
        int.aMethod ->
            exists ( met|
                -- un método met que se corresponde con la operación op de la clase,
                -- op y met tienen el mismo nombre,
                op.name = met.name and
                -- la misma visibilidad,
                op.visibility = met.visibility   and
                -- la misma cantidad de parámetros,
                op.parameter -> size()=met.parameter -> size()   and
                -- tienen los mismos tipos de parámetros
                op.parameter ->  forAll ( p |
                    met.parameter ->  exists ( pp |
                    p.type = pp.type and p.direction = pp.direction  ) ) )
```

Metamodelo Origen Adaptar Interfaz:: JavaClass::
opWithoutCodeThatCorrespondWith
(Metamodelo Origen Adaptar Interfaz:: int: Interfaz):
Set (JavaOperation);

```
-- Retorna el conjunto de operaciones con código nulo de la clase a la cual se
-- aplica la operación que se corresponde con métodos de la interfaz int.
opWithoutCodeThatCorrespondWith (int) =
-- Por cada operación sin código de la clase, se selecciona la operación op tal que
     self.operationWithoutCode -> select (op|
        -- existe en el conjunto de métodos de la interfaz int,
        int.aMethod ->
            exists ( met|
                -- un método met que se corresponde con la operación op de la clase,
                -- op y met tienen el mismo nombre,
                op.name = met.name and
```

```
        -- la misma visibilidad,
        op.visibility = met.visibility   and
        -- la misma cantidad de parámetros,
        op.parameter -> size()=met.parameter -> size()   and
        -- tienen los mismos tipos de parámetros
        op.parameter ->  forAll ( p |
            met.parameter ->  exists ( pp |
            p.type = pp.type and p.direction = pp.direction  ) ) )
```

postconditions
post

```
    -- Para toda clase sourceClass, de tipo AClass, en el paquete source,
    source.ownedMember ->  select(oclIsTypeOf(AClass)) -> forAll ( sourceClass |

        -- existe una clase targetClass, en el paquete target tal que,
        target.ownedMember ->  select(oclIsTypeOf(AClass)) -> exists ( targetClass |

            -- atributos y asociaciones heredados de JavaClass
            -- ambas clases pertenecen al mismo package
            targetClass.oclAsType(AClass).JavaPackage =
                sourceClass.oclAsType(AClass).JavaPackage  and

            -- targetClass tiene la misma superclase que sourceClass,
            targetClass.oclAsType(AClass).superClass  =
                sourceClass.oclAsType(AClass).superClass  and

            -- el conjunto de clases anidadas de sourceClass está incluido en el
            -- conjunto de clases anidadas de targetClass siendo este último conjunto
            -- mayor ya que contiene las clases anónimas de tipo AnAdapter,
            targetClass.oclAsType(AClass).nestedClass -> size() >
                sourceClass.oclAsType(AClass).nestedClass -> size()    and
            targetClass.oclAsType(AClass).nestedClass -> includesAll
                (sourceClass.oclAsType(AClass).nestedClass) and

            -- targetClass  tiene los mismos valores que sourceClass para los
            -- siguientes atributos:
            targetClass.oclAsType(AClass).isFinal =
                sourceClass.oclAsType(AClass).isFnal    and
            targetClass.oclAsType(AClass).isStatic =
                sourceClass.oclAsType(AClass).isStatic    and
            targetClass.oclAsType(AClass).field =
                sourceClass.oclAsType(AClass).field     and
            targetClass.oclAsType(AClass).parameter =
                sourceClass.oclAsType(AClass).parameter    and

            -- operaciones propias de targetClass:
            -- targetClass  posee el conjunto de operaciones de sourceClass del cual
            -- se excluyen las operaciones con código y sin código que corresponden
```

```
-- a las relaciones de implementación de interfaz que posee sourceClass,
targetClass.oclAsType(AClass).javaOperation -> includesAll (
    sourceClass.oclAsType(AClass).javaOperation ->
        excludesAll(sourceClass.oclAsType(AClass).operationWithoutCode) ->
        excludesAll(sourceClass.oclAsType(AClass).operationWithCode)  )  and
-- targetClass posee además un conjunto de métodos de creación: un
-- método de creación por cada relación Class-Interface-Relation que
-- posee sourceClass
targetClass.oclAsType(AClass).creationMethod -> size() =
    sourceClass.oclAsType(AClass).class-Interface-Relation -> size()   and
-- si en los cuerpos de las operaciones de sourceClass una sentencia de
-- llamada a un método tiene como argumento una expresión de tipo
-- AnInterface,
sourceClass.oclAsType(AClass).javaOperation.body.block.
    blockStatement.subBlock -> exists (statement|
        statement.oclIsTypeOf(MethodInvocation) and
        statement.oclAsType(MethodInvocation).argument -> exists(arg|
                arg.expression.type = AnInterface)     implies
-- en targetClass será del tipo AnAdapter.
targetClass.oclAsType(AClass).javaOperation.body.block.
    blockStatement.subBlock -> exists (statement|
        statement.oclIsTypeOf(MethodInvocation) and
        statement.oclAsType(MethodInvocation).argument -> exists(arg|
                arg.expression.type = AnAdapter )

-- el conjunto de atributos propios de targetClass es igual al conjunto de
-- atributos propios de sourceClass,
targetClass.oclAsType(AClass).field =
    sourceClass.oclAsType(AClass).field   and

-- por cada relación de realización de interfaz del tipo Class-Interface-
-- Relation que posee sourceClass, targetClass tendrá una clase anidada
-- anónima de tipo AnAdapter que implementará cada interfaz.
sourceClass.oclAsType(AClass).class-Interface-Relation -> size()  =
        targetClass.oclAsType(AClass).anAdapter -> size()

-- Por cada interfaz que implementa sourceClass
sourceClass.oclAsType(AClass).class-Interface-Relation.anInterface  ->
     forAll ( int|

    --existirá en targetClass una clase anidada anónima de tipo AnAdapter
    -- tal que,
    targetClass.oclAsType(AClass).anAdapter   -> exists ( adap|
        -- adap tendrá un conjunto de operaciones con código:
        -- aquellas operaciones con código de sourceClass que se
        -- corresponden con  métodos de la interfaz int,
        adap.operationWithCode  -> includesAll
            (sourceClass.oclAsType(AClass).
                opWithCodeThatCorrespondWith(int) ) and
```

```
                -- adapt extiende a SuperAdapter e implementa dicha interfaz,
                adap.adapterGeneralization.superAdapter.
                    adapter-Interface-Relation.anInterface  =  int   and

                -- SuperAdapter tendrá las operaciones con código nulo que se
                -- corresponden con  métodos de la interfaz int,
                adapt.adapter-Generalization.superAdapter.operationWithoutCode
                    -> includesAll (sourceClass.oclAsType(AClass).
                        opWithoutCodeThatCorrespondWith(int))  ) ) and

            -- atributos heredados de NamedElement
            targetClass.name = sourceClass.name and
            targetClass.visibility = sourceClass.visibility and

            -- attributo heredado de Classifier
            targetClass.oclAsType(AClass).isAbstract =
                sourceClass.oclAsType(AClass).isAbstract and

            -- asociaciones heredadas de Namespace
            -- ambas clases tienen las mismas restricciones
            targetClass.oclAsType(AClass).ownedRule =
                sourceClass.oclAsType(AClass).ownedRule

            -- ambas clases tienen el mismo conjunto de elementos importados
            targetClass.oclAsType(AClass).elementImport =
                sourceClass.oclAsType(AClass).elementImport     ))
```

post

```
-- Para toda clase Java en el paquete source,
source.ownedMember ->  select(oclIsTypeOf(JavaClass)) -> forAll ( sourceClass |

    -- existe una clase en el paquete target tal que,
    target.ownedMember ->  select(oclIsTypeOf(JavaClass)) -> exists ( targetClass |

        -- sourceClass y targetClass  tienen las mismas características.
        targetClass = sourceClass ))
```

post

```
-- Para toda interfaz Java en el paquete source,
source.ownedMember ->  select(oclIsKindOf(JavaInterface)) -> forAll ( sInterface |

    -- existe una interfaz en el paquete target tal que,
    target.ownedMember ->  select(oclIsKindOf(JavaInterface)) ->exists ( tInterface |

        -- sInterface y tInterface tienen las mismas características.
        tInterface = sInterface  ))
```

6.2.1.6 Ejemplo

La figura 6.8.a muestra un modelo que consta de una clase llamada *CardComponent* que extiende a la clase *Component* e implementa la interfaz *MouseMotionListener*. El código de las operaciones de *CardComponent* revela que esta clase sólo implementa uno de los dos métodos declarados por la interfaz, el método *mouseMoved* tiene código nulo.

Este modelo es una instancia del metamodelo origen del refactoring adaptar interfaz (figura 6.6). La clase *CardComponent* del ejemplo de la figura 6.8.a es una instancia de la metaclase *AClass* del metamodelo origen. Esta clase tiene una relación de realización de interfaz, instancia de la metaclase *Class-Interface-Relation*, con la clase *MouseMotionListener* la cual es una instancia de *AnInterface*. La clase *CardComponent* tiene métodos con implementación y con implementación nula, que corresponden a métodos declarados en la interfaz que implementa, los cuales son instancias de las metaclases *OperationWithCode* y *OperationWithoutCode* respectivamente.

La aplicación de la regla conduce a reemplazar la interfaz implementada parcialmente con un Adapter y los pasos son los siguientes:

- crear un método en la clase *CardComponent* de creación de instancia de un adapter, instancia que extiende al adapter *MouseMotionAdapter* que JDK provee para la interfaz *MouseMotionListener*, utilizando la facilidad de Java de clases anidadas anónimas;
- eliminar el método *mouseMoved()* con código nulo de la clase *CardComponent*;
- mover el método *mouseDragged()* de la clase *CardComponent* a la instancia de *MouseMotionAdapter*;
- eliminar la relación de realización de interfaz que la clase *CardComponent* tiene con *MouseMotionListener*;
- proveer a los clientes la nueva instancia de adapter, en el constructor de la clase *CardComponent* se reemplaza el argumento por el nuevo método de creación de instancia del adapter.

El modelo resultante se muestra en la figura 6.8.b. La clase *MouseMotionAdapter*, instancia de la metaclase *SuperAdapter* del metamodelo destino (ver figura 6.7), posee los métodos con código nulo. El método *createMouseMotionAdapter()* de la clase *CardComponent* es una instancia de la metaclase *CreationMethod*. El método devuelve un objeto, que extiende a *MouseMotionAdapter*, a través de la definición de una clase anidada anónima en el cuerpo de dicho método. Esta clase anidada anónima, instancia de la metaclase *AnAdapter* del metamodelo destino, contiene el método implementado *mouseDragged()*.

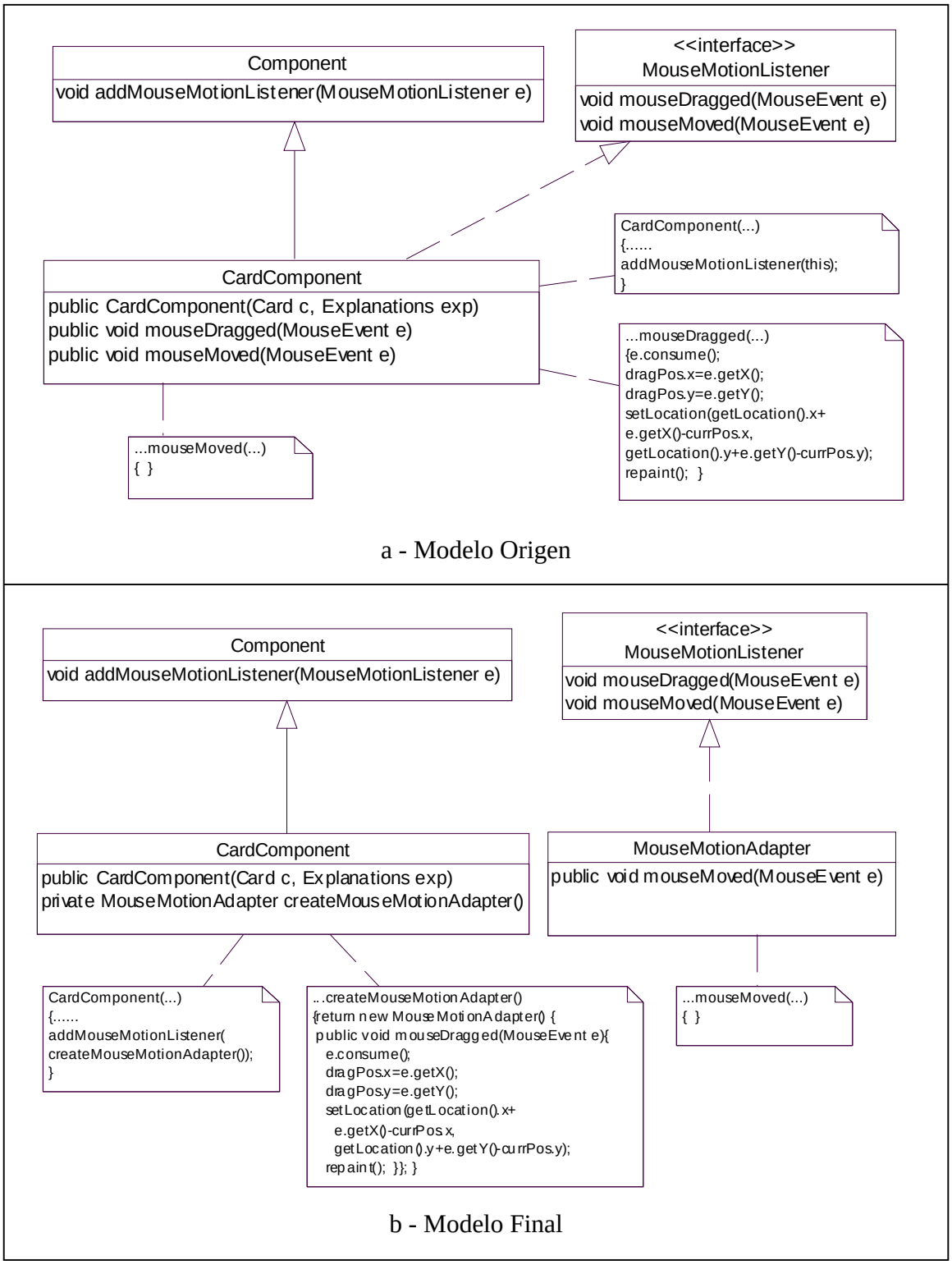


Figura 6.8 – Una instancia del refactoring *Adaptar Interfaz*

CAPÍTULO 7

FORMALIZACIÓN DE REFACTORINGS

Los metamodelos en el contexto de MDA son expresados usando MOF. Este define una manera de capturar los distintos estándares de modelado e intercambiar construcciones que son usadas en MDA. Su objetivo es definir lenguajes de la misma manera y luego integrarlos semánticamente. MOF y el núcleo de UML están alineados en sus conceptos de modelado.

Los metamodelos MOF son expresados como una combinación de diagramas de clases UML y restricciones OCL. UML y OCL son imprecisos y ambiguos cuando se trata de simular, verificar y validar propiedades de un sistema y también cuando se pretende generar modelos o implementaciones a través de transformaciones. OCL es un lenguaje textual, sin embargo, las expresiones OCL dependen de los diagramas de clases, es decir, el contexto sintáctico se determina gráficamente. La especificación de UML está definida usando la técnica de metamodelado pero *es importante notar que la especificación de UML como un metamodelo no imposibilita que sea especificado posteriormente mediante un lenguaje formal* (UML-Infrastructure, 2007; página 11).

Técnicas formales y semiformales tienen roles complementarios en los procesos de desarrollo de software basados en MDA. Las dos técnicas aportan sus beneficios. Por un lado, las técnicas semiformales carecen de semántica precisa, sin embargo, tienen la ventaja de visualizar las construcciones del lenguaje, permitiendo gran productividad en los procesos de especificación especialmente cuando está soportado por herramientas. Por otro lado, las especificaciones formales permiten producir especificaciones de software precisas y analizables y automatizar las transformaciones de modelo a modelo, sin embargo, requieren cierta familiaridad con la notación formal que la mayoría de los diseñadores no poseen y el aprendizaje de estas técnicas requiere un tiempo considerable. La integración de OCL con lenguajes algebraicos permite una definición de tipos e interrelaciones en términos de álgebras, aprovechando todos los avances en el área.

Una técnica de especificación formal debe proveer al menos, una sintaxis, una semántica y un sistema de inferencia. El sistema de inferencia permite definir deducciones que pueden ser elaboradas a partir de una especificación formal. Estas deducciones permiten derivar y chequear nuevas fórmulas. De esta manera, el sistema de inferencia puede ayudar a automatizar los procesos de testeo, prototipado o verificación.

Por lo tanto, se propone en esta tesis formalizar los metamodelos y refactorings usando el lenguaje de metamodelado algebraico NEREUS. El análisis y elección de este lenguaje para la formalización de artefactos de software corresponde a etapas previas del proyecto en el cual esta tesis está enmarcada. NEREUS es un lenguaje alineado con MOF, es decir, la mayoría de los conceptos de los metamodelos MOF pueden ser traducidos a NEREUS. Además, este lenguaje puede ser visto como una notación intermedia abierta a otras especificaciones formales. Se definió, en particular, la traducción de construcciones

NEREUS a CASL (Favre, 2006) lo que permitiría razonar sobre la consistencia de los refactorings utilizando las herramientas que provee este último, tales como prototipo de consistencia de especificaciones algebraicas, asistente para pruebas de teoremas *Isabelle* (COFI, 2008).

En (Favre, Martinez y Pereira, 2001, 2002, 2003, 2005) se describe la integración de técnicas formales y semiformales en procesos de ingeniería forward. En particular, se presenta la integración de modelos estáticos UML con lenguajes algebraicos y el lenguaje Eiffel. Lo expuesto en este capítulo fue presentado en (Favre y Pereira, 2008).

En este capítulo se detalla, en el apartado 7.1, la formalización en NEREUS de los metamodelos MOF y en 7.2 la formalización de los contratos de refactorings.

7.1 Traducción de metamodelos MOF a NEREUS

Para la formalización de metamodelos MOF en el lenguaje de especificación algebraica NEREUS se utilizaron los resultados de investigaciones previas (Favre, 2006).

La traducción de los metamodelos MOF a especificaciones NEREUS se construye automática y gradualmente. Los metamodelos MOF y NEREUS tienen construcciones y mecanismos de estructuración similares, por lo tanto toda la información contenida en los metamodelos MOF se refleja en la correspondiente especificación en NEREUS.

Cada paquete del metamodelo MOF es traducido a un paquete en NEREUS. Cada elemento del paquete del metamodelo se traduce de la siguiente manera:

- cada clase del metamodelo MOF es traducida a una clase NEREUS instanciando el componente reusable que provee el lenguaje NEREUS para tal efecto, de esta manera, la signatura y algunos axiomas de la clase se obtienen por instanciación del esquema reusable BOX;
- cada asociación del metamodelo es traducida a una asociación en NEREUS instanciando el esquema reusable ASSOCIATION;
- las especificaciones OCL de los metamodelos se traducen a través de un sistema de reglas de transformación teniendo en cuenta que en los metamodelos pueden encontrarse expresiones OCL como precondiciones y postcondiciones de las operaciones, invariantes de clase, restricciones a atributos y a asociaciones;
- los tipos OCL también tienen su correspondiente traducción a NEREUS.

La figura 7.1 muestra las etapas del proceso de traducción de metamodelos MOF a NEREUS.

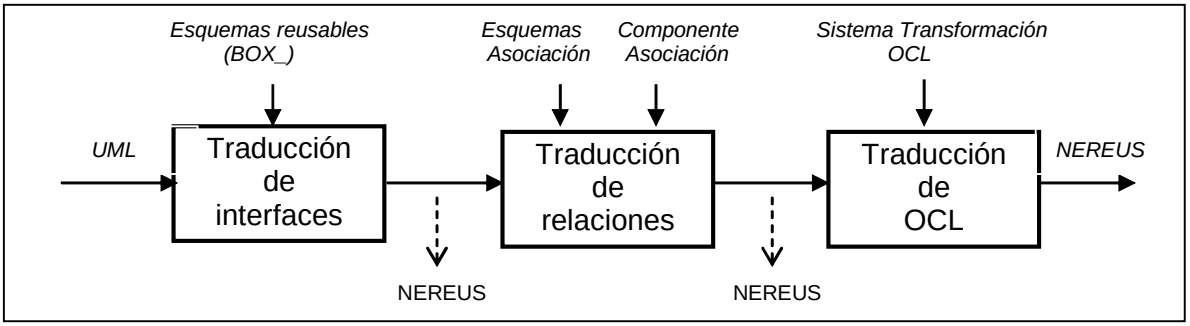


Figura 7.1 - Desde UML/OCL a NEREUS

7.1.1 Traducción de clases

Las clases de un metamodelo MOF se traducen a NEREUS por instanciación del esquema BOX_ que se muestra en la figura 7.2.

Las relaciones de cliente de la clase se expresan por medio de la cláusula IMPORTS y las relaciones de generalización/especialización por medio de la cláusula INHERITS o SUBTYPE. A través de la cláusula ASSOCIATES se especifican las relaciones que posee la clase y son definidas por instanciación de esquemas existentes en la jerarquía *Association* que provee el lenguaje NEREUS.

La instanciación de *Box_* define una operación constructora *create*, cuya funcionalidad se obtiene traduciendo cada tipo de atributo. Para cada tipo de argumento del constructor se define las operaciones *get-_* y *set-__*.

Para cada operación en la clase se genera la signatura en la clase NEREUS y se introduce un axioma sobre *create*. Esto se realiza instanciando TP1, TP2,... con los tipos de los argumentos de las operaciones. Esta información es extraída de la especificación de la clase en el diagrama.

Las precondiciones, postcondiciones e invariantes de clase que forman parte del metamodelo MOF se traducen a precondiciones y axiomas NEREUS como se detalla en el apartado 7.1.3.

CLASS *Box_*
IMPORTS *TP1,.., TPm, T-attr$_1$, T-attr$_{2,...,}$ T-att*
INHERITS *B1,B2,...,Bm*
IS-SUBTYPE-OF *C1, C2,...,Cr*
ASSOCIATES
<< Aggregation-E$_1$>>,...,<<Composition_C$_1$>>,...,<<Association-D$_1$>>
EFFECTIVE
TYPES *Box_*
FUNCTIONS
create: *T-attr$_1$ x ... x T-attr$_n$ → Box_*
set-i: *Box_ x T-attr$_i$ → Box_*
get-i: *Box_ → T-attr$_i$* *1 ≤ i ≤ n*
DEFERRED
FUNCTIONS
meth$_1$: *Box_ x TPi$_1$ x TPi$_2$ x .....TPi$_n$ → TPi$_j$*
...
meth$_r$: Box_ x TPr$_1$ xTPr$_2$.....x TPr$_p$ → TPr$_k$
AXIOMS cp: Box_; t1:*T-ATTR1*; ti, ti´:*T-ATTR-i*,...,tn:*T-ATTR-n*
get-i (create(t1,t2,...,tn)) = ti
set-i(create(t1,t2,....tn),ti´) = create(t1,t2,..,ti´,..tn) 1 ≤ i ≤ n
END-CLASS

Figura 7.2 – El esquema reusable BOX

7.1.2 Traducción de asociaciones

Las asociaciones de un metamodelo MOF se traducen a NEREUS por instanciación del esquema ASSOCIATION_ que se muestra en la figura 7.3.

La traducción comienza con la palabra reservada ASSOCIATION seguido del cual se instanciará el esquema con el nombre de la asociación. La cláusula IS vincula la asociación a un constructor de tipo, perteneciente a un repertorio de constructores de tipo del esquema ASSOCIATION que provee el lenguaje, y será instanciado a partir de las clases intervinientes en la asociación, sus multiplicidades, roles y visibilidad.

La cláusula CONSTRAINED_BY permite especificar restricciones de la asociación.

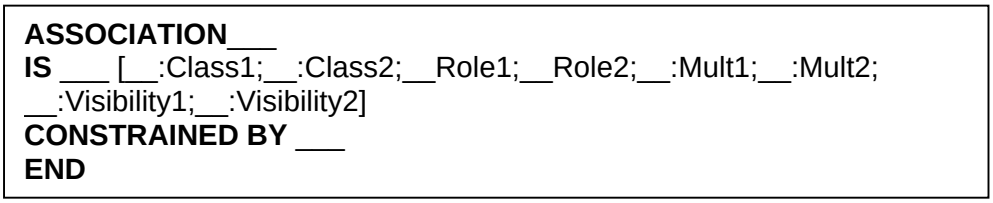
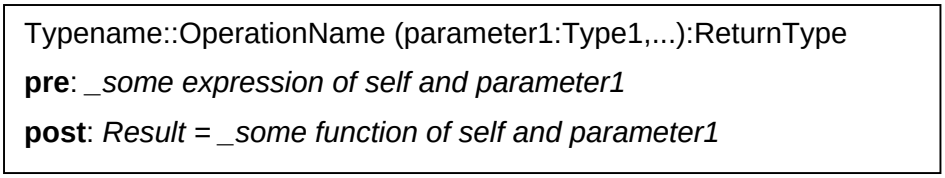


Figura 7.3 – El esquema reusable ASSOCIATION

7.1.3 Traducción de especificaciones OCL

En los metamodelos pueden encontrarse especificaciones OCL como precondiciones, postcondiciones o invariantes de clases, restricciones a atributos y restricciones a asociaciones. Estas especificaciones OCL serán analizadas para derivar axiomas que serán incluidos en las especificaciones NEREUS.

Una operación se especifica en OCL por medio de precondiciones y postcondiciones:



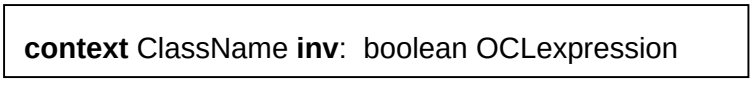
En la expresión, *self* puede ser usado para hacer referencia al objeto al cual se aplica la operación y el nombre *Result* es el nombre del objeto retornado si existe alguno. Los nombres de los parámetros pueden ser usados en las expresiones. En una postcondición las expresiones OCL pueden referirse a dos conjuntos de valores para cada propiedad del objeto:

- el valor de una propiedad al comienzo de una operación y
- el valor de una propiedad al finalizar la operación.

El valor de una propiedad en una postcondición es el valor después de finalizada la operación. Para hacer referencia al valor de una propiedad al comienzo de la operación se debe agregar el postfijo @ seguido de la palabra reservada *pre* al nombre de la propiedad.

Las precondiciones OCL se traducen en precondiciones NEREUS, mientras que las postcondiciones permiten generar axiomas en NEREUS.

Los invariantes se denotan en el contexto de una clase siguiendo al rótulo **inv:** permiten generar axiomas de la clase ClassName en la traducción a NEREUS.



Los tipos básicos OCL *Integer, Real, String, Char, Boolean y Enumeration* se corresponden con tipos primitivos NEREUS del mismo nombre. El tipo *OCLAny* con el *sort ANY* de NEREUS. Las especificaciones OCL de *Collection, Set, Bag* y *Sequence* se corresponden con las clases NEREUS del mismo nombre.

El proceso de traducción de especificaciones OCL a NEREUS está soportado por un sistema de reglas de transformación, resultado de investigaciones previas realizadas dentro del proyecto en el cual esta tesis está enmarcada (Favre, 2007). Algunas de ellas son mostradas en la figura 7.4.

REGLA	OCL	NEREUS
1	v. operation (v') (v es una variable)	operation (v, v')
2	v->operation (v') (v es una variable)	operation (v, v')
3	v.attribute	attribute (v)
4	**context** A object.rolename (A es una asociación)	get_ rolename (a, object) *Sea a:A*
5	OCLexp1 opBin OCLexp2 opBin::= and| or| xor| =| <>| <| <=| >| >=| +| -| / | *	Translate $_{NEREUS}$ (OCLexp1) opBin Translate $_{NEREUS}$ (OCLexp2) o Translate $_{NEREUS}$ (opBin) (Translate $_{NEREUS}$ (OCLexp1) , Translate $_{NEREUS}$ (OCLexp2))
6	e.op (e es una expresión OCL)	op (Translate $_{NEREUS}$ (e))
7	collection-> collect (v: Elem | expr-with-v) expr-with-v : S	**LET** **FUNCTIONS** f: Elem -> S **AXIOMS v : Elem** f (v)= *Translate $_{NEREUS}$* (expr-with-v) **IN** collect (collection, f) **END-LET** --- *Notación concisa equivalente:* collect $_{v}$ (collection, [Translate $_{NEREUS}$ (expr-with-v)])
8	collection->op (v: Elem | bool-expr-with-v) op ::=select| forAll| reject| exists	**LET** **FUNCTIONS** f: Elem → Boolean **AXIOMS v : Elem** f (v)=Translate $_{NEREUS}$ (bool-expr-with-v) **IN** op (collection, f) **END-LET** --- *Notación concisa equivalente:* op $_{v}$ (collection, [Translate $_{NEREUS}$ (bool-expr-with-v)])

Figura 7.4 – Traducción OCL a NEREUS: reglas de transformación

En (Favre, Martinez y Pereira, 2000) se presenta un proceso riguroso de ingeniería forward de modelos estáticos UML, en particular, se describe la traducción de especificaciones OCL al nivel algebraico.

7.1.4. Traducción del metamodelo origen *Extraer Composite*

La figura 7.5 muestra la traducción a NEREUS de la clase *Component*, del metamodelo origen *Extraer Composite* (figura 4.2), obtenida por instanciación del esquema reusable BOX (figura 7.2) y la aplicación de las reglas de transformación detalladas en la figura 7.4.

```
CLASS Component
   IS-SUBTYPE-OF M_Class
   ASSOCIATES
      << Component-ComponentCompositeGeneralization>>,
      << Component-ComponentLeafGeneralization >>,
      << AssEndComponent-Component >>
   TYPES Component
   FUNCTIONS
   create: * → Component
   AXIOMS     c: Component;
      Ap: AssEndComponent-Component;
      Aa:AssEndComponent-CompositeComponentAssoc;
      Cp: Component-ComponentCompositeGeneralization;
      Oc: Class-Operation;
      Cc: ComponentCompositeGeneralization-Composite

      forAll a1,a2 ( collect assEnd (get_associationEnd(Ap,c),
        [get_association(Aa, assEnd)]),  [a1=a2 or isEquivalentTo(a1,a2)] )

      forAll cl (  collect child    (get_compositeSpecialization (Cp, c),
         [get_child(Cc,child)]),
            [exists op (get_ownedOperation(Oc, cl),
               [forAll c (excluding (collect child (get_compositeSpecialization( Cp, c),
                  [get_child(Cc,child)] ), cl),
                     [exists o (get_ownedOperation(Oc, c)),
                           [isEquivalentTo (op, o)]  ]]
END-CLASS
```

Figura 7.5 – Formalización de la clase *Component* en NEREUS

Para construir la formalización de la clase *Component* se produjeron las siguientes instanciaciones sobre el esquema BOX:

- seguido a la palabra clave CLASS se instancia con el nombre de la clase;
- seguido a la palabra clave IS-SUBTYPE-OF se instancia con la relación de subtipo de la clase, reflejando que *Component* es subtipo de Class (del metamodelo UML);
- a continuación de ASSOCIATES se instancian con las asociaciones que la clase posee con las clases ComponentCompositeGeneralization, ComponentLeafGeneralization y AssEndComponent;

- a continuación de AXIOMS se traducen las restricciones que esta clase posee en el metamodelo teniendo en cuenta las reglas de traducción de OCL a NEREUS, por ejemplo, la siguiente restricción especificada en OCL:

Las asociaciones entre *Composite* y *Component* son equivalentes.

```
self.associationEnd -> collect ( assEnd |  assEnd.association)
        -> forAll ( a1, a2 |
                a1 = a2  or
                a1.isEquivalentTo(a2)  )
```

se traduce a NEREUS de la siguiente manera:

1. self.associationEnd

   regla 4, OCL: context A object.rolename donde,
   A: AssEndComponent-Component
   object: self
   rolename: associationEnd

   regla 4, NEREUS: get_ rolename (a, object) donde a: A

   get_associationEnd (Ap, c) donde
   Ap: AssEndComponent-Component
   c: Component

2. self.associationEnd → collect (assEnd | assEnd.association)
   es decir, del resultado anterior que son todos los extremos de asociación del objeto se obtienen las asociaciones a las que pertenecen

   regla 7, OCL: collection -> collect (v: Elem | expr-with-v) donde,

   collection: self.associationEnd (la cual se tradujo en el paso previo)

   get_associationEnd (AP, c) donde
   Ap: AssEndComponent-Component
   c: Component

   expr-with-v: assEnd.association

   regla 4, OCL: context A object.rolename donde,
   A: AssEndComponent-CompositeComponentAssoc
   object: assEnd
   rolename: association

   regla 4, NEREUS: get_ rolename (a, object) donde a: A

   get_association (Aa, assEnd) donde
   Aa: AssEndComponent-CompositeComponentAssoc
   assEnd: AssEndComponent

regla 7, NEREUS: collect $_{v}$ (collection, [Translate $_{NEREUS}$ (expr-with-v)])
donde v es cada uno de los elementos de la colección renombrado por assEnd

collect $_{assEnd}$ (get_associationEnd(Ap,c),
[get_association(Aa, assEnd)])

3. self.associationEnd → collect (assEnd | assEnd.association)
   → forAll (a1, a2 |
   a1 = a2 or
   a1.isEquivalentTo(a2))

a la colección de asociaciones obtenidas en el paso 2 se le aplica un producto cartesiano entre sus elementos para ver si dos asociaciones son iguales o equivalentes.

regla 8, OCL: collection -> op (v: Elem | bool-expr-with-v) donde,

collection: self.associationEnd → collect (assEnd | assEnd.association)
(la cual se tradujo en el paso previo)

collect $_{assEnd}$ (get_associationEnd(Ap,c),
[get_association(Aa, assEnd)])

op: forAll

bool-expr-with-v : a1 = a2 or a1.isEquivalentTo(a2)

regla 5, OCL: OCLexp1 opBin OCLexp2 donde,
OCLexp1: a1 = a2
opBin: or
OCLexp2: a1.isEquivalentTo(a2)

regla 5, NEREUS: Translate $_{NEREUS}$ (OCLexp1) opBin Translate $_{NEREUS}$ (OCLexp2)

Translate $_{NEREUS}$ (OCLexp1) = Translate $_{NEREUS}$ (a1 = a2)

regla 5, OCL: OCLexp1 opBin OCLexp2
OCLexp1= a1
opBin = =
OCLexp2= a2

regla 5, NEREUS: Translate $_{NEREUS}$ (OCLexp1) opBin Translate $_{NEREUS}$ (OCLexp2)

a1 = a2

De maner similar:
Translate $_{NEREUS}$ (OCLexp2) = Translate $_{NEREUS}$ (a1.isEquivalentTo(a2))

regla 1, OCL: v. operation (v')

v = a1
operation = isEquivalentTo
v' = a2

regla 1, NEREUS: operation (v , v')

isEquivalentTo (a1, a2)

por lo tanto se traduce a:

a1 = a2 or
isEquivalentTo(a1,a2)

regla 8, NEREUS: op_v (collection, [Translate $_{NEREUS}$ (bool-expr-with-v)])
donde v es un par de elementos de la colección renombrado por a1 y a2

forAll $_{a1,a2}$ (collect $_{assEnd}$ (get_associationEnd(Ap,c),
[get_association(Aa, assEnd)]), [a1=a2 or isEquivalentTo(a1,a2)])

La figura 7.6 muestra la traducción a NEREUS de la asociación *ComponentCompositeGeneralization-Composite* obtenida por instanciación del esquema reusable ASSOCIATION (figura 7.3). El esquema se instancia con el nombre de la asociación y a través de la cláusula IS se vincula al constructor de tipo *Composition-1* (ver figura 2.7). Este es un esquema de una composición con multiplicidad 1 a 1 que será instanciado con las clases que intervienen en la asociación, sus roles, multiplicidad y visibilidad de los extremos de asociación.

En las restricciones de la asociación se refleja que el rol de la clase *Composite* para esta asociación es redefinido por *child* con respecto al rol heredado *specific* y que el extremo *componentGeneralization* es un subconjunto de *generalization*.

ASSOCIATION ComponentCompositeGeneralization-Composite
IS Composition-1
[Composite: whole;
ComponentCompositeGeneralization: part;
child: role1;
componentGeneralization: role2;
1: mult1;
1: mult2;
+: visibility1;
+: visibility2]
CONSTRAINED-BY
child: redefines specific
componentGeneralization: subsets generalization
END

Figura 7.6 – Formalización de la asociación *ComponentCompositeGeneralization-Composite* en NEREUS

A continuación se detalla la especificación NEREUS del metamodelo origen del refactoring *Extraer Composite* y en el Anexo F se formaliza parcialmente el metamodelo UML.

PACKAGE ExtractCompositeSourceMetamodel

IMPORTS Kernel

CLASS AssEndComposite
IS-SUBTYPE-OF Property
ASSOCIATES
<< AssEndComposite-Composite >>,
<< AssEndComposite-CompositeComponentAssoc >>
TYPES AssEndComposite
FUNCTIONS
create: * → AssEndComposite
AXIOMS assEnd: AssEndComposite
get-aggregation(assEnd) = “shared” or get-aggregation(assEnd) = “composite”
END-CLASS

CLASS AssEndComponent
IS-SUBTYPE-OF Property
ASSOCIATES
<< AssEndComponent-Component >>,
<< AssEndComponent-CompositeComponentAssoc >>
TYPES AssEndComposite
FUNCTIONS
create: * → AssEndComponent
AXIOMS
END-CLASS

CLASS CompositeComponentAssoc
IS-SUBTYPE-OF Association
ASSOCIATES
<< AssEndComposite-CompositeComponentAssoc >>,
<< AssEndComponent-CompositeComponentAssoc >>
TYPES CompositeComponentAssoc
FUNCTIONS
create: * → CompositeComponentAssoc
isEquivalentTo: CompositeComponentAssoc x CompositeComponentAssoc ->boolean
AXIOMS c, cc: CompositeComponentAssoc; AP: Association-Property$_0$
size (get-memberEnd(AP, cc)) = 2

isEquivalentTo(c,cc) = (get-isDerived(c) = get-isDerived(cc) and
get-visibility(c) = get-visibility(cc) and
(get-name(c)=get-name(cc) or get-name(c)<>get-name(cc))and
forAll $_{end1}$ (get-memberEnd(AP,c), [
exists $_{end2}$ (get-memberEnd(AP,cc), [
(get-name(end1)=get-name(end2) or get-name(end1)<>get-name(end2))and
get-visibility(end1)=get-visibility(end2) and get-isLeaf(end1)=get-isLeaf(end2) and
get-isStatic(end1)=get-isStatic(end2) and
get-isDerived(end1)=get-isDerived(end2) and
get-isReadOnly(end1)=get-isReadOnly(end2) and
get-aggregation(end1)=get-aggregation(end2) and

get-upper(end1)=get-upper(end2) and get-lower(end1)=get-lower(end2) and
get-subsettedProperty(end1)=get-subsettedProperty(end2) and
get-redefinedProperty(end1)=get-redefinedProperty(end2)]]))
END-CLASS

CLASS Component
IS-SUBTYPE-OF M_Class
ASSOCIATES
<< Component-ComponentCompositeGeneralization >>,
<< Component-ComponentLeafGeneralization >>,
<< AssEndComponent-Component >>
TYPES Component
FUNCTIONS
create: * → Component
AXIOMS c: Component; Ap: AssEndComponent-Component;
Aa: AssEndComponent-CompositeComponentAssoc;
Cp: Component-ComponentCompositeGeneralization;
Oc: Class-Operation; Cc: ComponentCompositeGeneralization-Composite

forAll $_{a1,a2}$ (collect $_{assEnd}$ (get_associationEnd(Ap,c),
[get_association(Aa, assEnd)]), [a1=a2 or isEquivalentTo(a1,a2)])

forAll $_{cl}$ (collect $_{child}$ (get_compositeSpecialization (Cp, c),
[get_child(Cc,child)]),
[exists $_{op}$ (get_ownedOperation(Oc, cl),
[forAll $_{c}$ (excluding (collect $_{child}$ (get_compositeSpecialization(Cp, c),
[get_child(Cc,child)]), cl),
[exists $_{o}$ (get_ownedOperation(Oc, c)),
[isEquivalentTo (op, o)]]]
END-CLASS

CLASS ComponentCompositeGeneralization
IS-SUBTYPE-OF Generalization
ASSOCIATES
<< ComponentCompositeGeneralization-Composite >>,
<< Component-ComponentCompositeGeneralization >>
TYPES ComponentCompositeGeneralization
FUNCTIONS
create: * → ComponentCompositeGeneralization
AXIOMS
END-CLASS

CLASS ComponentLeafGeneralization
IS-SUBTYPE-OF Generalization
ASSOCIATES
<< Component-ComponentLeafGeneralization >>,
<< ComponentLeafGeneralization-Leaf >>
TYPES ComponentLeafGeneralization
FUNCTIONS
create: * → ComponentLeafGeneralization
AXIOMS
END-CLASS

CLASS Leaf
IS-SUBTYPE-OF M_Class
ASSOCIATES
<< ComponentLeafGeneralization-Leaf >>
TYPES Leaf
FUNCTIONS
create: * → Leaf
AXIOMS
END-CLASS

CLASS Composite
IS-SUBTYPE-OF M_Class
ASSOCIATES
<< ComponentCompositeGeneralization-Composite >>,
<< AssEndComposite-Composite >>
TYPES Composite
FUNCTIONS
create: * → Composite
AXIOMS
END-CLASS

ASSOCIATION AssEndComposite-Composite
IS Bidirectional-1 [Composite: class1; AssEndComposite: class2; participant: role1; associationEnd: role2; 1: mult1; 1: mult2; +: visibility1; +: visibility2]
END

ASSOCIATION AssEndComponent-Component
IS Bidirectional-5 [AssEndComponent: class1; Component: class2; associationEnd: role1; participant: role2; 2..*: mult1; 1: mult2; +: visibility1; +: visibility2]
END

ASSOCIATION AssEndComposite-CompositeComponentAssoc
IS Bidirectional-1 [AssEndComposite: class1; CompositeComponentAssoc: class2; assEndComposite: role1; association: role2; 1: mult1; 1: mult2; +: visibility1; +: visibility2]
END

ASSOCIATION AssEndComponent-CompositeComponentAssoc
IS Bidirectional-1 [AssEndComponent: class1; CompositeComponentAssoc: class2; assEndComponent: role1; association: role2; 1: mult1; 1: mult2; +: visibility1; +: visibility2]
END

ASSOCIATION Component-ComponentCompositeGeneralization
IS Bidirectional-5 [ComponentCompositeGeneralization: class1; Component: class2; compositeSpecialization: role1; parent: role2; 2..*: mult1; 1: mult2; +: visibility1; +: visibility2]
CONSTRAINED-BY
parent: redefines general
END

ASSOCIATION Component-ComponentLeafGeneralization
IS Bidirectional-5 [ComponentLeafGeneralization: class1; Component: class2; leafSpecialization: role1; parent: role2; 1..*: mult1; 1: mult2; +: visibility1; +: visibility2]

CONSTRAINED-BY
parent: redefines general
END

ASSOCIATION ComponentCompositeGeneralization-Composite
IS Composition-1 [Composite: whole; ComponentCompositeGeneralization: part; child: role1; componentGeneralization: role2; 1: mult1; 1: mult2; +: visibility1; +: visibility2]
CONSTRAINED-BY
child: redefines specific
componentGeneralization: subsets generalization
END

ASSOCIATION ComponentLeafGeneralization-Leaf
IS Composition-1 [Leaf: whole; ComponentLeafGeneralization: part; child: role1; componentGeneralization: role2; 1: mult1; 1: mult2; +: visibility1; +: visibility2]
END

7.2. Traducción de Refactorings OCL a NEREUS

Los contratos de refactorings especificados en OCL son traducidos a NEREUS para asegurar que los modelos transformados en los distintos niveles de abstracción (PIM, PSM, ISM) se mantengan consistentes.

La figura 7.7 muestra un metamodelo para los refactorings que incluye las metaclases *Metamodel* y *Refactoring* y sus asociaciones.

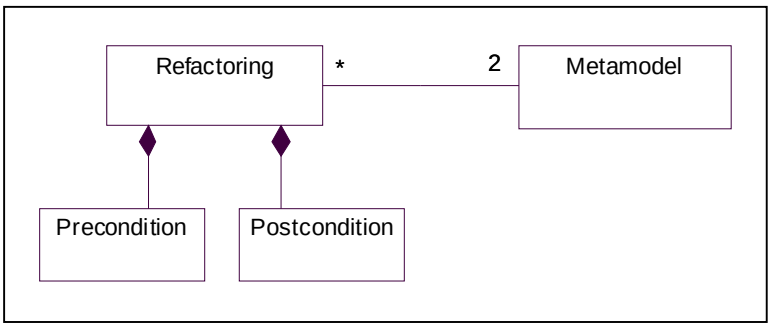


Figura 7.7 – Metamodelo de los refactorings

Un refactoring tiene una composición con las metaclases *Precondition* y *Postcondition*, indicando que estas últimas son parte de un refactoring. La metaclase *Metamodel* describe una familia de metamodelos y *Refactoring* describe a una familia de refactorings entre metamodelos. Por ejemplo, cada uno de los refactoring descriptos en capítulos anteriores para diferentes niveles de abstracción, son instancias de este metamodelo. Cada refactoring vincula dos metamodelos: fuente y destino y posee precondiciones para poder aplicarlo y postcondiciones que deben valer una vez finalizada la transformación.

La formalización en NEREUS de instancias del metamodelo de refactoring se construye automáticamente por medio de la instanciación de esquemas reusables. La figura 7.8 muestra un esquema reusable para traducir los refactorings como vínculos entre metamodelos donde:

REFACTORING *name*
GENERATED-BY create
EFFECTIVE
TYPE *name*
FUNCTIONS
create: *sourceMetamodel x targetMetamodel* → *name*
pre: $Translate_{NEREUS}$ *(OCLexp1)*

get_source: *name* → *sourceMetamodel*

get_target: *name* → *targetMetamodel*

-op_1: T_1 x TP_{11} x TP_{12} x ..... x $TP_{1n} \rightarrow TR_1$
...
-op_r: T_r x TP_{r1} x TP_{r2} x....x $TP_{rp} \rightarrow TR_r$

AXIOMS m1: *sourceMetamodel*; m2: *targetMetamodel*

get_source (create (m1, m2)) = m1

get_target (create (m1, m2)) = m2

op_1 (T_1 , TP_{11} ,TP_{12} ,..... , TP_{1n}) = $Translate_{NEREUS}$ *(*op_1*OCLexp)*
...
op_r (T_r ,TP_{r1} , TP_{r2} ,...., TP_{rp}) = $Translate_{NEREUS}$ *(*op_r*OCLexp)*

$Translate_{NEREUS}$ *(OCLexp2)*
…
END-CLASS

Figura 7.8 – Formalización de refactorings: esquema reusable

- *name* es instanciado con el nombre del refactoring,
- *sourceMetamodel* es instanciado con el nombre del metamodelo fuente del refactoring;
- *targetMetamodel* es instanciado con el nombre del metamodelo destino del refactoring;
- *OCLexp1* es reemplazada por las expresiones OCL que forman la precondición de la transformación y $Translate_{NEREUS}$ *(OCLexp1)* se refiere a la traducción a NEREUS de dicha precondición;
- op_1 ...op_r son reemplazados por los nombres de las operaciones locales del refactoring, las cuales tendrán visibilidad privada, cada T_i será instanciado con el tipo sobre el cual se aplica la operación, TP_{i1} ... TP_{in} serán instanciados con los tipos de los parámetros de las operaciones, TRi es instanciado con el tipo de retorno de la operación;
- op_i*OCLexp* es reemplazado por la expresión OCL que corresponde a la definición de la operación op_i y $Translate_{NEREUS}$ *(*op_i*OCLexp)* corresponde a su traducción a NEREUS;
- *OCLexp2* es reemplazada por las expresiones OCL que forman la postcondición de la transformación y $Translate_{NEREUS}$ *(OCLexp2)* se refiere a la traducción a NEREUS de dicha postcondición.

7.2.1. Traducción a NEREUS del refactoring *Extraer Composite*

La figura 7.9 muestra la formalización del refactoring *Extraer Composite* construido por instanciación del esquema reusable de la figura 7.8, donde se producen los siguientes reemplazos:

```
REFACTORING ExtractComposite
GENERATED-BY create
EFFECTIVE
TYPE  ExtractComposite

FUNCTIONS
   create: ExtractCompositeSourceMetamodel x
           ExtractCompositeTargetMetamodel → ExtractComposite

   get_source: ExtractComposite → ExtractCompositeSourceMetamodel

   get_target: ExtractComposite →  ExtractCompositeTargetMetamodel
   ...

AXIOMS
   m1: ExtractCompositeSourceMetamodel;
   m2: ExtractCompositeTargetMetamodel;
   PP: PackageableElement-Package

   get_source (create (m1, m2)) = m1

   get_target (create (m1, m2)) = m2


   forAll sourceClass
        (select sourceC (get_ownedMember(PP, m1),
                         [oclIsTypeOf(sourceC, Component)],

      [exists targetClass
          (select targetC (get_ownedMember(PP, m2),
                         [oclIsTypeOf(targetC, Component)]),

                              name (targetClass) = name (sourceClass)  and
                              visibility (targetClass) = visibility (sourceClass)

          .....

END-CLASS
```

Figura 7.9 – Formalización del refactoring *Extraer Composite*

- *name* es instanciado por ExtractComposite, el cual es el nombre del refactoring;
- *sourceMetamodel* es instanciado por el nombre del metamodelo fuente del refactoring Extraer Composite, ExtractCompositeSourceMetamodel;
- *targetMetamodel* es reemplazado por el nombre del metamodelo destino del refactoring Extraer Composite, ExtractCompositeTargetMetamodel;
- no se producirán los reemplazos de *OCLexp1* y $Translate_{NEREUS}$ *(OCLexp1*) ya que este refactoring no posee precondiciones;
- op_1 ...op_r no serán reemplazados porque este refactoring no tiene operaciones locales adicionales;
- *OCLexp2* es reemplazada por las expresiones OCL que forman la postcondición de la transformación y $Translate_{NEREUS}$ *(OCLexp2*) corresponde a su traducción a NEREUS, mostada en la figura con sombreado gris.

A continuación se detalla la generación de la expresión NEREUS correspondiente a la aplicación de la función $Translate_{NEREUS}$ *(OCLexp2*), donde *OCLexp2* es la postcondición del refactoring *Extraer Composite* y teniendo en cuenta las reglas de transformación detalladas en la figura 7.4.

OCLexp2 =

```
-- Para toda clase Component en el paquete source,
source.ownedMember → select(oclIsTypeOf(Component)) →
   forAll ( sourceClass |

   -- existe una clase Component en el paquete target tal que,
   target.ownedMember →  select(oclIsTypeOf(Component)) →
      exists ( targetClass |

         -- atributos heredados de NamedElement
         targetClass.name = sourceClass.name and
         targetClass.visibility = sourceClass.visibility and... ))
```

$Translate_{NEREUS}$ *(OCLexp2*) =

expresión OCL:

```
-- Para toda clase Component en el paquete source,
source.ownedMember → select (oclIsTypeOf(Component))
```

regla 8, OCL: collection -> op (v: Elem | bool-expr-with-v) donde,

collection: source.ownedMember

regla 4, OCL: context A object.rolename donde,

A: PackageableElement-Package
object: source
rolename: ownedMember

regla 4, NEREUS: get_ rolename (a, object) donde a: A

get_ownedMember(PP, m1) donde
PP: PackageableElement-Package
m1: *sourceMetamodel*

op: select

bool-expr-with-v : oclIsTypeOf(Component)

regla 1, OCL: v. operation (v') donde,
operation: oclIsTypeOf
v: source (elemento al cual se aplica la operación)
v': Component

regla 1, NEREUS: operation (v, v')

oclIsTypeOf(source, Component)

regla 8, NEREUS: op_v (collection, [Translate $_{NEREUS}$ (bool-expr-with-v)])
donde v es cada uno de los elementos de la colección renombrado por sourceC

select $_{sourceC}$ (get_ownedMember(PP, m1),
[oclIsTypeOf(sourceC, Component)]),

Esta expresión da como resultado una colección de clases a la cual se aplica la operación *forAll* para recorrer cada una de estas clases (*sourceClass*):

```
source.ownedMember → select(oclIsTypeOf(Component)) →
   forAll ( sourceClass |

      target.ownedMember →  select(oclIsTypeOf(Component)) →
         exists ( targetClass |
            targetClass.name = sourceClass.name and

            targetClass.visibility = sourceClass.visibility and... ))
```

regla 8, OCL: collection -> op (v: Elem | bool-expr-with-v) donde,

collection: source.ownedMember → select(oclIsTypeOf(Component))
(la cual se tradujo en el paso previo)

op: forAll

bool-expr-with-v :

```
   target.ownedMember →  select(oclIsTypeOf(Component)) →
      exists ( targetClass |
         targetClass.name = sourceClass.name and

         targetClass.visibility = sourceClass.visibility and... ))
```

La traducción de la primera línea de esta expresión es similar a la obtenida en el paso previo, por lo tanto por aplicación de las reglas 8, 4 y 1 a la subexpresión:

target.ownedMember → select(oclIsTypeOf(Component))

se obtiene

select $_{targetC}$ (get_ownedMember(PP, m2),
[oclIsTypeOf(targetC, Component)])

Esta expresión da como resultado una colección de clases a la cual se aplica la operación *exists* a cada una de estas clases (*targetClass*):

regla 8, OCL: collection -> op (v: Elem | bool-expr-with-v) donde,

collection: target.ownedMember → select(oclIsTypeOf(Component))
(la cual se tradujo en el paso previo)

op: exists

bool-expr-with-v :
targetClass.name = sourceClass.name and
targetClass.visibility = sourceClass.visibility ...

regla 5, OCL: OCLexp1 opBin OCLexp2 donde,
OCLexp1: targetClass.name = sourceClass.name
opBin: and
OCLexp2: targetClass.visibility = sourceClass.visibility

regla 5, NEREUS: Translate $_{NEREUS}$ (OCLexp1) opBin Translate $_{NEREUS}$ (OCLexp2)

Translate $_{NEREUS}$ (OCLexp1) =
Translate $_{NEREUS}$ (targetClass.name = sourceClass.name)

regla 5, OCL: OCLexp1 opBin OCLexp2
OCLexp1= targetClass.name
OCLexp2= sourceClass.name

regla 5, NEREUS: Translate $_{NEREUS}$ (OCLexp1) opBin Translate $_{NEREUS}$ (OCLexp2) donde

Translate $_{NEREUS}$ (OCLexp1)
Translate $_{NEREUS}$ (targetClass.name)

regla 3, OCL: v.attribute donde
v = targetClass
attribute = name

regla 3, NEREUS: attribute (v)

name (targetClass)

De manera análoga,
Translate $_{NEREUS}$ (OCLexp2) =
Translate $_{NEREUS}$ (sourceClass.name) =

se traduce a:
name (sourceClass)

por lo tanto:
name (targetClass) = name (sourceClass)

De maner similar:
Translate $_{NEREUS}$ (OCLexp2) =
Translate $_{NEREUS}$ (targetClass.visibility =
sourceClass.visibility)

se traduce a:
visibility (targetClass) = visibility (sourceClass)

por lo tanto se traduce a:

name (targetClass) = name (sourceClass) and
visibility (targetClass) = visibility (sourceClass)

regla 8, NEREUS: se completa de la siguiente manera:

exists $_{targetClass}$
(select $_{targetC}$ (get_ownedMember(PP, m2),
[oclIsTypeOf(targetC, Component)]),
name (targetClass) = name (sourceClass) and
visibility (targetClass) = visibility (sourceClass)

La traducción completa del fragmento de la expresión OCL será:

forAll $_{sourceClass}$
(select $_{sourceC}$ (get_ownedMember(PP, m1),
[oclIsTypeOf(sourceC, Component)],

[exists $_{targetClass}$
(select $_{targetC}$ (get_ownedMember(PP, m2),
[oclIsTypeOf(targetC, Component)]),
name (targetClass) = name (sourceClass) and
visibility (targetClass) = visibility (sourceClass)

Este texto aparece como un axioma en la formalización del refactoring *Extract Composite* (ver figura 7.9).

CAPÍTULO 8

CONCLUSIONES

En esta tesis se presenta un framework para el refactoring de modelos enmarcado en la arquitectura model-driven (MDA). Se presenta una técnica de definición de refactorings basada en metamodelados especificados como contratos OCL. Esta técnica permite describir refactorings de manera uniforme para los distintos modelos de MDA, desde modelos independientes a la plataforma hasta modelos de código.

Además, esta propuesta permite integrar la especificación de los refactorings con técnicas formales. La ventaja de los métodos formales es que evitan ambigüedades e interpretaciones erróneas y al mismo tiempo proveen un proceso de desarrollo de software "correcto" basado en pruebas matemáticas. Sin embargo, las especificaciones formales son generalmente accesibles por especialistas en métodos formales, no todos los diseñadores de software están familiarizados con los formalismos. Para esto, se definió un sistema de traducción de los refactorings al lenguaje formal NEREUS, con el objetivo de que el nivel formal permanezca subyacente al proceso de refactoring de modelos y de permitir la interoperabilidad con otros lenguajes formales. De esta manera, se aprovechan las ventajas de los métodos formales librando al diseñador de manipular los formalismos.

Esta técnica de refactoring se adecua a los distintos procesos de desarrollo de software model-driven permitiendo la evolución de los modelos a fin de mejorar ciertos factores de calidad como modularidad, extensibilidad, reusabilidad, complejidad y eficiencia. En ingeniería forward, donde se transforman modelos independientes de la computación hasta obtener modelos de código, es importante reestructurar cada modelo producto de la secuencia de transformaciones. También, son importantes los refactorings de los modelos generados en los procesos de ingeniería reversa, donde se crean modelos de mayor nivel de abstracción a partir de modelos de código y en reingeniería de software donde se transforma una representación de bajo nivel en otra, mientras se construyen modelos de mayor nivel de abstracción durante el proceso.

A continuación se describen las principales contribuciones de esta propuesta.

- **Clasificación de refactorings en niveles de abstracción**. Se define un conjunto de refactorings aplicables a cada modelo de diseño de la arquitectura MDA, es decir refactorings que se ajustan a modelos independientes de la plataforma (PIM), a modelos específicos a la plataforma (PSM) y modelos específicos a la implementación (ISM).
- **Definición de una técnica de especificación de refactorings basada en metamodelos**. Para cada refactoring se especifica un metamodelo fuente y un metamodelo destino representados por metamodelos MOF. El metamodelo fuente describe una familia de modelos a los cuales puede aplicarse el refactoring correspondiente. Los modelos resultantes de la aplicación del mismo se describen a través del metamodelo destino. Los refactorings se especifican por medio de contratos OCL entre dichos metamodelos, los cuales consisten de precondiciones y

postcondiciones que deben verificar los modelos antes y después de la transformación respectivamente. Esta técnica permite definir los refactorings de manera uniforme para los modelos en los distintos niveles de la arquitectura MDA.

- **Identificación de funcionalidad equivalente en elementos de un modelo**. La identificación de funcionalidad equivalente en un modelo fuente para la aplicación de refactorings, consiste en el proceso de identificar operaciones funcionalmente equivalentes de diferentes clases de un modelo. Este proceso se basa en *matcheos* que permiten comparar dos operaciones basadas en las descripciones de comportamiento de las mismas y han sido tomados y adaptados del trabajo de Zaremski y Wing (1997).
- **Integración de la especificación de los refactorings con técnicas formales**. Los metamodelos origen y destino de cada uno de los refactorings, como así también los contratos de las transformaciones son traducidos al lenguaje de especificación formal NEREUS.
- **Definición de una propuesta de formalización que soporta la interoperabilidad de lenguajes formales**. El lenguaje de especificación formal NEREUS puede ser visto como una notación intermedia abierta a otras especificaciones formales. La semántica de NEREUS fue dada por traducción a CASL (*Common Algebraic Specification Language*). CASL está en el centro de una familia de lenguajes de especificación. Está soportado por herramientas y facilita la interoperabilidad de herramientas de verificación y prototipado.

8.1 Futuros Trabajos

A continuación se detallan futuros trabajos y extensiones aplicables a la presente propuesta de tesis.

- La técnica de especificación de refactorings se muestra sobre modelos de diagramas de clases UML, sin embargo se ha analizado su aplicación sobre otros modelos UML como diagramas de comportamiento y diagramas de casos de uso. Para que el presente trabajo contemple esta extensión, se deberán definir los metamodelos fuente y destino correspondientes a refactorings particulares y especificar las transformaciones en términos de contratos OCL.
- La aplicación de refactorings en diagramas de clases UML puede tener consecuencias sobre los modelos que reflejan otra perspectiva del sistema. Se propone, como extensión a este trabajo, analizar el impacto que los refactorings en diagramas estáticos tienen sobre otros modelos, como por ejemplo diagramas de comportamiento, para que las transformaciones mantengan consistente y coherente al modelo del sistema.
- Todo artefacto de software tiene atributos de calidad tales como, exactitud, robustez, adaptabilidad, reusabilidad, compatibilidad, desempeño, facilidad en su uso, portabilidad y comprensibilidad. Los refactorings podrían ser clasificados según los atributos de calidad que mejoran, permitiendo aplicar refactorings relevantes donde sea necesario y de esta manera incrementar la calidad del software. Las métricas de software asocian un valor numérico a cada atributo de calidad, podrían ser usadas para identificar áreas de problemas y para evaluar las mejoras resultantes después de la aplicación de un refactoring.

- La identificación del contexto de aplicación de una regla de refactoring sobre un modelo fuente, es el proceso de identificar el subconjunto de elementos del modelo fuente que cumple con las precondiciones del contrato de la regla de transformación. Una alternativa para llevar a cabo este proceso sería adaptar la propuesta de Zaremski y Wing (1997) de *matcheo* de especificaciones de componentes.
- El proceso de reestructuración de modelos orientado a objetos, basado en la aplicación sucesiva de reglas de refactoring, permitiría soportar *traceability* si se mantiene el historial de las transformaciones aplicadas.
- Se ha implementado un prototipo que asiste en el refactoring de jerarquías orientadas a objetos a nivel de código C++ (Enriques y otros, 2002). Se pretende prototipar los refactorings de modelos en diferentes niveles de abstracción y sobre diferentes plataformas e integrar los resultados en herramientas CASE MDA.

BIBLIOGRAFÍA

Barbier F.; Henderson-Sellers B.; Opdahl A. y Gogolla M. (2001). The WHOLE-PART Relationship in the Unified Modeling Language: A New Approach. Unified Modeling Language: System Analysis, Design and Development Issues (K. Siau; T. Halpin eds.) Chapter 12, Idea-Group Publishing, USA.

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

Beckert B.; Keller U. y Schmitt P. (2002). Translating the Object Constraint Language into First-order Predicate Logic. Proceedings of VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark.

Bidoit, M. y Mosses, P. (2004). CASL User Manual- Introduction to Using the Common Algebraic Specification Language. Lecture Notes in Computer Science 2900. Springer-Verlag, Berlin Heidelberg New York.

Borland Together (2007). Disponible en: www.borland.com/us/products/together

Bottoni P. (2000). Consistency Checking and Visualization of OCL Constraint. Proceedings of <<UML>> 2000. The Unified Modeling Language. Advancing the Standard, Third International Conference (A. Evans, S. Kent eds.), Lecture Notes in Computer Science 1939, pp. 294-339, Springer-Verlag.

CASE (2006). CASE TOOLS www.objectsbydesign.com/

Cengarle M. y Knapp A. (2001). A Formal Semantics for OCL 1.4. Proceedings of <<UML>> 2001, Modeling Languages, Concepts and Tools (M. Gogolla; C. Kobryn eds.) Lecture Notes in Computer Science 2185, pp. 118, Springer-Verlag.

COFI (2008)Disponible en http://www.informatik.uni-bremen.de/cofi/wiki/index.php/CoFI

Correa, A y Werner, C. (2004) Applying Refactoring Techniques to UML/OCL Models. Proceeding of International Conference UML 2004. Lecture Notes in Computer Science, Vol. 3273. Springer-Verlag. Páginas 173-187

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).

Du Bois, B, Van Gorp, P, Amsel, A, Van Eetvelde, N, Stenten, H, Demeyer, S y Mens, T. (2004) A Discussion of Refactoring in Research and Practice. Reporte Técnico. Universidad de Antwerpen, Bélgica.

Eckel, B. y Allison, C. (2004) Thinking in C++. Volumen 2: Practical Programming. Prentice Hall.

Enriques, S., Mariezcurrena, C. y Ortega, M. (2002) Refactoring de jerarquías orientadas a objetos. Tesis de grado, TR 287. Universidad Nacional del Centro de la Provincia de 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. Proceedings of IEEE International Conference on Software Maintenance, páginas 238-246.

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

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

Favre, L. (2006) A Rigorous Framework for Model Driven Development. Advanced Topics in Database Research, Vol. 5. Editor: Siau, K. Capítulo I, IGP, USA. Páginas 1-27.

Favre, L. (2007) El Lenguaje NEREUS. Reporte interno. Versión 2007. Grupo de Tecnologías de Software, INTIA. Universidad Nacional del Centro de la Provincia de Buenos Aires, Argentina.

Favre, L, Martinez, L y Pereira, C. (2000) Transforming UML Static Models to Object Oriented Code. Proceedings of Technology of Object Oriented Languages and Systems (TOOLS 37). Editorial: IEEE Computer Society. ISBN 0-7695-0918-5. TOOLS-37/PACIFIC 2000. Sydney, Australia. Páginas 170-181.

Favre, L, Martinez, L y Pereira, C. (2001) Una integración de modelos estáticos UML y Eiffel. Proceedings del VII Congreso Argentino de Ciencias de la Computación (CACIC 2001). Calafate, Argentina. Páginas 521-530.

Favre, L, Martinez, L y Pereira, C. (2002) Forward Engineering and UML: From UML Static Models to Eiffel Code. Proceedings of 2002 Information Resources Management Association (IRMA 2002). ISBN 1-930708-39-4. Seattle, USA. Páginas 584-588.

Favre, L, Martinez, L y Pereira, C. (2003) Forward Engineering and UML: From UML Static Models to Eiffel Code. UML and the Unified Process (Liliana Favre editor). Capítulo IX. IRM Press, 2003. ISBN 1-931777-44-6. USA. Páginas 199-217.

Favre, L, Martinez, L y Pereira, C. (2005) Forward Engineering of UML Static Models. Artículo invitado en: Encyclopedia of Information Science and Technology, Volume I-III Mehdi Khosrow-Pour (Editor). Idea Group Publishing, USA. ISBN 1-59140-533-X. Páginas 1212-1217.

Favre, L, Martinez, L y Pereira, C. (2008) Foundations for MDA CASE Tools. Artículo aceptado para Encyclopedia of Information Science and Technology, Second Edition, IGI Global, USA. A ser publicado en 2008.

Favre, L y Pereira, C. (2007) Improving MDA-based Process Quality through Refactoring Patterns. Publicado en: Proceedings of the 1st International Workshop on Software Patterns and Quality (SPAQu'07). Nagoya, Japón. Diciembre, 2007. ISBN 978-4-915256-69-1 C3040. Editor: Information Processing Society of Japan. Páginas 17-22.

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

Fernández Alemán J. y Alvarez Toval A. (2001). Seamless Formalizing the UML Semantics Through Metamodels. Unified Modeling Language: System Analysis, Design and Development Issues (K. Siau; T. Halpin eds.) Chapter 14, Idea-Group Publishing, USA

Folli, A y Mens, T. (2007) Refactoring of UML models using AGG. Proceedings of the Third International ERCIM Workshop on Software Evolution. Francia. Disponible en: ftp://ftp.umh.ac.be/pub/ftp_infofs/2007/ERCIM-Evol2007.pdf

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

France, R y Bieman, J. (2001) Multi-view Software Evolution: A UML-based Framework for Evolving Object-Oriented Software. In Proceedings of International Conference on Software Maintenance (ICSM 2001).

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

Gogolla, M., Bohling, J. y Richters, M. (2005) Validating UML and OCL Models in USE by Automatic Snapshot Generation http://db.informatik.uni-bremen.de/publications.

Hamie A.; Howse J. y Kent S. (1998). Interpreting the Object Constraint Language. Proceedings of Asia-Pacific Software Engineering Conference (APSEC'98), IEEE Computer Society Press.

Ivkovic, I y Kontogiannis, K. (2006) A Framework for Software Architecture Refactoring using Model Transformations and Semantic Annotations. Proceeding of the IEEE Conference on Software Maintenance and Reengineering (CSMR'06). Italia. Páginas 135-144.

Jeanneret, C, Eyer, Leander, Markovic, S y Baar, T. (2006) RoclET – Refactoring OCL Expressions by Transformations. Proceeding of Software & Systems Engineering and their Applications, 19 th International Conference ICCSEA 2006. Francia.

Judson, S., Carver, D. y France, R. (2003) A Metamodeling Approach to Model Refactoring. OOPSLA. California. USA.

Judson, S., France, R. y Carver, D. (2004) Supporting Rigorous Evolution of UML Models. Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems. Páginas: 128-137.

Kerievsky, J. (2004). Refactoring to Patterns. Addison-Wesley.

Kim S. y Carrington D. (1999). Formalizing the UML Class Diagram using OBJECT-Z, Proceedings of UML 99, Lecture Notes in Computer Science1723, pp. 83-98, Springer-Verlag.

Kim S. y Carrington D. (2002). A Formal Model of the UML Metamodel: The UML State Machine and Its Integrity Constraints. Lecture Notes in Computer science 2272, pp. 497, Springer-Verlag.

Kleppe, A., Warmer, J., y Bast, W. (2003) MDA Explained: The Model Driven Architecture™: Practice and Promise. Addison Wesley.

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

Kuske S.; Gogolla M.; Kollmann R. y Kreowski H. (2002). An Integrated Semantics for UML Class, Object and State Diagrams based on Graph Transformation. 3rd Int. Conf. Integrated Formal Methods (IFM'02) (M. Butler; K. Sere editors), Springer-Verlag.

Lano, K. y Bicarregui, J. (1998) Semantics and Transformations for UML Models. Proceeding of <<UML 98>>, The Unified Modeling Language- Beyond the Notation (eds. J. Bezivin; P. Alain Muller) Lecture Notes in Computer Science 1618, páginas 107-119, Springer-Verlag.

Long, Q., Jifeng, H. y Liu, Z. (2005) Refactoring and Pattern-directed Refactoring: A Formal Perspective. UNU/HIST. Report Nº 318.

Mandel L. y Cengarle M. (1999). On the Expressive Power of the Object Constraint Language OCL. http://www.fast.de/projeckte/forsoft/ocl

Markovic, S. y Baar, T. (2005) Refactoring OCL Annotated UML Class Diagrams. Proceeding ACM/IEEE 8 th International Conference of Model Driven Engineering Languages and Systems, (MODELS 2005). LNCS 3713, Springer. Páginas 280-294.

Markovic, S. y Baar, T. (2007) Synchronizing Refactored UML Class Diagrams and OCL Constraints. 1st Workshop on Refactoring Tools, ECOOP07 Conference Workshop, Berlin, Alemania. Danny Dig y Michael Cebulla (Eds.), TU Berlin Technical Report, ISSN 1436-9915, 2007, páginas 15-17.

Massoni, T; Gheyi, R; Borba, P. (2005) A Model-driven Approach to Formal Refactoring. OOPSLA'05. USA. ACM 1-59593-193-7.

MDA (2003). MDA Guide Version 1.0.1. Especificación OMG: omg/03-06-01. Disponible en www.omg.org/mda

MDA (2007). The Model Driven Architecture. Disponible en www.omg.org/mda

Mealy, E, Carrington, D, Strooper, P y Wyeth, P. (2007) Improving Usability of Software Refactoring Tools. Proceedings of the Australian Software Engineering Conference (ASWEC'07) IEEE.

Mens, T. y Tourwé, T. (2004) A Survey of Software Refactoring. IEEE Transactions on Software Engineering. Vol 30. N 2. Páginas 126-139.

Mens, T y Van Deursen, A. (2003) Refactoring: Emerging Trends and Open Problems. Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE), British Columbia.

Meyer, B. (1992) Applying "Design by Contract", in Computer (IEEE), vol. 25, no. 10, páginas 40-51.

Meyers, S. (2005). Effective C++: 55 Specific Ways to Improve Your Programs and Designs. Third Edition. Addison-Wesley.

MOF (2006). MOF: Meta Object Facility (MOF ™), Version 2.0. Especificación OMG: formal/2006-01-01. Disponible en: www.omg.org/mof .

Moore, I. (1995) Guru – A tool for Automatic Restructuring of Self Inheritance Hierarchies. TOOL/ISE.

Mosses, P. (2004). CASL Reference Manual-The Complete Documentation of the Common Algebraic Specification Language. Lecture Notes in Computer Science 2960. Springer-Verlag, Berlin Heidelberg New York.

OCL (2006). Object Constraint Language. Versión 2.0. Especificación OMG: formal/06-05-01. Disponible en: www.omg.org

OMG (2005). Unified Modeling Language Specification. Versión 1.5. Object Management Group. Disponible en www.omg.org.

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

Padawitz P. (2000). Swinging UML: How to Make Class Diagrams and State Machines Amenable to Constraint Solving and Proving. Proceedings <<UML>> 2000 The Unified Modeling Language. Advancing the Standard, Third International Conference (A.Evans, S. Kent eds.) Lecture Notes in Computer Science 1939, 162-177, Springer-Verlag.

Pereira, C. y Favre, L. (2004). Refactoring de Diagramas de Clases UML. VI Workshop de Investigadores en Ciencias de la Computación (WICC 2004). Neuquén, Argentina. Páginas 225-229.

Pereira, C., y Favre, L. (2006). Specifying Refactorings as Metamodel-Based Transformations. Proceedings of 2006 Information Resources Management Association International Conference (IRMA 2006), Washington, D.C., USA. Páginas 264-268.

Pereira, C., Favre, L y Martinez, L. (2004). Refactoring UML Class Diagram. Proceedings of 2004 Information Resources Management Association International Conference (IRMA 2004). New Orleans, USA. ISBN 1-59140-261-1. Páginas 506-510.

Philipps, J. y Rumpe, B. (2003) Refactoring of Programs and Specifications. Publicado en: Practical foundations of business and system specifications. Editores: Kilov and Baclawski. Páginas: 281-297.

Porres, I. (2003) Model Refactorings as Rule-Based Update Transformations. Proceedings of <<UML 2003>>. Lecture Notes in Computer Science, Vol. 2863, Springer Verlag. 159-174.

QVT (2007) MOF Query/Views/Transformation Specification. Especificación OMG: ptc/07/07/07. Disponible en: www.omg.org

Richters, M. y Gogolla, M. (1998) On Formalizing the UML Object Constraint Language OCL. Proceedings International Conference on Conceptual Modeling (ER'98), Singapore, November 16-19, Lecture Notes in Computer Science 1507, pp. 449-464, Springer-Verlag.

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, Issue 4. Editor: John Wiley & Sons, Inc. Páginas: 253-263.

Saksena M.; France R.; Larrondo-Petrie M. y Evett M. (1998). Extending aggregation constructs in UML. Proceedings of <<UML>> 98, The Unified Modeling Language, UML'98 - Beyond the Notation (J. Bezivin; P. Alain Muller, eds.). Lecture Notes in Computer Science 1618, pp. 434- 441, Springer-Verlag.

Simmonds, J y Mens, T. (2002) A Comparison of Software Refactoring Tools. Reporte Técnico. URL: ftp://prog.vub.ac.be/tech_report/2002/vub-prog-tr-02-15.pdf

Stevens P. (2001) On Associations in the Unified Modeling Language. Proceedings <<UML>> 2001-Modeling Languages, Concepts and Tools, Lecture Notes in Computer Science 2185 (M. Gogolla and C. Kobryn eds.), pp. 361, Springer-Verlag.

Stroggylos, K y Spinellis, D. (2007) Refactoring – Does it improve software quality?. Proceedings of the 5th International Workshop on Software Quality. ICSE Workshops. Página 10.

Sunyé, G., Pollet, D., LeTraon y Jézéquel, J. (2001) Refactoring UML Models. Proceedings of UML 2001. Lecture Notes in Computer Science, Vol. 2185, Springer-Verlag. Páginas 134-138.

Thomas, D. (2005) Refactoring as Meta Programming?. Journal of Object Technology. Volumen 4, No.1. Páginas 7-11.

UML-Superstructure (2007). Unified Modeling Language: Superstructure. Version 2.1.1. Especificación de OMG: formal/07-02-05. Disponible en: www.omg.org.

UML-Infrastructure (2007). Unified Modeling Language: Infrastructure Version 2.1.1 Especificación OMG: formal/07-02-06. Disponible en: www.omg.org.

Van Gorp, P., Stenten, H., Mens, T. y Demeyer, S. (2003) Towards automating source-consistent UML Refactorings. Proceedings of <<UML 2003>>. Lecture Notes in Computer Science, Vol. 2863, Springer Verlag. 144-158.

Whittle, J. (2002) Transformations and Software Modeling Languages: Automating Transformations in UML. Proceedings of <<UML 2002>> The Unified Modeling Language. Lecture Notes in Computer Science, Vol. 2460 Springer-Verlag. 227-241.

Zaremski, A. y Wing, J. (1997). Specification Matching of Software Components. ACM Transactions on Software Engineering and methodology, Vol. 6, 4 333-369.

ANEXO A

Diagramas principales del paquete *Kernel* del metamodelo UML

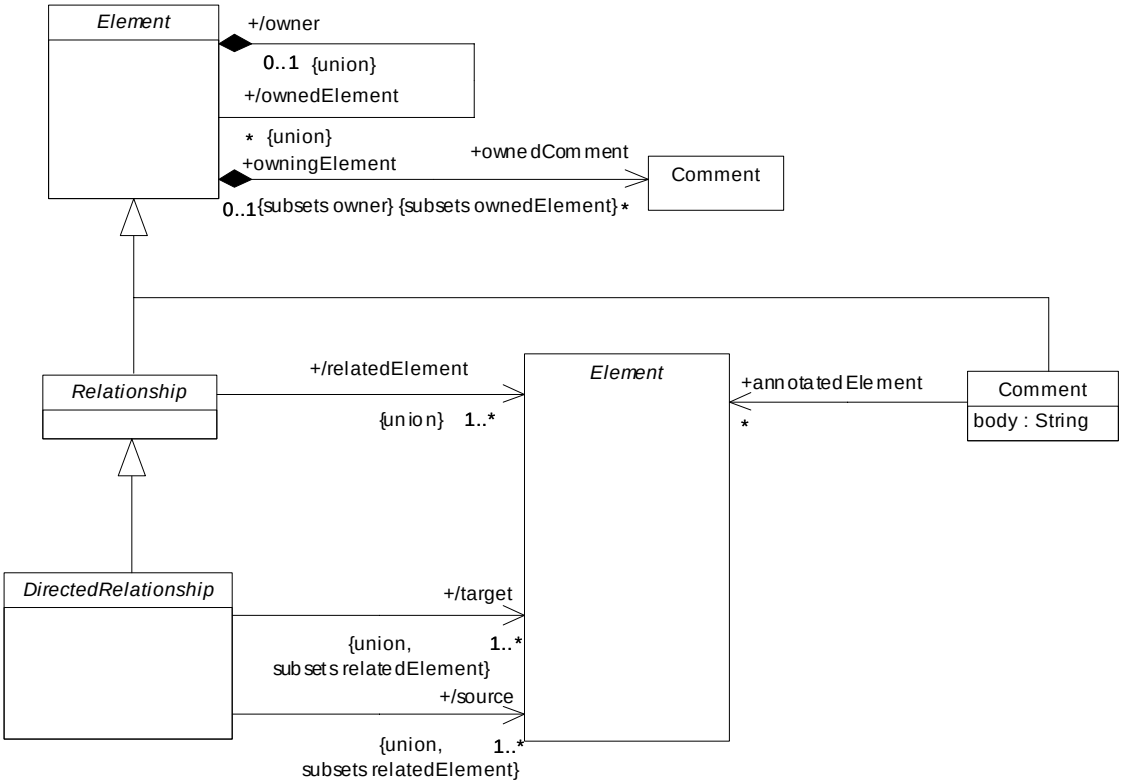


Diagrama de clases *Root*

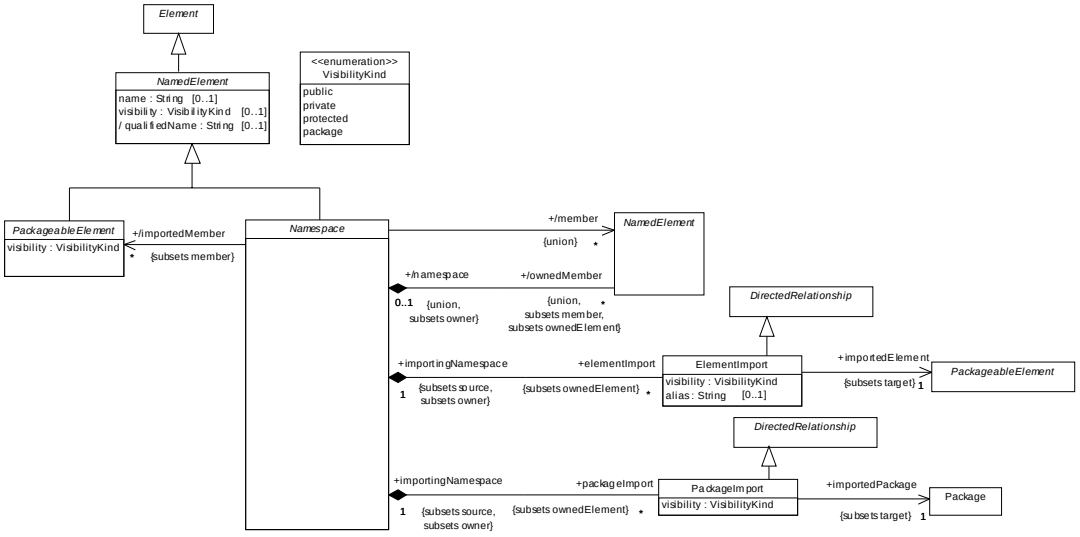


Diagrama de clases de *Namespaces*

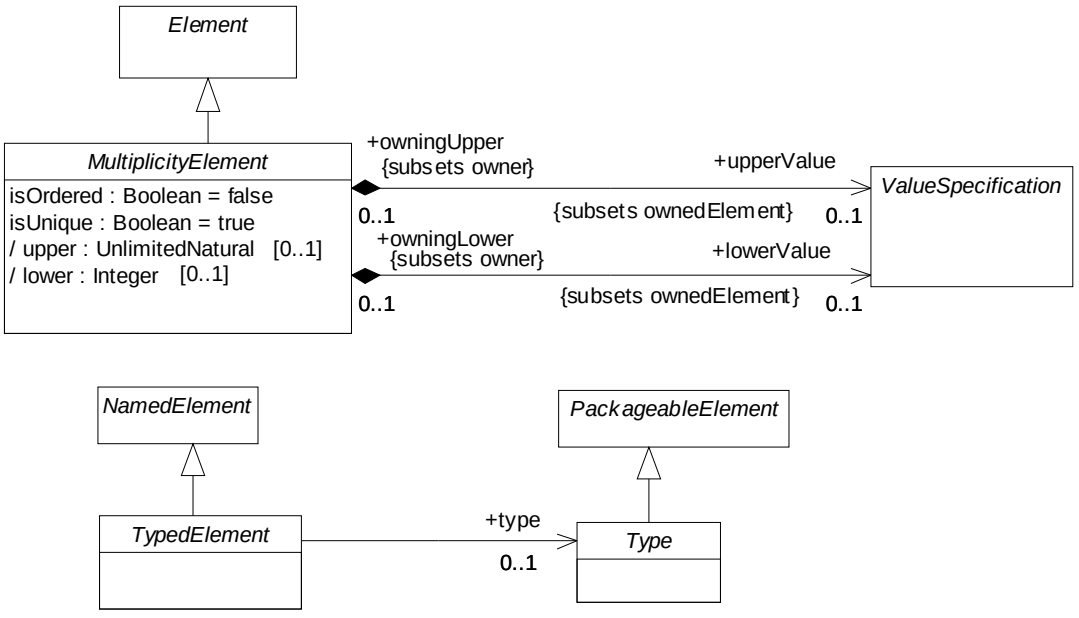


Diagrama de clases de *Multiplicities*

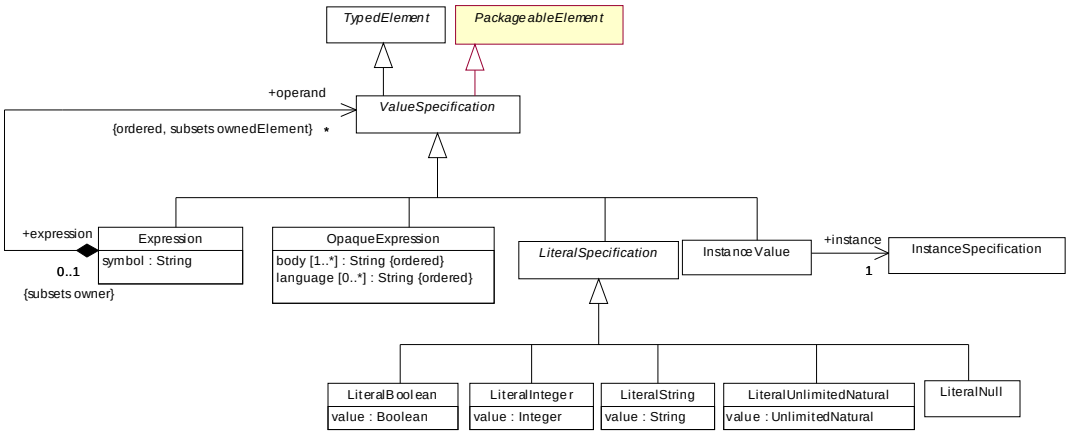


Diagrama de clases de *Expressions*

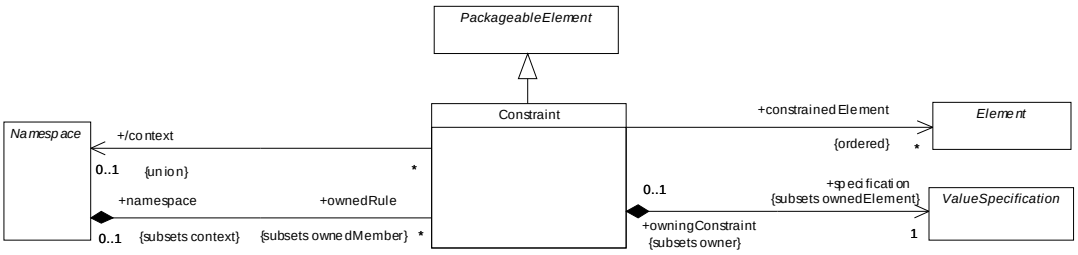


Diagrama de clases de *Constraints*

Operaciones Adicionales

Constraint :: AsOclExp : OclExpression;

```
-- Retorna la restricción como una expresión en OCL.
self.body -> select (oclIsTypeOf(ExpressionInOcl).
                asTypeOf (ExpressionInOcl).bodyExpression
```

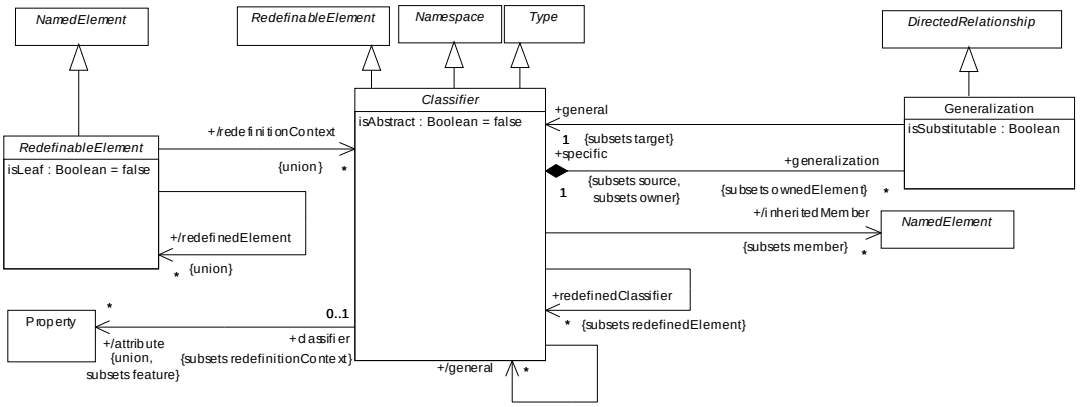


Diagrama de clases de *Classifiers*

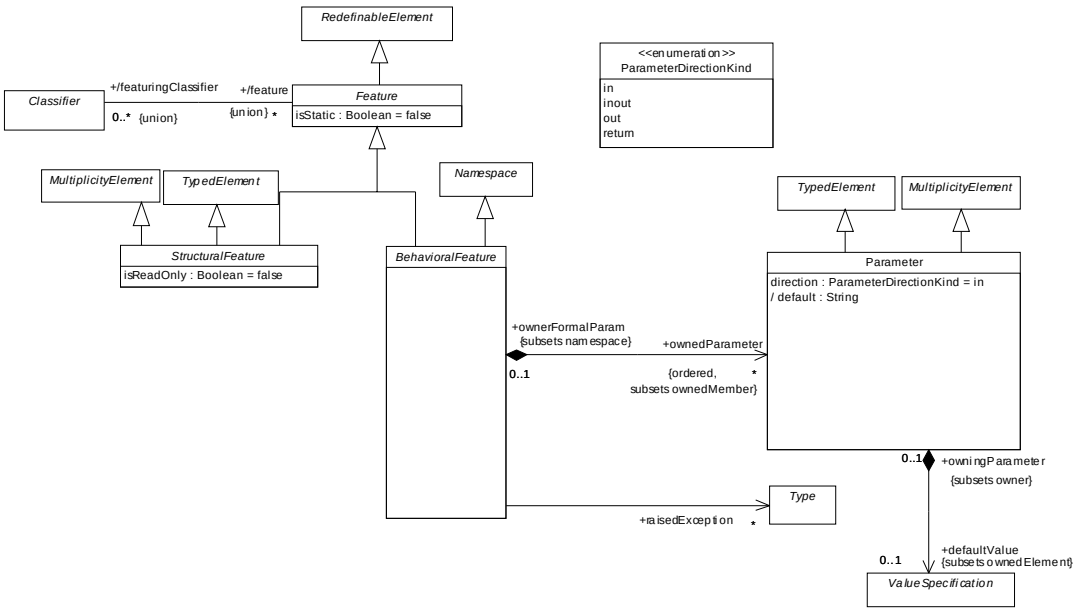


Diagrama de clases de *Features*

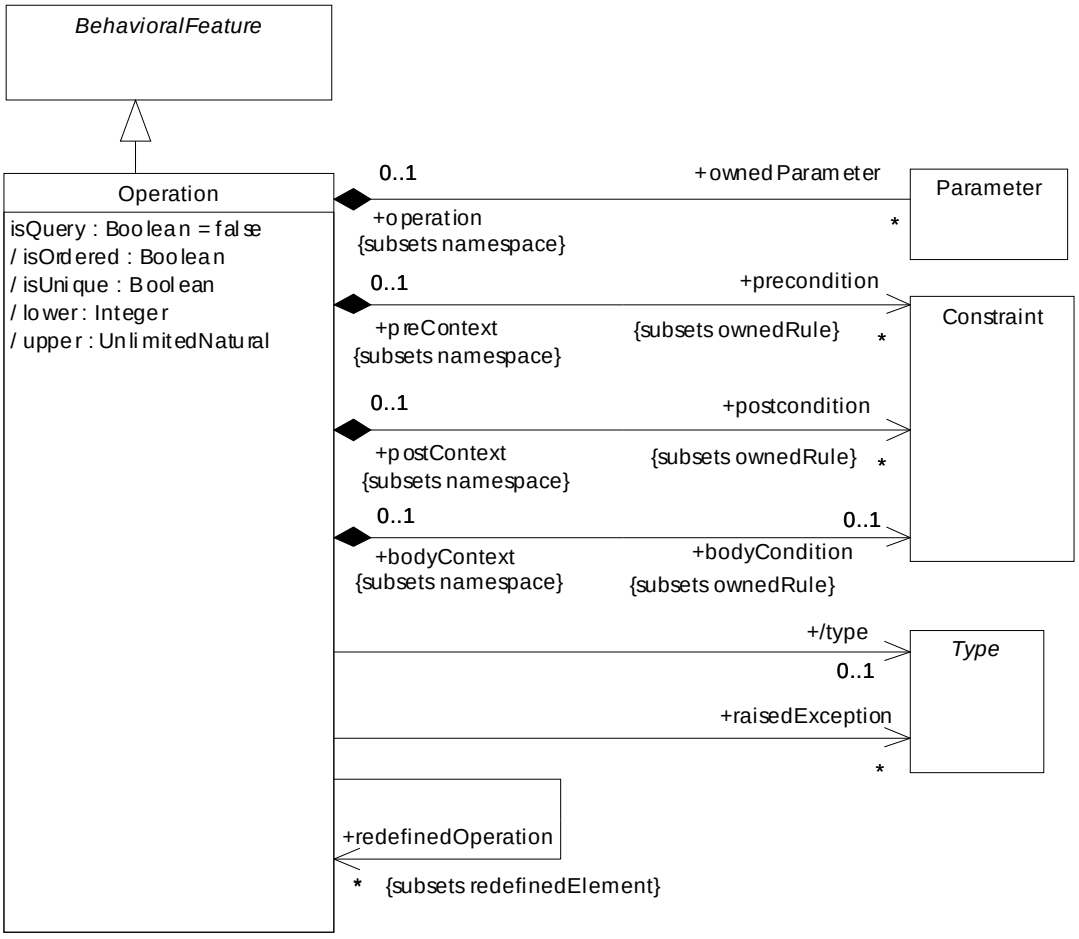


Diagrama de clases de *Operations*

Operaciones Adicionales

Operation :: isEquivalentOperationTo (op: Operation): Boolean;

```
-- Verifica si el argumento es funcionalmente equivalente a la operación a la
-- cual se aplica la operación, a través de
isEquivalentOperationTo (op) =
-- matching de signatura de las funciones, es decir, si tienen
    -- la misma visibilidad,
    self.visibility = op.visibility   and
    -- la misma cantidad de parámetros,
    self.ownedParameter -> size() = op.ownedParameter -> size()   and
    -- tienen los mismos tipos de parámetros aunque pueden no estar en el
    -- mismo orden
    self.ownedParameter ->  forAll ( p |
        op.ownedParameter ->  exists ( pp |
            p.type = pp.type and p.direction = pp.direction ))  and

-- matcheo semántico que compara la especificación del comportamiento
-- dinámico de las funciones:
```

-- *matcheo* exacto pre/post
-- verifica si dos componentes son equivalentes y por lo tanto
-- completamente intercambiables. Dos especificaciones S y S'
-- satisfacen el *matcheo* exacto pre/post si sus precondiciones son
-- equivalentes y sus postcondiciones son equivalentes.
-- match $_{\text{E-pre/post}}$ (Sop, S'self) = (S'self $_{\text{pre}}$ <=> Sop $_{\text{pre}}$) $\wedge$
(Sop $_{\text{post}}$ <=> S'self $_{\text{post}}$)

(self.precondition.AsOclExp implies op.precondition.AsOclExp and
op.precondition.AsOclExp implies self.precondition.AsOclExp and
op.postcondition.AsOclExp implies self.postcondicion.AsOclExp and
self.postcondition.AsOclExp implies op.postcondicion.AsOclExp) or

-- *matcheo* plug-in
-- bajo este *matcheo*, S'self es *matcheada* por cualquier especificación
-- Sop cuya precondición sea más débil (para permitir al menos todas las
-- condiciones que S'self permite) y cuya postcondición sea más fuerte
-- (para proveer una garantía tan fuerte como S'self).
-- match $_{\text{plug-in}}$ (Sop, S'self) = (S'self $_{\text{pre}}$ => Sop $_{\text{pre}}$) $\wedge$
(Sop $_{\text{post}}$ => S'self $_{\text{post}}$)
-- Sop es funcionalmente equivalente a S'self, ya que se puede
-- reemplazar Sop por S'self y tener el mismo comportamiento
-- observable, pero la relación simétrica no se cumple.

(self.precondition.AsOclExp implies op.precondition.AsOclExp and
op.postcondition.AsOclExp implies self.postcondicion.AsOclExp)

Operation::referencedOperation: Set(Operation);
-- Obtiene el conjunto de operaciones invocadas en la implementación de la
-- operación a la cual se aplica la operación.
referencedOperation =
self.ownedMember -> select (oclIsTypeOf(CallOperationAction)).
asTypeOf(CallOperationAction).operation.asTypeOf(Operation) -> select
(class = self.class)

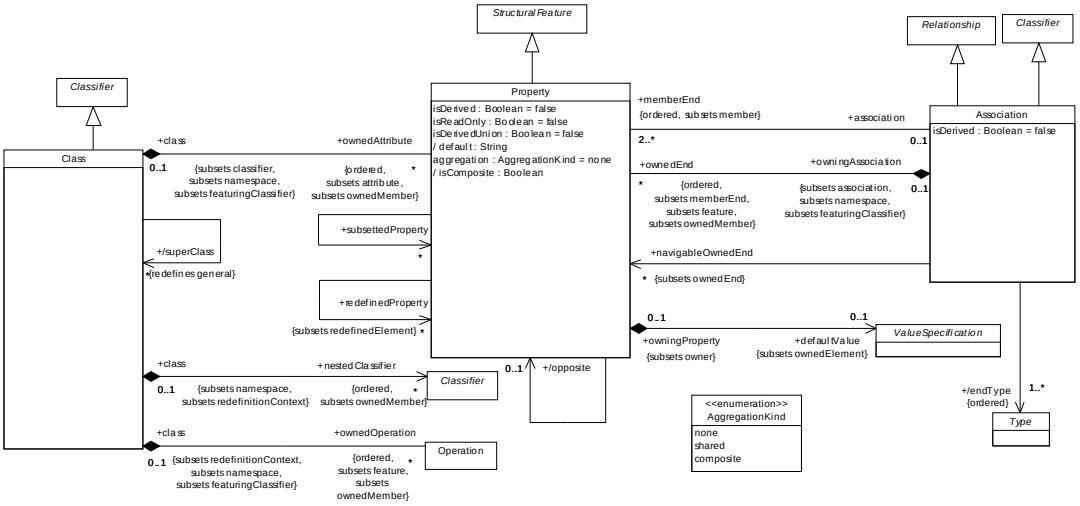


Diagrama de clases de *Classes*

Operaciones Adicionales

Class::allDescendant: Set (Class);

-- Obtiene el conjunto de clases que son descendientes directos o indirectos
-- de la clase a la cual se aplica la operación.

allDescendant =

```
source.ownedMember -> collect (m| m.oclIsTypeOf(Class)).asTypeOf(Class)
    -> select(c| c.allParent -> includes(self)) -> including (self)
```

Class::referencedProperty: Set(Property);

-- Obtiene el conjunto de propiedades referenciadas en las implementaciones de
-- las operaciones de la clase a la cual se aplica la operación o de una clase de su
-- descendencia.

referencedProperty =

```
self.allDescendant.ownedMember -> select
    (oclIsKindOf(StructuralFeatureAction)).
        asTypeOf(StructuralFeatureAction). structuralFeature.asTypeOf(Property)
```

Class::referencedPropertyInOcl: Set(Property);

-- Obtiene el conjunto de propiedades referenciadas en las expresiones OCL
-- (invariantes de clases, precondiciones, postcondiciones y cuerpo de
-- operaciones) de la clase a la cual se aplica la operación o de una clase de su
-- descendencia.

referencedPropertyInOcl =

```
self.allDescendant.ownedRule.body -> select
    (oclIsTypeOf(ExpressionInOcl).asTypeOf (ExpressionInOcl).bodyExpression
        select -> (oclIsTypeOf(PropertyCallExp).asTypeOf (PropertyCallExp).
                referredProperty
```

Property::isEquivalentPropertyTo (p: Property): Boolean;
-- Verifica si el argumento es equivalente a la propiedad a la cual se aplica la
-- operación, es decir, salvo el nombre las demás características se conservan.
isEquivalentPropertyTo (p) =
self.visibility = p.visibility and
self.isLeaf = p.isLeaf and
self.isStatic = p.isStatic and
self.isDerived = p.isDerived and
self.isReadOnly = p.isReadOnly and
self.isDerivedUnion = p.isDerivedUnion and
self.aggregation = p.aggregation and
self.upper = p.upper and
self.lower = p.lower and
self.subsettedProperty = p.subsettedProperty and
self.redefinedProperty = p.redefinedProperty

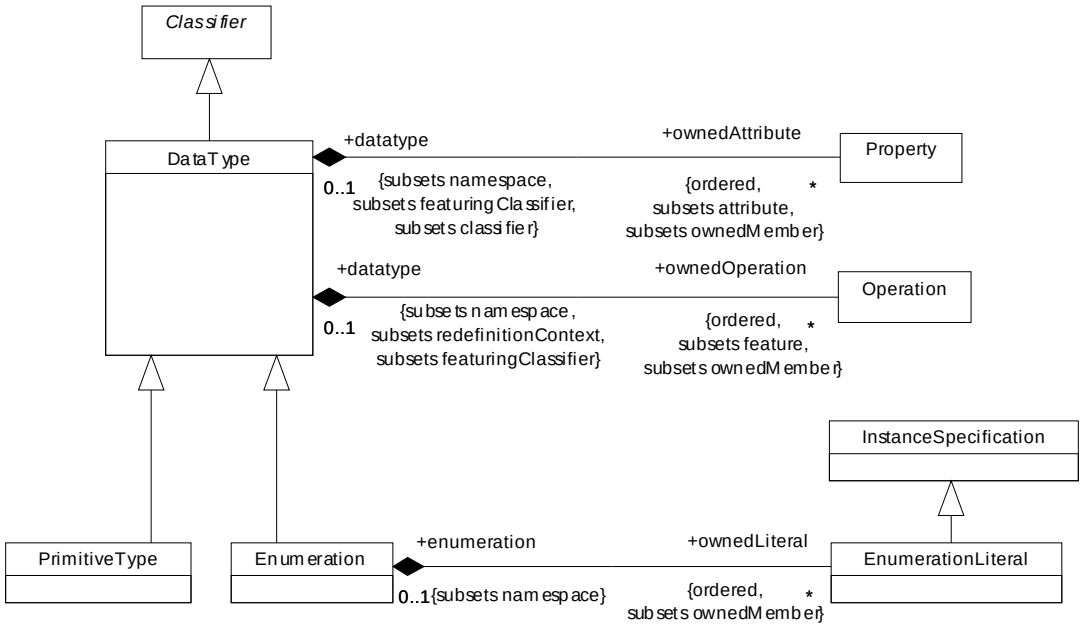


Diagrama de clases de *Data Types*

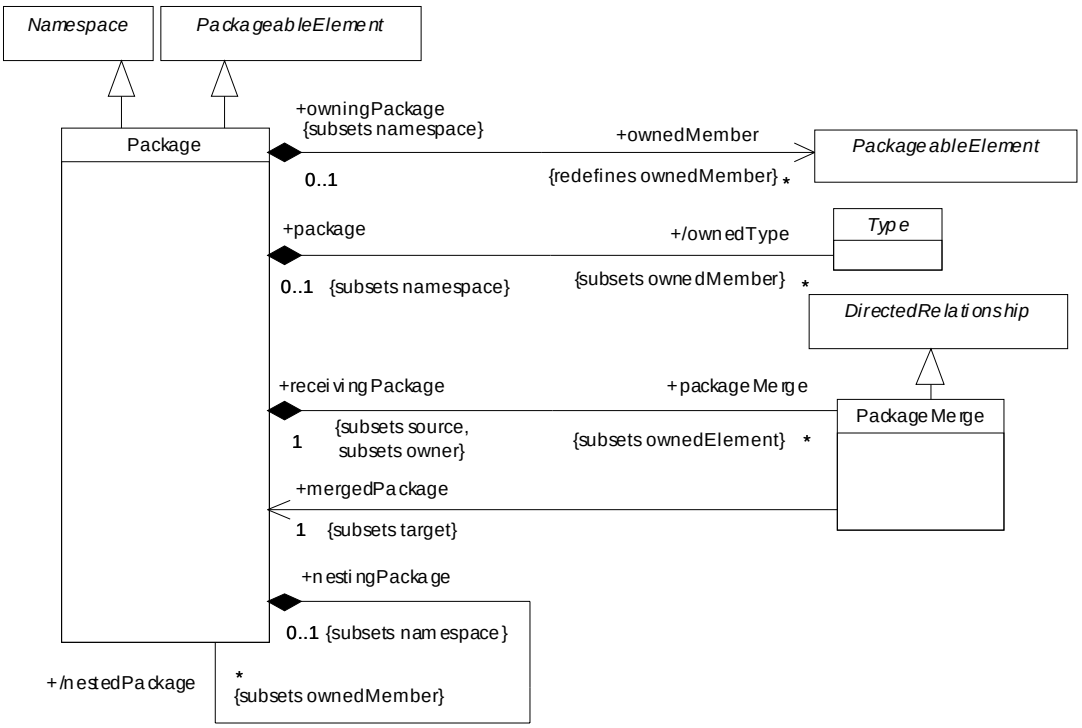


Diagrama de clases de *Packages*

Diagrama del paquete *Interfaces*

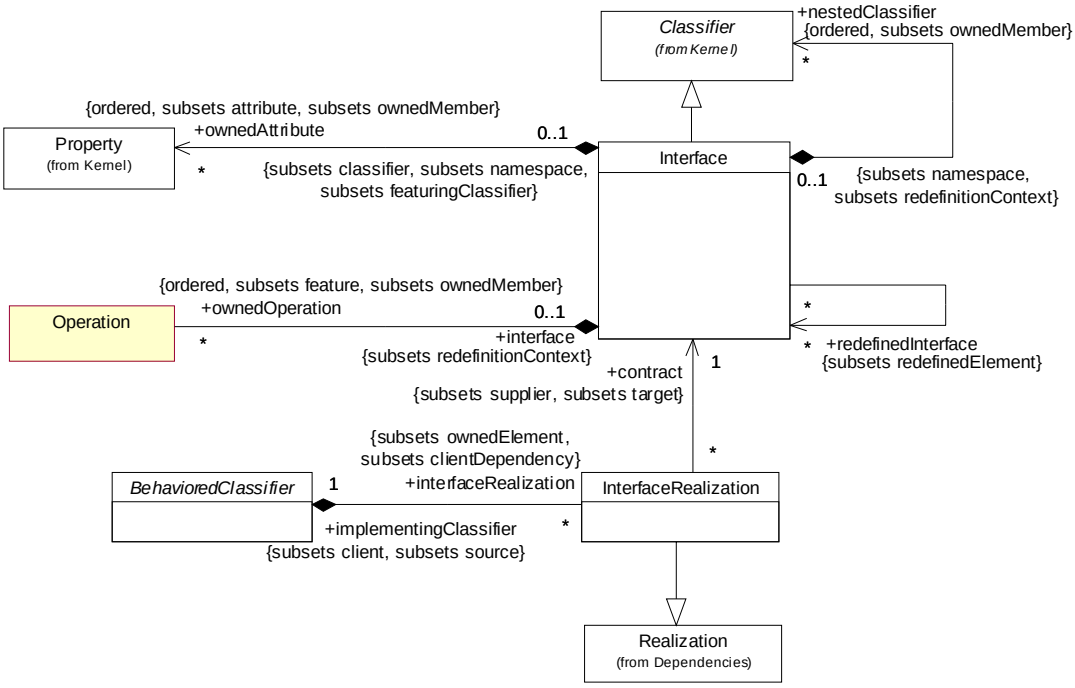


Diagrama de clases de *Interfaces*

Diagrama del paquete *Dependencies*

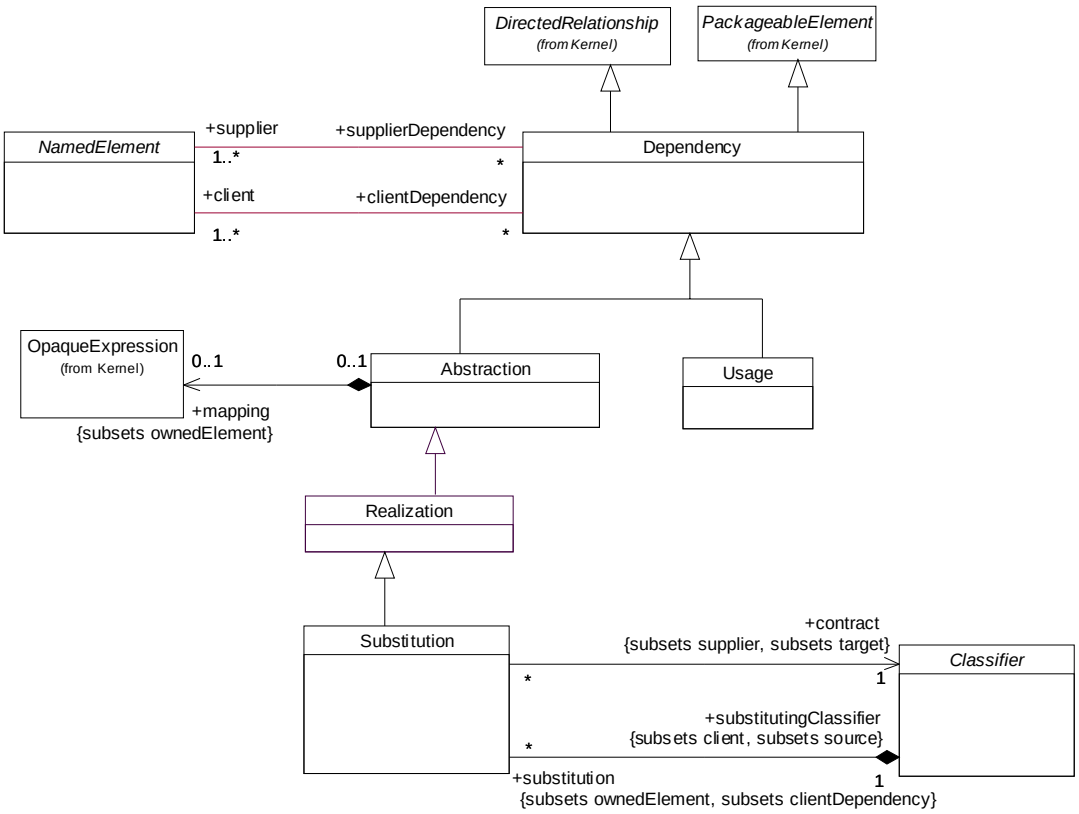


Diagrama de clases de *Dependencies*

ANEXO B

Metamodelo Simplificado Específico a la Plataforma C++

Sintaxis Abstracta

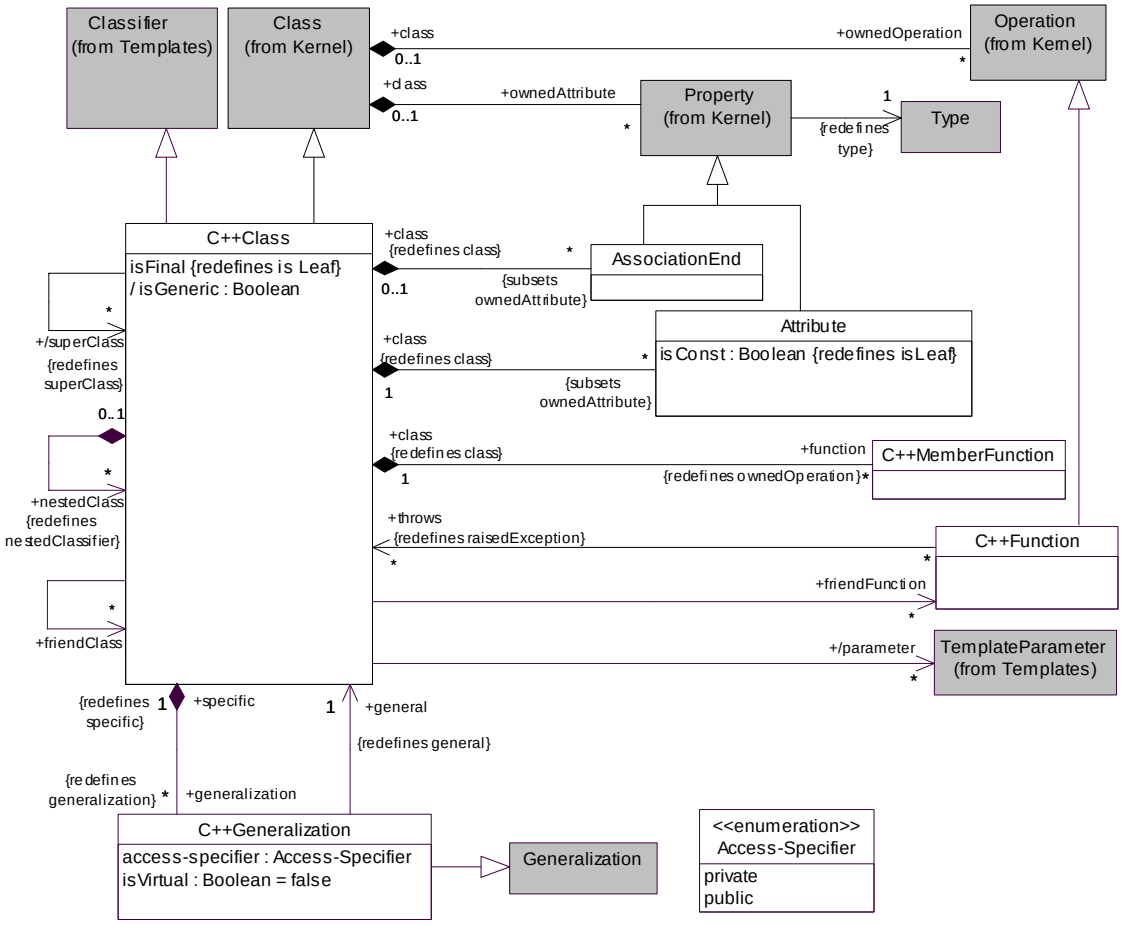


Diagrama de Clases C++

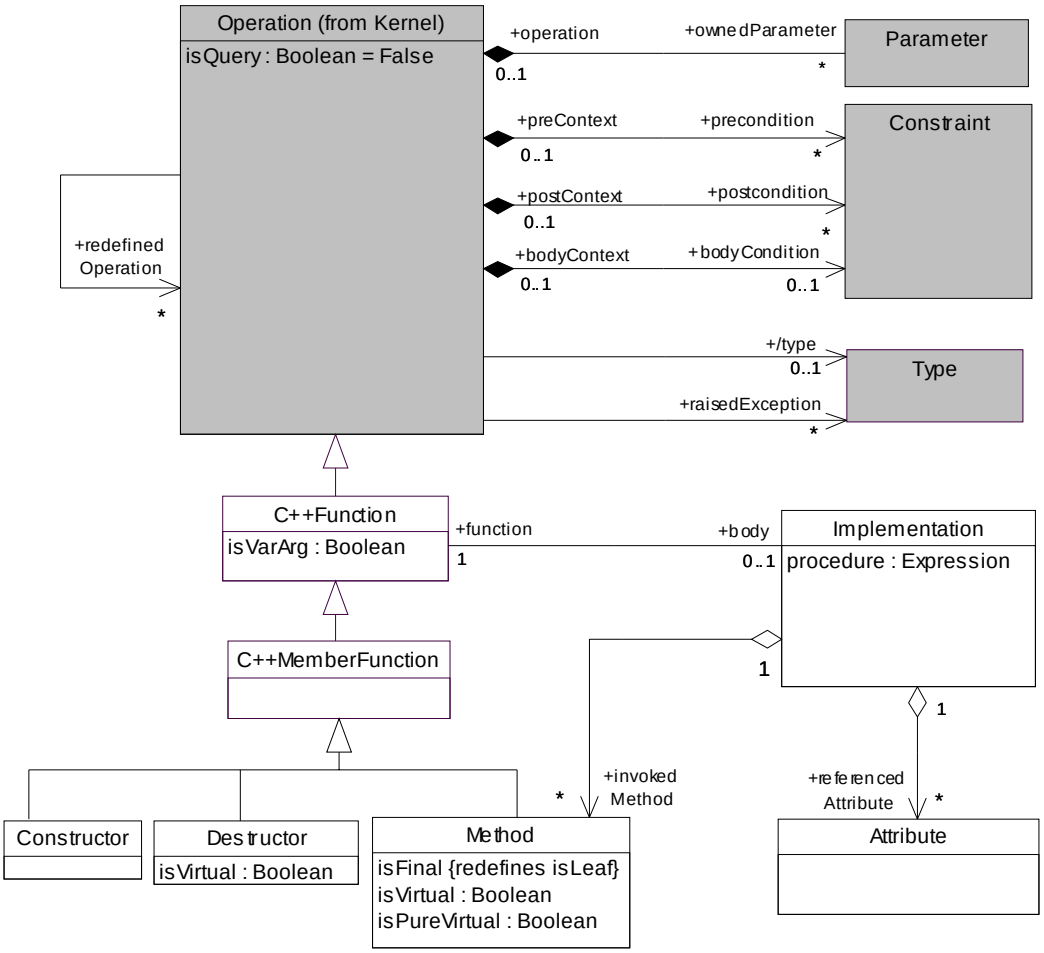


Diagrama de Funciones C++

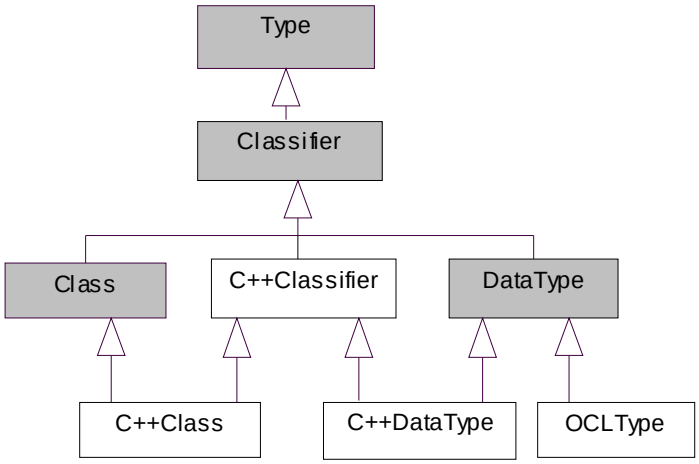


Diagrama de Clasificadores

Descripciones de Clases

AssociationEnd

Representa a los extremos de asociación propios de la clase.

Generalizaciones

- Property (de Kernel)

Atributos

No posee atributos adicionales.

Asociaciones

- class: C++Class [1]

  Referencia a la clase de la cual este extremo es parte. Redefine *Property::class*.

Restricciones

[1] Un extremo de asociación es una propiedad que es miembro de una asociación.

```
self.association->size() = 1
```

Attribute

Representa a los atributos declarados en la clase.

Generalizaciones

- Property (de Kernel)

Atributos

- isConst: Boolean [1]

  Especifica si un atributo es constante. Si es final debe tener valor inicial obligatoriamente. Redefine *RedefinableElement::isLeaf.*

Asociaciones

- class: C++Class [1]

  Referencia a la clase que declara este atributo. Redefine *Property::class*.

Restricciones

[1] Un atributo es una propiedad que es parte de una clase y no es miembro de ninguna asociación.

```
self.class ->size() = 1 and self.association ->isEmpty() and
self.opposite -> isEmpty()
```

C++Class

Una clase C++ describe un conjunto de objetos que comparten las mismas especificaciones de features, restricciones y semántica.

Generalizaciones

- Class (de Kernel), Classifier (de Templates)

Atributos

- isFinal: Boolean

  Especifica si la clase puede tener subclases. Redefine *RedefinableElement::isLeaf*.

- /isGeneric: Boolean

  Especifica si la clase es genérica. Es un atributo derivado.

Asociaciones

- associationEnd: AssociationEnd [*]

  Referencia a los extremos de asociación propios de la clase C++. Subconjunto de *Class::ownedAttribute*.

- attribute: Attribute [*]

  Referencia a las variables propias de la clase C++. Subconjunto de *Class::ownedAttribute*.

- nestedClass: C++Class [*]

  Referencia a las clases C++ que están declaradas dentro del cuerpo de una C++Class (clases anidadas). Subconjunto de *Class::nestedClassifier*.

- /superClass: C++Class [*]

  Referencia a las superclases de una clase C++. Redefine *Class::superClass*. Es derivado.

- function: C++MemberFunction [*]

  Referencia las funciones propias de la clase. Redefine *Class::ownedOperation*.

- generalization: C++Generalization [*]

  Referencia las relaciones de generalización que posee la clase. Redefine *Class::Generalization*.

- friendClass: C++Class [*]

  Referencia las clases amigas de la clase.

- friendFunction: C++Function [*]

  Referencia las funciones amigas de la clase.

- /parameters: TemplateParameter [*]

  Referencia el conjunto de parámetros de la clase. Es derivado

Restricciones

[1] Una clase que tiene alguna función virtual pura debe ser declarada abstracta.

```
self.function -> select (oclIsTypeOf(Method)) -> exists (m |
    m.oclAsType(Method).isPureVirtual)  implies self.isAbstract
```

[2] Una clase declarada final no puede tener subclases, es decir, ninguna clase en el paquete la tendrá como superclase.

```
self.isFinal  implies
    self.package.ownedMember -> select (oclIsTypeOf(C++Class)) -> forAll ( c |
        c.oclAsType(C++Class).superClass <> self )
```

[3] Las funciones privadas de una clase no pueden ser declaradas abstractas.

```
self.function -> select (oclIsTypeOf(Method)) -> forAll (m |
    m.visibility = #private implies
        not m.oclAsType(Method).isPureVirtual)
```

[4] Los métodos finales de una clase no pueden ser declarados abstractos.

```
self.function -> select (oclIsTypeOf(Method)) -> forAll (m |
    m.oclAsType(Method).isFinal implies not m.oclAsType(Method).isVirtual)
```

[5] Una clase es genérica si tiene una signatura template

```
isGeneric = ( self.ownedTemplateSignature -> size ( ) =1 )
```

[6] Parameters se deriva a partir de los parámetros de la signatura template redefinible.

```
/parameters= self.ownedTemplateSignature.parameter
```

[7] Las funciones amigas de una clase son funciones C++, pero no son funciones miembros de clase.

```
self.friendFunction -> forAll (f | f.isTypeOf(C++Function))
```

[8] Una clase puede tener solamente un destructor.

```
self.function -> select (oclIsTypeOf(Destructor)) ->size() <=1
```

Operaciones locales

[1] allSuperClass retorna el conjunto de clases antecesoras a la clase a la cual se aplica la operación

```
allSuperClass(): Set (C++Class)
allSuperClass() = self.superClass -> union (self.superClass.allSuperClass)
```

C++Function

Es una función C++.

Generalizaciones

- Operation (de Kernel)

Atributos

- isVarArg: Boolean

  Especifica si la función puede tener argumentos variables.

Asociaciones

- throws: C++Class [*]

  Referencia a los tipos que representan las excepciones que pueden surgir durante una invocación de esta operación. Redefine *Operation::raisedException*.

- body: Implementation [0..1]

  Referencia a la implementación de la función.

Restricciones

No posee restricciones adicionales.

C++Generalization

Representa una relación de generalización en C++.

Generalizaciones

- Generalization (de Kernel)

Atributos

- access-specifier: Access-Specifier

  Especifica el tipo de acceso a los miembros de la clase base.

- isVirtual:Boolean

  Especifica si la herencia es virtual.

Asociaciones

- general: C++Class [1]

  Referencia a la clase más general en la relación de generalización. Redefine *Generalization::general*.

- specific: C++Class [1]

  Referencia a la clase más específica en la relación de generalización. Redefine *Generalization::specific.*

Restricciones

No posee restricciones adicionales.

C++MemberFunction

Es una función que es miembro de una clase C++.

Generalizaciones

- C++Function

Atributos

No posee atributos adicionales.

Asociaciones

- class: C++Class [1]

  Referencia la clase que posee la función. Redefine *Operation::class*.

Restricciones

No posee restricciones adicionales.

Constructor

Designa una función usada para crear instancias de una clase. No pueden ser invocados explícitamente mediante expresiones de invocación a métodos. Los constructores no poseen tipo de retorno y tienen el mismo nombre de la clase que contiene la declaración del constructor. Las declaraciones de constructores no son heredadas.

Generalizaciones

- C++MemberFunction

Atributos

No tiene atributos adicionales.

Asociaciones

No tiene asociaciones adicionales.

Restricciones

[1] Un constructor no tiene tipo de retorno

```
self.type -> isEmpty()
```

[2] El nombre de un constructor es el mismo nombre de la clase que contiene la declaración.

```
self.name = self.class.name
```

Destructor

Un destructor es una función miembro con igual nombre que la clase, pero precedido por el carácter ~. Una clase sólo tiene una función destructor que, no tiene argumentos y no devuelve ningún tipo. Un destructor realiza la operación opuesta de un constructor, limpiando el almacenamiento asignado a los objetos cuando se crean.

Generalizaciones

- C++MemberFunction

Atributos

- isVirtual: Boolean [1]

  Especifica si el destructor es virtual, es decir, si puede ser redefinido en las subclases.

Asociaciones

No tiene asociaciones adicionales.

Restricciones

[1] Un destructor no tiene argumentos ni tipo de retorno

```
self.ownedParameter -> isEmpty() and self.type -> isEmpty()
```

[2] El nombre de un destructor es el mismo nombre de la clase que contiene la declaración precedido por el carácter ~.

```
self.name = "~".concat(self.class.name)
```

Implementation

Especifica un procedimiento que lleva a cabo el resultado de la función.

Generalizaciones

- Element (de Kernel)

Atributos

- Procedure: Expression [0..1]

  Referencia el procedimiento de la función.

Asociaciones

- function: C++Function [1]

  Referencia a la función a la que pertenece.

- invokedMethod: Method [*]

  Referencia a los métodos invocados en el cuerpo de una función.

- referencedAttribute: Attribute [*]

  Referencia a las variables referenciadas en el cuerpo de una función.

Restricciones

No hay restricciones adicionales.

Method

Declara una función miembro de una clase que puede ser invocada pasando una cantidad fija de argumentos.

Generalizaciones

- C++MemberFunction

Atributos

- isFinal: Boolean [1]

  Especifica si un método es final, es decir, si no puede ser redefinido en las subclases. Redefine *RedefinableElement::isLeaf*.

- isVirtual: Boolean [1]

  Especifica si un método es virtual, es decir, si puede ser redefinido en las subclases.

- isPureVirtual: Boolean [1]

  Especifica si un método es puro, es decir, si no tiene implementación.

Asociaciones

No hay asociaciones adicionales.

Restricciones

No hay restricciones adicionales.

ANEXO C

Metamodelo Simplificado Específico a la Plataforma Java

Sintaxis Abstracta

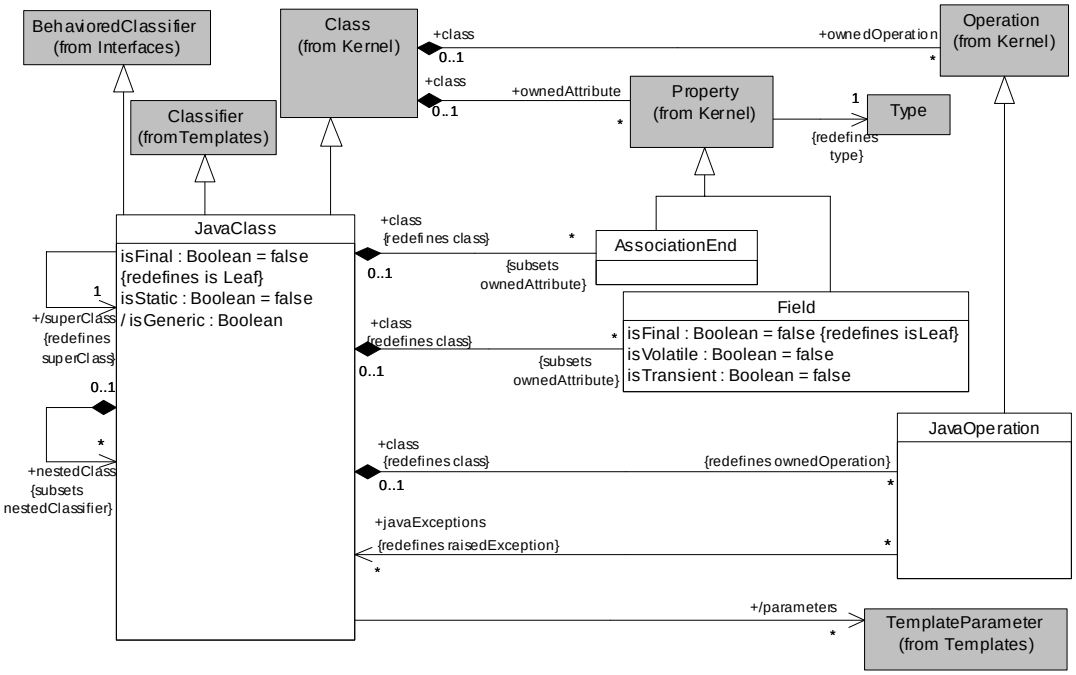


Diagrama de Clases Java

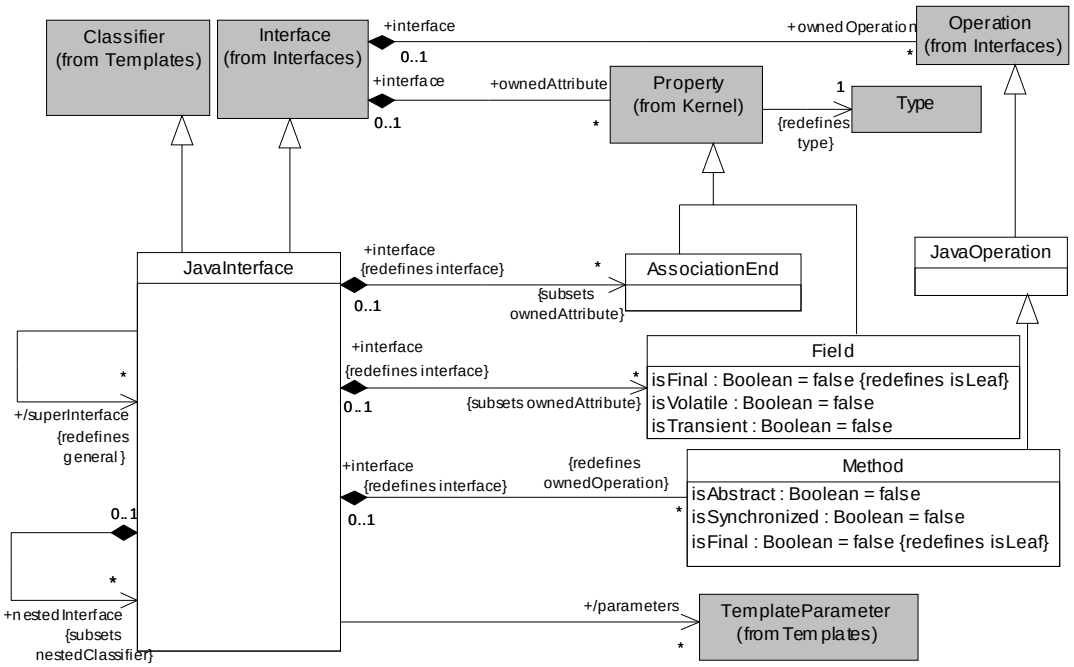


Diagrama de Interfaces Java

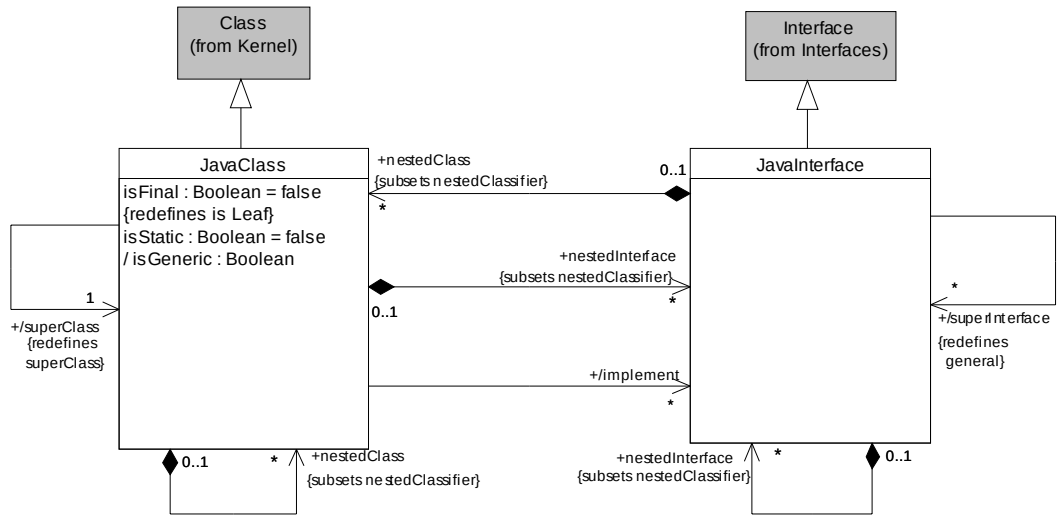


Diagrama de relaciones entre clases e interfaces Java

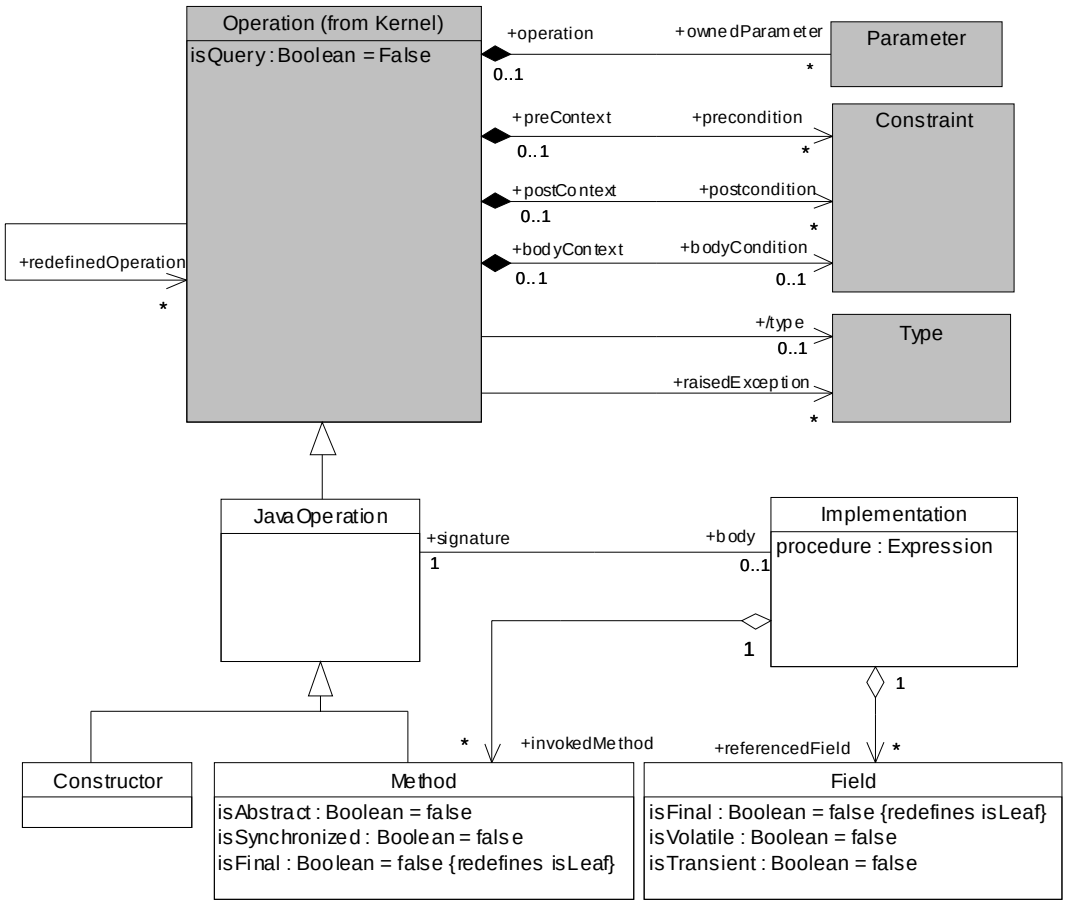


Diagrama de Operaciones Java

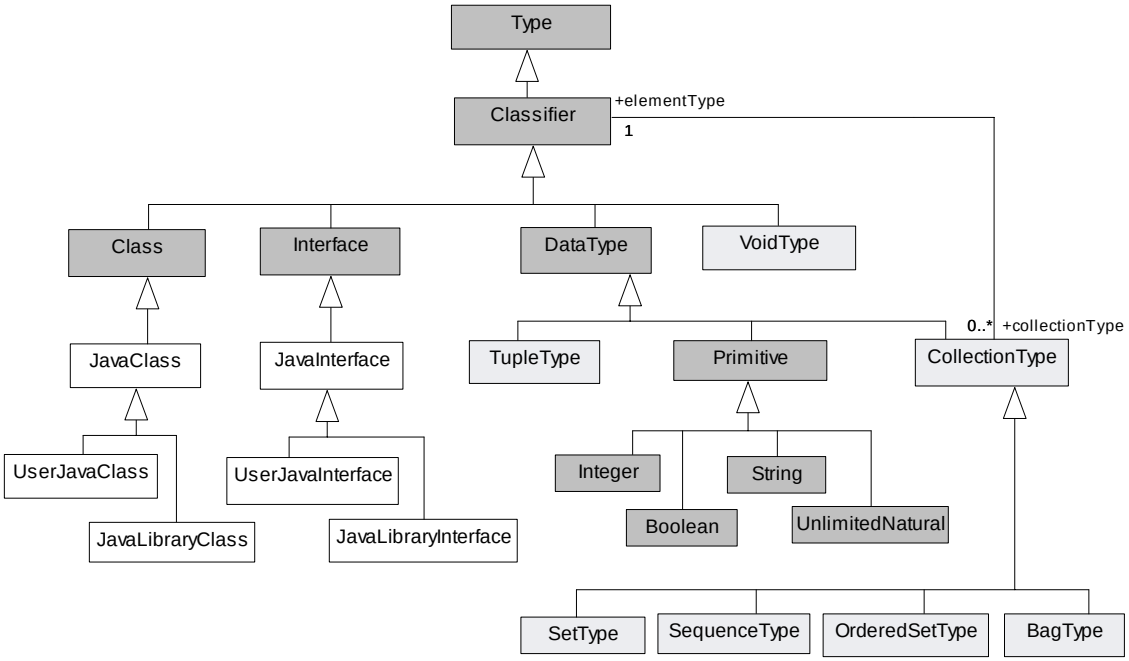


Diagrama de Tipos (UML, OCL y Java)

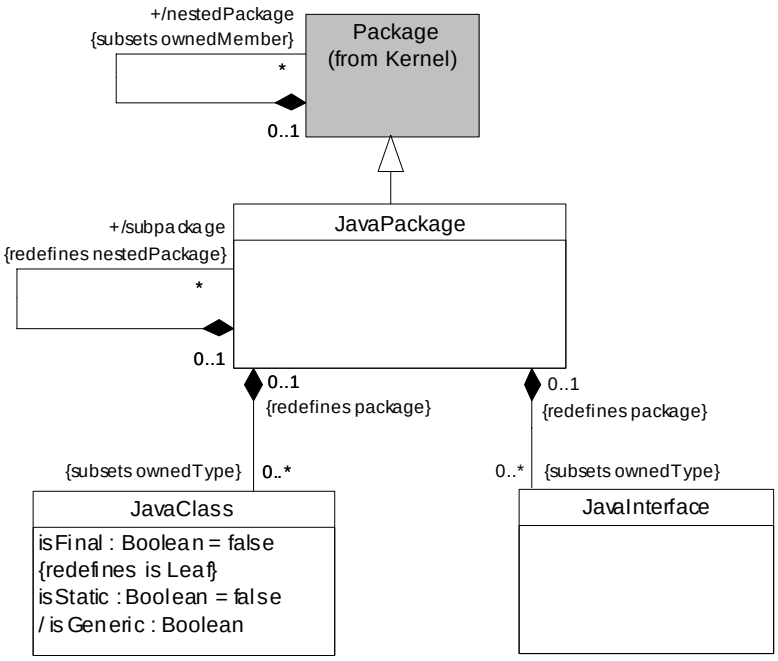


Diagrama de Paquete Java

Descripciones de Clases

AssociationEnd

Representa a los extremos de asociación propios de la clase.

Generalizaciones

- Property (de Kernel)

Atributos

No tiene atributos adicionales.

Asociaciones

- class: JavaClass [0..1]

  Referencia a la clase de la cual este extremo es parte. Redefine *Property::class*.

Restricciones

[1] Un extremo de asociación es una propiedad que es miembro de una asociación.

```
self.association->size() = 1
```

Constructor

Designa una operación usada para crear instancias de una clase. No pueden ser invocados explícitamente mediante expresiones de invocación a métodos. Los constructores no poseen tipo de retorno y tienen el mismo nombre de la clase que contiene su declaración. Las declaraciones de constructores no son heredadas.

Generalizaciones

- JavaOperation

Atributos

No tiene atributos adicionales.

Asociaciones

No tiene asociaciones adicionales.

Restricciones

[1] Un constructor no tiene tipo de retorno

```
self.type -> isEmpty()
```

[2] El nombre de un constructor es el mismo nombre de la clase que contiene la declaración.

```
self.name = self.class.name
```

Field

Especifica los atributos declarados en una clase o en una interfaz.

Generalizaciones

- Property (de Kernel)

Atributos

- isFinal: Boolean [1]

  Especifica si un atributo es final, es decir, si es una constante. Si es final debe tener valor inicial obligatoriamente. Redefine *RedefinableElement::isLeaf*.

- isVolatile: Boolean [1]

  Especifica si un atributo es volátil, es decir, si es accedido de forma asincrónica.

- isTransient: Boolean [1]

  Especifica si un atributo es parte del estado persistente del objeto.

Asociaciones

- class: JavaClass [0..1]

  Referencia a la clase que declara este atributo. Redefine *Property::class*.

Restricciones

[1] Un atributo es una propiedad que es parte de una clase y no es miembro de ninguna asociación.

```
self.class->size() = 1 and self.association->isEmpty() and self.opposite-> isEmpty()
```

Implementation

Especifica el procedimiento de la operación.

Generalizaciones

- Element (de Kernel)

Atributos

- Procedure: Expression [0..1]

  Referencia el procedimiento de la operación.

Asociaciones

- signature: JavaOperation [1]

  Especifica la operación que ésta implementa.

- invokedMethod: Method [*]

  Referencia a los métodos invocados en el cuerpo de una operación.

- referencedField: Field [*]

  Referencia a las variables referenciadas en el cuerpo de una operación.

Restricciones

No hay restricciones adicionales.

JavaClass

Una clase Java describe un conjunto de objetos que comparten las mismas especificaciones de features, restricciones y semántica.

Generalizaciones

- Class (de Kernel), Classifier (de Templates), BehavoiredClassifier (de Interfaces)

Atributos

- isFinal: Boolean

  Especifica si la clase puede tener subclases. Redefine *RedefinableElement::isLeaf*.

- isStatic: Boolean

  Especifica si la clase es estática.

- /isGeneric: Boolean

  Especifica si la clase es genérica. Es un atributo derivado.

Asociaciones

- associationEnd: AssociationEnd [*]

  Referencia a los extremos de asociación propios de la clase Java. Subconjunto de *Class::ownedAttribute*.

- field: Field [*]

  Referencia a las variables propias de la clase Java. Subconjunto de *Class::ownedAttribute*.

- nestedClass: JavaClass [*]

  Referencia a las clases Java que están declaradas dentro del cuerpo de una JavaClass (clases anidadas).Subconjunto de *Class::nestedClassifier*.

- nestedInterface: JavaInterface [*]

  Referencia a las interfaces Java que están declaradas dentro del cuerpo de una JavaClass (interfaces anidadas). Subconjunto de *Class::nestedClassifier*.

- /implement: JavaInterface [*]

  Referencia a las interfaces Java implementadas por esta clase. Es derivado.

- /superClass: JavaClass [1]

  Referencia a la superclase de una clase Java. Redefine *Class::superClass*. Es derivado.

- javaOperation: JavaOperation [*]

  Referencia las operaciones propias de la clase. Redefine *Class::ownedOperation*.

- javaPackage: JavaPackage [0..1]

  Referencia el paquete en el cual está declarada. Redefine *Type::package*.

- /parameters: TemplateParameter [*]

  Referencia el conjunto de parámetros de la clase. Es derivado.

Restricciones

[1] Los clasificadores anidados de una clase o de una interfaz Java sólo pueden ser del tipo JavaClass o JavaInterface.

```
self.nestedClassifier->forAll(c |
    c.oclIsTypeOf(JavaClass) or c.oclIsTypeOf(JavaInterface))
```

[2] Las interfaces implementadas son aquellas referenciadas a través de la relación de realización de interfaz.

```
implement = self.interfaceRealization.contract
```

[3] Una clase que tiene algún método abstracto debe ser declarada abstracta.

```
self.javaOperation->select(op| op.oclIsTypeOf(Method) ) -> exists (m |
    m.oclAsType(Method).isAbstract)  implies self.isAbstract
```

[4] Una clase declarada abstracta no tiene constructor definido explícitamente

```
self.isAbstract implies
    self.javaOperation->select(op| op.oclIsTypeOf (Constructor) ) -> isEmpty()
```

[5] Una clase declarada final no puede tener subclases, es decir, ninguna clase en el paquete la tendrá como superclase.

```
self.isFinal  implies
    self.javaPackage.ownedMember ->select(m|
      m.oclIsTypeOf(JavaClass)) ->forAll (c | c.oclAsType(JavaClass).superClass <> self )
```

[6] Los modificadores protegido, privado o estático sólo pueden ser aplicados a las clases anidadas, es decir, a las declaradas dentro de la declaración de otra clase.

```
(self.visibility = #protected or self.visibility = #private or self.isStatic)   implies
    self.javaPackage.ownedMember->select(m| m.oclIsTypeOf(JavaClass)) ->
      exists ( c | c.oclAsType(JavaClass).nestedClass -> includes(self) )
```

[7] Los métodos privados de una clase no pueden ser declarados abstractos.

```
self.javaOperation->select(op| op.oclIsTypeOf(Method) ) ->
    forAll (m | m.visibility = #private implies not m.oclAsType(Method).isAbstract)
```

[8] Los métodos estáticos de una clase no pueden ser declarados abstractos.

```
self.javaOperation->select(op| op.oclIsTypeOf(Method) ) ->
    forAll (m | m.isStatic implies not m.oclAsType(Method).isAbstract)
```

[9] Los métodos finales de una clase no pueden ser declarados abstractos.

```
self.javaOperation->select(op| op.oclIsTypeOf(Method) ) ->
    forAll (m | m.oclAsType(Method).isFinal implies not m.oclAsType(Method).isAbstract)
```

[10] Una clase es genérica si tiene una signatura template

```
isGeneric = ( self.ownedTemplateSignature -> size ( ) =1 )
```

[11] Parameters se deriva a partir de los parámetros de la signatura template.

/parameters= self.ownedTemplateSignature.parameter

JavaInterface

Describe las características de las interfaces en la plataforma Java.

Generalizaciones

- Interface (de Interfaces).

Atributos

No tiene atributos adicionales.

Asociaciones

- nestedClass: JavaClass [*]

  Referencia todas las clases que son declaradas dentro del cuerpo de una JavaInterface (clases anidadas). Subconjunto de *Interface::nestedClassifer*.

- nestedInterface: JavaInterface [*]

  Referencia todas las interfaces declaradas dentro del cuerpo de una JavaInterface (interfaces anidadas). Subconjunto de *Interface::nestedClassifer*.

- /superInterface: JavaInterface [*]

  Referencia las superinterfaces de una JavaInterface. Es derivado. Redefine *Classifier::general*.

- associationEnd: AssociationEnd [*]

  Referencia los extremos de asociación propios de una JavaInterface. Subconjunto de *Interface::ownedAttribute*

- field: Field [*]

  Referencia los campos propios de una JavaInterface. Subconjunto de *Interface::ownedAttribute*.

- method: Method [*]

  Referencia los métodos propios de una JavaInterface. Redefine *Interface::ownedOperation*.

- javaPackage: JavaPackage [0..1]

  Referencia el paquete en el cual está declarada. Redefine *Type::package*.

Restricciones

[1] Las interfaces en Java son implícitamente abstractas.

```
self.isAbstract
```

[2] Los miembros propios de una interfaz son implícitamente públicos.

```
self.ownerMember->forAll (m| m.visibility = #public)
```

[3] Los clasificadores anidados de una interfaz sólo pueden ser del tipo JavaClass o JavaInterface.

```
self.nestedClassifier->forAll(c |
```

```
c.oclIsTypeOf(JavaClass) or c.oclIsTypeOf(JavaInterface))
```

[4] Una interfaz sólo puede ser declarada privada o protegida si está directamente anidada en la declaración de una clase.

```
(self.visibility= #protected or self.visibility = #private) implies
    self.package.ownedMember->select(m | m.oclIsTypeOf(JavaClass)) ->
        exists(c | c.oclAsType(JavaClass).nestedInterface->includes (self) )
```

[5] Una interfaz sólo puede ser declarada estática si está directamente anidada en la declaración de una clase o interfaz.

```
self.isStatic  implies
    self.package.ownedMember->select(m | m.oclIsTypeOf(JavaClass) ) ->
        exists(c | c.oclAsType(JavaClass).nestedInterface->includes (self) ) or
            self.package.ownedMember->select(m | m.oclIsTypeOf (JavaInterface) ) ->
                exists(i | i.oclAsType(JavaInterface).nestedInterface->includes (self) )
```

[6] Los métodos declarados en una interfaz son abstractos por lo tanto no tienen implementación.

```
self.method->forAll (m| m.isAbstract and m.body->isEmpty() )
```

[7] Los métodos de una interfaz no pueden ser declarados estáticos.

```
self.method->forAll (m| not m.isStatic)
```

[8] Los métodos de una interfaz no pueden ser sincronizados.

```
self.method->forAll (m| not m.isSynchronized)
```

[9] Los campos de una interfaz son implícitamente públicos, estáticos y finales.

```
self.field->forAll (f | f.visibility = #public and f.isStatic and f.siFinal)
```

[10] superInterface se deriva de la relación de generalización.

```
/superInterface = self.generalization.general
```

[11] parameters se deriva a partir de los parámetros de la signatura template.

```
/parameters= self.ownedTemplateSignature.parameter
```

JavaOperation

Es una operación de una clase Java.

Generalizaciones

- Operation (de Kernel)

Atributos

No tiene atributos adicionales.

Asociaciones

- class: JavaClass [0..1]

  Referencia a la clase que declara esta operación. Redefine *Operation::class*.

- javaException: JavaClass [*]

Referencia a los tipos que representan las excepciones que pueden surgir durante una invocación de esta operación.

- body: Implementation [0..1]

  Referencia a la implementación de la operación.

Restricciones

[1] Si una operación es abstracta no tiene implementación

```
self.isAbstract implies self.body->isEmpty()
```

JavaPackage

Es utilizado para agrupar elementos. Sus miembros pueden ser tipos clases o interfaces y subpaquetes.

Generalizaciones

- Package (de Kernel)

Atributos

No tiene atributos adicionales.

Asociaciones

- javaClass: JavaClass [*]

  Referencia todas las clases que son miembros de este paquete. Subconjunto de *Package::ownedType*.

- javaInterface: JavaInterface [*]

  Referencia todas las interfaces que son miembros de este paquete. Subconjunto de *Package::ownedType*.

- /subpackage: Package [*]

  Referencia a los paquetes miembros de este paquete. Redefine *Package::nestedPackage*. Es derivado.

Restricciones

No hay restricciones adicionales.

Method

Declara un método que puede ser invocado pasando una cantidad fija de argumentos.

Generalizaciones

- JavaOperation

Atributos

- isAbstract: Boolean [1]

  Especifica si un método es abstracto. Es verdadero si no tiene implementación.

- isSyncronized: Boolean [1]

Especifica si un método es sincronizado. Es verdadero si adquiere un lock antes de su ejecución.

- isFinal: Boolean [1]

  Especifica si un método es final. Si es verdadero no puede ser sobrescrito en una clase derivada. Redefine *RedefinableElement::isLeaf*.

Asociaciones

- interface: JavaInterface [0..1]

  Declara a la interfaz que declara este método. Redefine *Operation::interface*.

Restricciones

No hay restricciones adicionales.

ANEXO D

Metamodelo Simplificado Específico a la Implementación C++

Sintaxis Abstracta

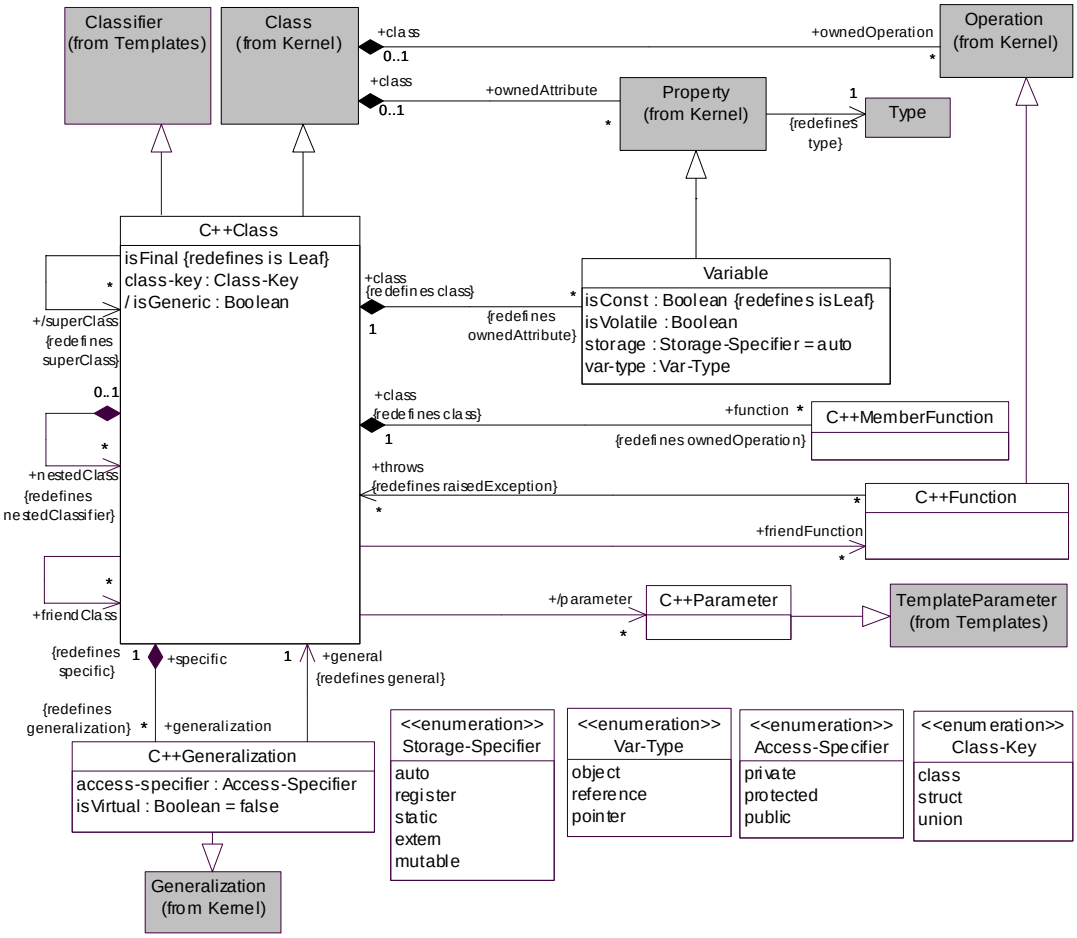


Diagrama de Clases C++

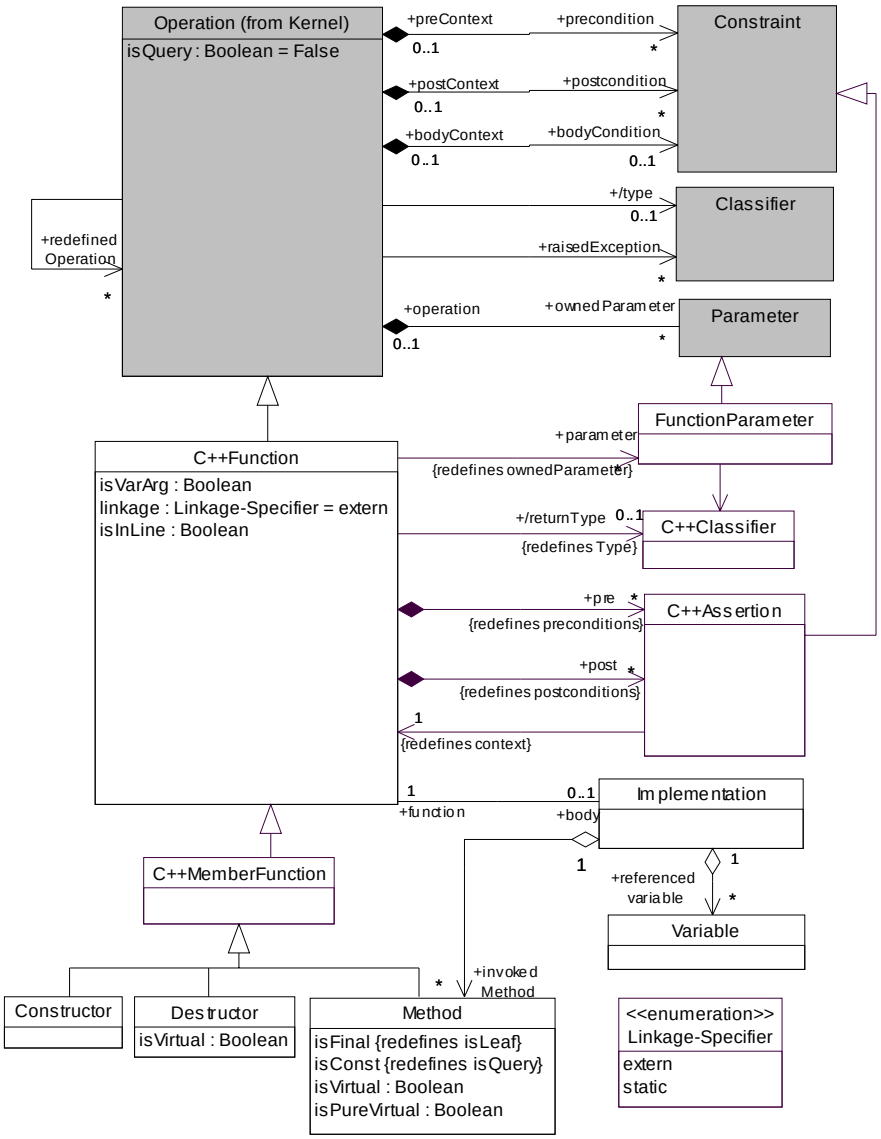


Diagrama de Funciones C++

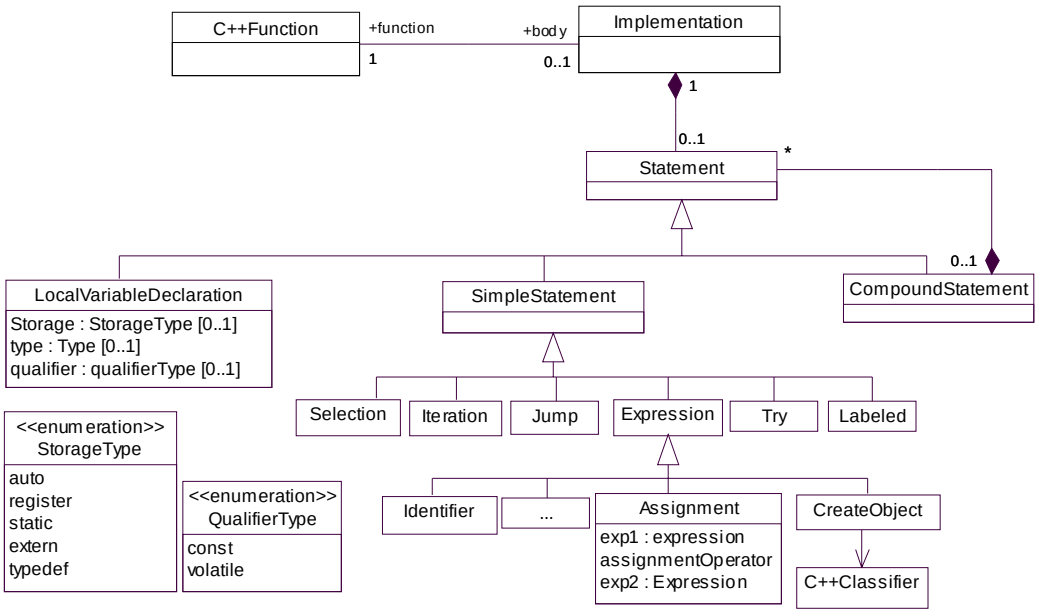


Diagrama de Implementación

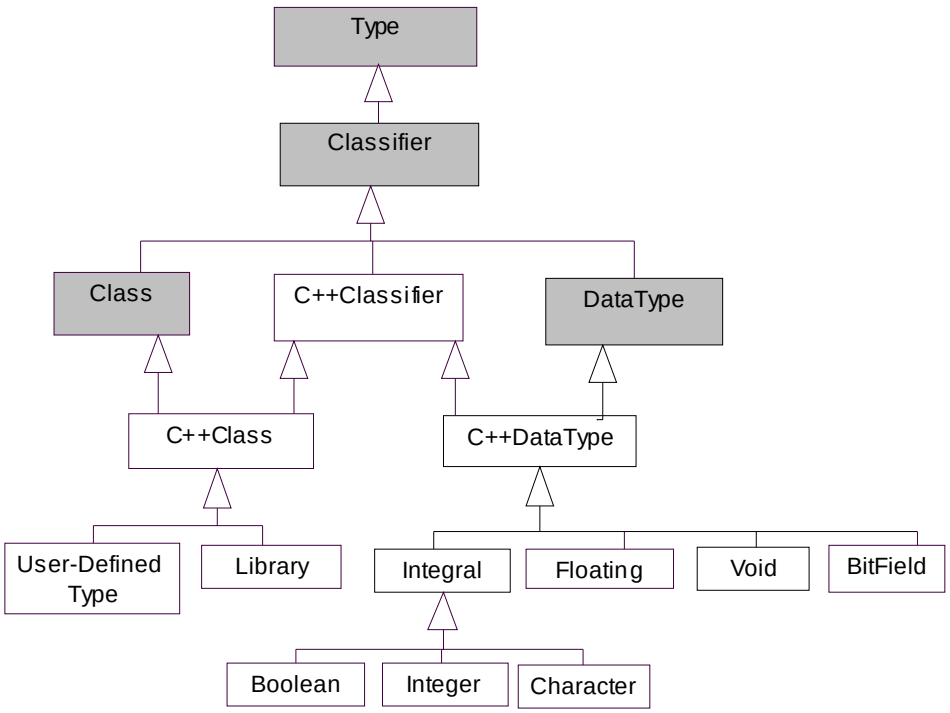


Diagrama de Clasificadores

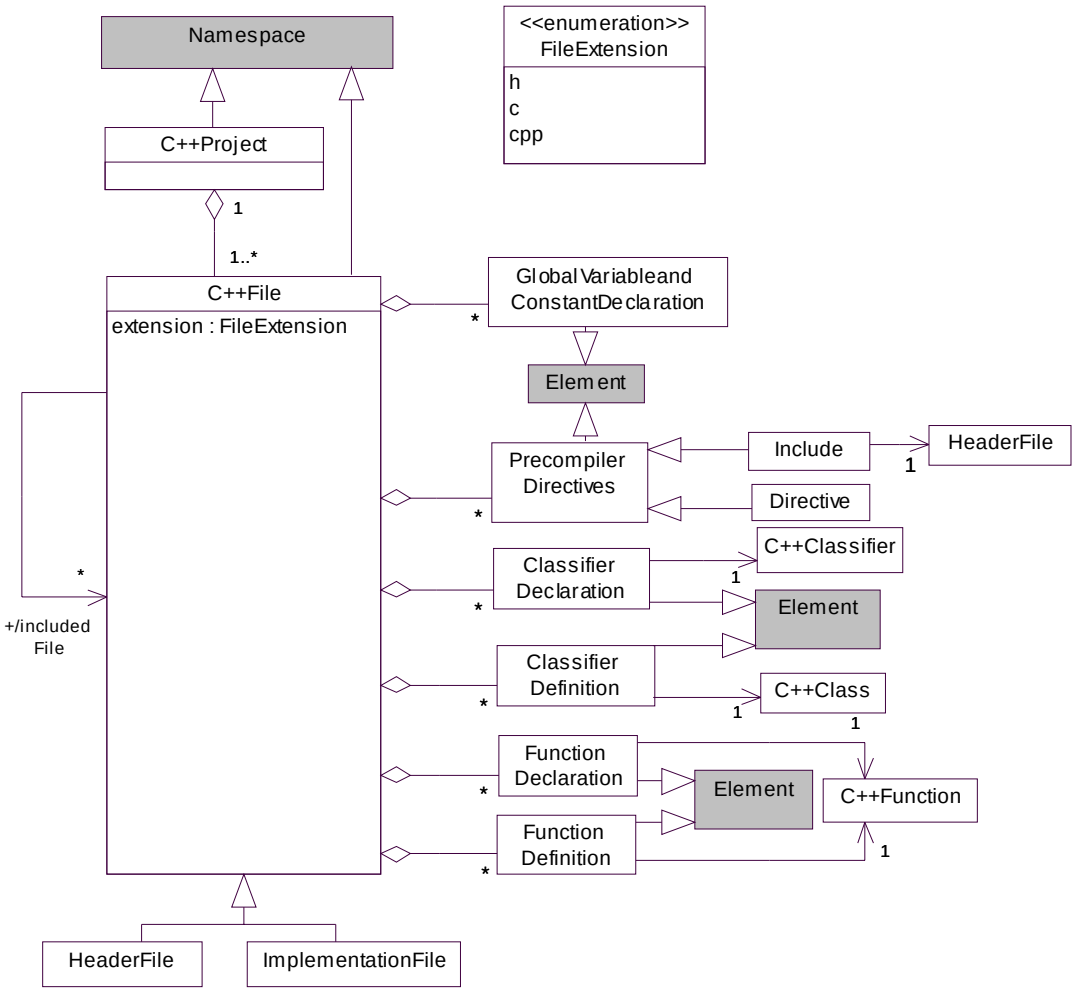


Diagrama de Proyecto

Descripciones de Clases

C++Class

Una clase C++ describe un conjunto de objetos que comparten las mismas especificaciones de features, restricciones y semántica.

Generalizaciones

- Class (de Kernel), Classifier (de Templates)

Atributos

- class-key: Class-Key [1]

  Especifica el tipo de clase, es decir, si es una clase, una estructura o una unión.

- isFinal: Boolean [1]

  Especifica si la clase puede tener subclases. Redefine *RedefinableElement::isLeaf*.

- /isGeneric: Boolean

  Especifica si la clase es genérica. Es un atributo derivado.

Asociaciones

- variable: Variable [*]

  Referencia a las variables propias de la clase C++. Redefine *Class::ownedAttribute*.

- nestedClass: C++Class [*]

  Referencia a las clases C++ que están declaradas dentro del cuerpo de una C++Class (clases anidadas). Subconjunto de *Class::nestedClassifier*.

- /superClass: C++Class [*]

  Referencia a las superclases de una clase C++. Redefine *Class::superClass*. Es derivado.

- function: C++MemberFunction [*]

  Referencia las funciones propias de la clase. Redefine *Class::ownedOperation*.

- generalization: C++Generalization [*]

  Referencia las relaciones de generalización que posee la clase. Redefine *Class::Generalization*.

- friendClass: C++Class [*]

  Referencia las clases amigas de la clase.

- friendFunction: C++Function [*]

  Referencia las funciones amigas de la clase.

- /parameters: C++Parameter [*]

  Referencia el conjunto de parámetros de la clase. Es derivado.

Restricciones

[1] Una clase que tiene alguna función virtual pura debe ser declarada abstracta.

```
self.function -> select (oclIsTypeOf(Method)) -> exists (m |
    m.oclAsType(Method).isPureVirtual)  implies self.isAbstract
```

[2] Una clase declarada final no puede tener subclases, es decir, ninguna clase en el paquete la tendrá como superclase.

```
self.isFinal  implies
    self.package.ownedMember -> select (oclIsTypeOf(C++Class)) -> forAll ( c |
        c.oclAsType(C++Class).superClass <> self )
```

[3] Las funciones privadas de una clase no pueden ser declaradas abstractas.

```
self.function -> select (oclIsTypeOf(Method)) -> forAll (m |
    m.visibility = #private implies
        not m.oclAsType(Method).isPureVirtual)
```

[4] Los métodos finales de una clase no pueden ser declarados abstractos.

```
self.function -> select (oclIsTypeOf(Method)) -> forAll (m |
    m.oclAsType(Method).isFinal implies not m.oclAsType(Method).isVirtual)
```

[5] Una clase es genérica si tiene una signatura template

```
isGeneric = ( self.ownedTemplateSignature -> size ( ) =1 )
```

[6] Parameters se deriva a partir de los parámetros de la signatura template redefinible.

```
/parameters= self.ownedTemplateSignature.parameter
```

[7] Las funciones amigas de una clase son funciones C++, pero no son funciones miembros de clase.

```
self.friendFunction -> forAll (f | f.isTypeOf(C++Function))
```

[8] Una clase sólo tiene una función destructor.

```
self.function -> select (oclIsTypeOf(Destructor)) ->size() <=1
```

C++File

Representa un archivo C++.

Generalizaciones

- Namespace (de Kernel)

Atributos

- extension: FileExtension [1]

  Especifica la extensión del archivo, es decir, h si es un archivo header, c o cpp si es un archivo de imlementación.

Asociaciones

- c++Project: C++Project [1]

Referencia al proyecto del cual es parte el archivo.

- /includedFile: C++File [*]

  Referencia el conjunto de archivos incluidos. Es derivado.

- globalVariableandConstantDeclaration:GlobalVariableandConstantDeclaration[*]

  Referencia al conjunto de variables y constantes globales declaradas.

- precompilerDirectives: PrecompilerDirectives [*]

  Referencia el conjunto de directivas al precompilador.

- classifierDeclaration: ClassifierDeclaration [*]

  Referencia el conjunto de declaraciones de clasificadores.

- classifierDefinition: ClassifierDefinition [*]

  Referencia el conjunto de definiciones de clasificadores.

- functionDeclaration: FunctionDeclaration [*]

  Referencia el conjunto de declaraciones de funciones.

- functionDefinition: FunctionDefinition [*]

  Referencia el conjunto de definiciones de funciones.

Restricciones

[1] Los archivos incluidos se derivan a través de los archivos incluidos por medio de las directivas *#include*.

```
/includedFile = self.precompilerDirective -> collect(oclIsTypeOf(Include).headerFile)
```

C++Function

Es una función C++.

Generalizaciones

- Operation (de Kernel)

Atributos

- isVarArg: Boolean

  Especifica si la función puede tener argumentos variables.

- linkage: Linkage-Specifier

  Especifica si la función es *extern*, indicando al compilador que la definición de la función se encuentra en otro archivo distinto del actual, o *static*, es decir, si su nombre no es visible fuera del archivo en el cual es declarado.

- isInLine: Boolean

  Si *isInLine* es verdadero significa que el compilador sustituirá la invocación a la función por el código de la misma.

Asociaciones

- parameter: FunctionParameter [*]

  Especifica los parámetros de la función. Redefine *Operation::ownedParameter*.

- pre: C++Assertion [*]

  Referencia las precondiciones de la función. Redefine *Operation::precondition*.

- post: C++Assertion [*]

  Referencia las postcondiciones de la función. Redefine *Operation::postcondition*.

- /returnType: C++Classifier [0..1]

  Especifica el tipo de retorno de la función. Redefine *Operation::type*. Es derivado.

- throws: C++Class [*]

  Referencia a los tipos que representan las excepciones que pueden surgir durante una invocación de esta operación. Redefine *Operation::raisedException*.

- body: Implementation [0..1]

  Referencia a la implementación de la función.

Restricciones

[1] Si una función es virtual pura no tiene implementación.

```
self.isPureVirtual  implies self.body -> isEmpty()
```

C++Generalization

Representa una relación de generalización en C++.

Generalizaciones

- Generalization (de Kernel)

Atributos

- access-specifier: Access-Specifier

  Especifica el tipo de acceso a los miembros de la clase base.

- isVirtual:Boolean

  Especifica si la herencia es virtual.

Asociaciones

- general: C++Class [1]

  Referencia a la clase más general en la relación de generalización. Redefine *Generalization::general*.

- specific: C++Class [1]

  Referencia a la clase más específica en la relación de generalización. Redefine *Generalization::specific*.

Restricciones

No posee restricciones adicionales.

C++MemberFunction

Es una función que es miembro de una clase C++.

Generalizaciones

- C++Function

Atributos

No posee atributos adicionales.

Asociaciones

- class: C++Class [1]

  Referencia la clase que posee la función. Redefine *Operation::class*.

Restricciones

[1] Una función miembro de una clase no puede ser declarada extern.

self.linkage <> “extern”

C++Project

Representa un proyecto C++.

Generalizaciones

- Namespace (de Kernel)

Atributos

No posee atributos adicionales.

Asociaciones

- c++File: C++File [1.. *]

  Referencia el conjunto de archivos C++ que posee el proyecto.

Restricciones

No posee restricciones adicionales.

ClassifierDeclaration

Representa las declaraciones de clasificadores.

Generalizaciones

- Element (de Kernel)

Atributos

No posee atributos adicionales.

Asociaciones

- c++Classifier: C++Classifier [1]

Referencia al clasificador que declara.

Restricciones

No posee restricciones adicionales.

ClassifierDefinition

Representa las definiciones de clasificadores.

Generalizaciones

- Element (de Kernel)

Atributos

No posee atributos adicionales.

Asociaciones

- c++Class: C++Class [1]

  Referencia a la clase que define.

Restricciones

No posee restricciones adicionales.

Constructor

Designa una función usada para crear instancias de una clase. No pueden ser invocados explícitamente mediante expresiones de invocación a métodos. Los constructores no poseen tipo de retorno y tienen el mismo nombre de la clase que contiene la declaración del constructor. Las declaraciones de constructores no son heredadas.

Generalizaciones

- C++MemberFunction

Atributos

No tiene atributos adicionales.

Asociaciones

No tiene asociaciones adicionales.

Restricciones

[1] Un constructor no tiene tipo de retorno

```
self.type-> isEmpty()
```

[2] El nombre de un constructor es el mismo nombre de la clase que contiene la declaración.

```
self.name = self.class.name
```

Destructor

Un destructor es una función miembro con igual nombre que la clase, pero precedido por el carácter ~. Una clase sólo tiene una función destructor que, no tiene argumentos y no devuelve ningún tipo. Un destructor realiza la operación opuesta de un constructor, limpiando el almacenamiento asignado a los objetos cuando se crean.

Generalizaciones

- C++MemberFunction

Atributos

- isVirtual: Boolean [1]

  Especifica si el destructor es virtual, es decir, si puede ser redefinido en las subclases.

Asociaciones

No tiene asociaciones adicionales.

Restricciones

[2] Un destructor no tiene argumentos ni tipo de retorno

```
self.ownedParameter -> isEmpty() and self.type -> isEmpty()
```

[3] El nombre de un destructor es el mismo nombre de la clase que contiene la declaración precedido por el carácter ~.

```
self.name = “~”.concat(self.class.name)
```

FunctionDeclaration

Representa las declaraciones de funciones.

Generalizaciones

- Element (de Kernel)

Atributos

No posee atributos adicionales.

Asociaciones

- c++Function: C++Function [1]

  Referencia a la función que declara.

Restricciones

No posee restricciones adicionales.

FunctionDefinition

Representa las definiciones de funciones.

Generalizaciones

- Element (de Kernel)

Atributos

No posee atributos adicionales.

Asociaciones

- c++Function: C++Function [1]

  Referencia a la función que define.

Restricciones

No posee restricciones adicionales.

GlobalVariableandConstantDeclaration

Representa a las variables y constantes globales declaradas en un archivo C++ .

Generalizaciones

- Element (de Kernel)

Atributos

No posee atributos adicionales.

Asociaciones

No posee asociaciones adicionales.

Restricciones

No posee restricciones adicionales.

HeaderFile

Representa un archivo header C++.

Generalizaciones

- C++File

Atributos

No posee atributos adicionales.

Asociaciones

No posee asociaciones adicionales.

Restricciones

No posee restricciones adicionales.

Implementation

Especifica un procedimiento que lleva a cabo el resultado de la función.

Generalizaciones

- Element (de Kernel)

Atributos

- Procedure: Expression [0..1]

  Referencia el procedimiento de la función.

Asociaciones

- function: C++Function [1]

  Referencia a función a la que pertenece.
- invokedMethod: Method [*]

  Referencia a los métodos invocados en el cuerpo de una función.
- referencedVariable: Variable [*]

  Referencia a las variables referenciadas en el cuerpo de una función.
- statement: Statement [0..1]

  Referencia al bloque de sentencias del cuerpo de la función.

Restricciones

No hay restricciones adicionales.

ImplementationFile

Representa un archivo de implementación C++.

Generalizaciones

- C++File

Atributos

No posee atributos adicionales.

Asociaciones

No posee asociaciones adicionales.

Restricciones

No posee restricciones adicionales.

Include

Representa a las directivas al precompilador del tipo *#include*.

Generalizaciones

- PrecompilerDirective

Atributos

No posee atributos adicionales.

Asociaciones

- headerFile: HeaderFile [1]

  Referencia al archivo header que incluye a través de la directiva.

Restricciones

No posee restricciones adicionales.

Method

Declara una función miembro de una clase que puede ser invocada pasando una cantidad fija de argumentos.

Generalizaciones

- C++MemberFunction

Atributos

- isConst: Boolean [1]

  Especifica si un método es constante. Redefine *Operation::isQuery*.

- isFinal: Boolean [1]

  Especifica si un método es final, es decir, si no puede ser redefinido en las subclases. Redefine *RedefinableElement::isLeaf.*

- isVirtual: Boolean [1]

  Especifica si un método es virtual, es decir, si puede ser redefinido en las subclases.

- isPureVirtual: Boolean [1]

  Especifica si un método es puro, es decir, si no tiene implementación.

Asociaciones

No hay asociaciones adicionales.

Restricciones

No hay restricciones adicionales.

PrecompilerDirective

Representa a las directivas al precompilador.

Generalizaciones

- Element (de Kernel)

Atributos

No posee atributos adicionales.

Asociaciones

No posee asociaciones adicionales.

Restricciones

No posee restricciones adicionales.

Statement

Representa el bloque de código que implementa una función.

Generalizaciones

- Action (de Action)

Atributos

No hay atributos adicionales.

Asociaciones

- implementation: Implementation [1]

  Referencia a la implementación de la cual ésta es parte.

- compoundStatement: CompoundStatement [0..1]

  Referencia al bloque de sentencias de la cual ésta es parte.

Restricciones

No hay restricciones adicionales.

Variable

Representa a las variables declaradas en la clase.

Generalizaciones

- Property (de Kernel)

Atributos

- isConst: Boolean [1]

  Especifica si una variable es constante. Si es final debe tener valor inicial obligatoriamente. Redefine *RedefinableElement::isLeaf.*

- isVolatile: Boolean [1]

  Especifica si una variable es volátil, es decir, si es accedido de forma asincrónica.

- storage: Storage-Specifier [1]

  Especifica el tipo de almacenamiento de la variable.

- var-type: Var-Type [1]

  Especifica el tipo de la variable, es decir, si la variable es un objeto, es una referencia o un puntero.

Asociaciones

- class: C++Class [1]

  Referencia a la clase que declara esta variable. Redefine *Property::class*.

Restricciones

[1] Una variable es una propiedad que es parte de una clase y no es miembro de ninguna asociación.

```
self.class -> size() = 1 and self.association -> isEmpty() and
self.opposite -> isEmpty()
```

ANEXO E

Metamodelo Simplificado Específico a la Implementación Java

Sintaxis Abstracta

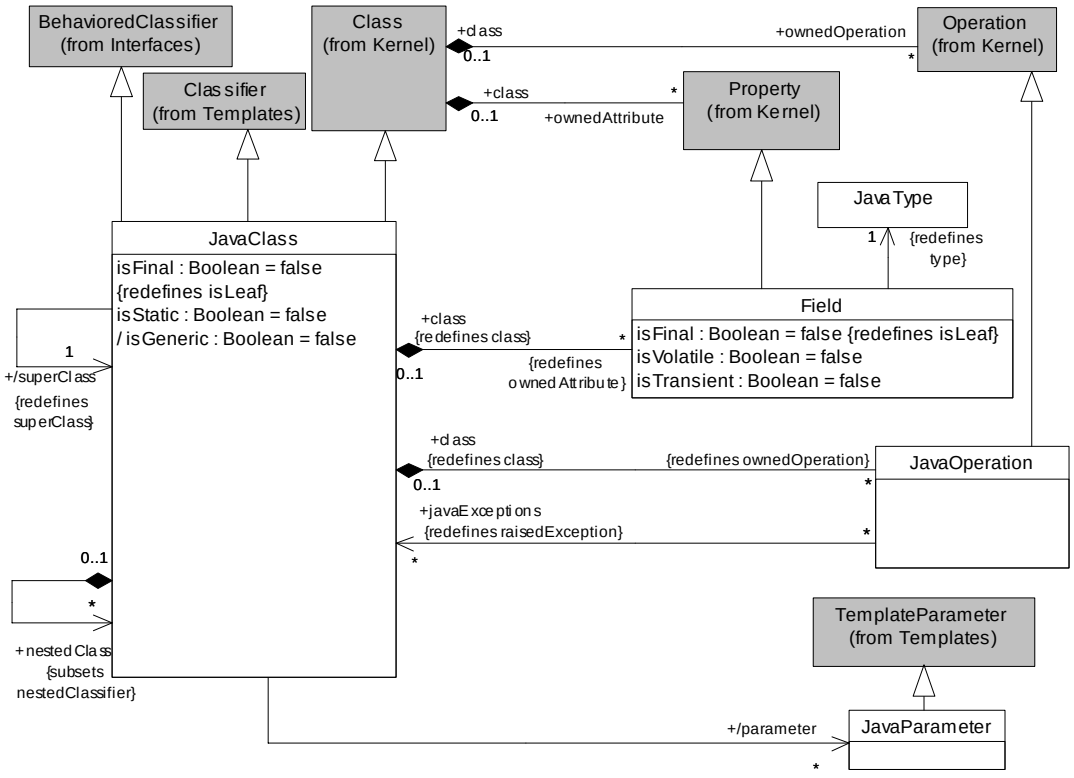


Diagrama de Clases

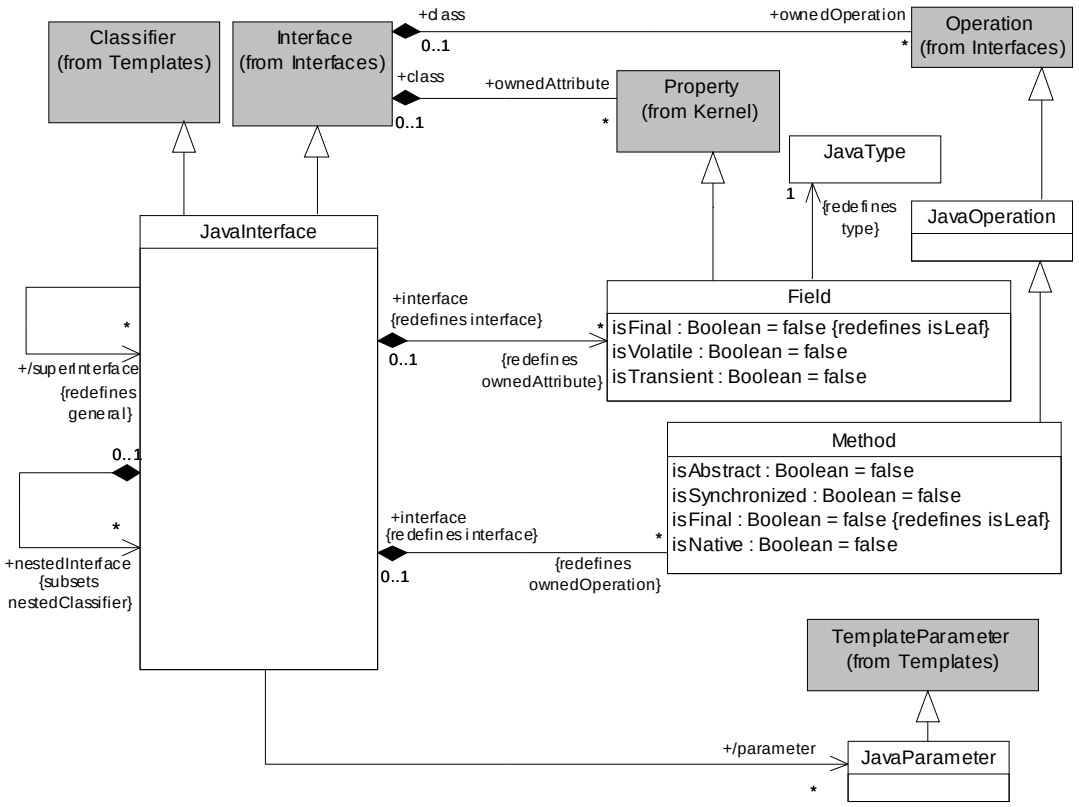


Diagrama de Interfaces

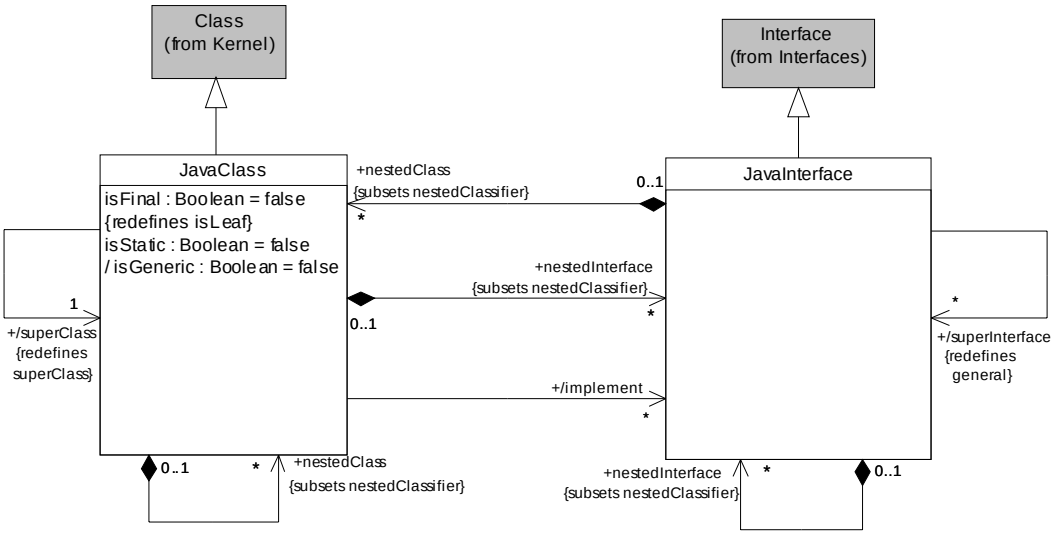


Diagrama de relaciones entre clases e interfaces Java

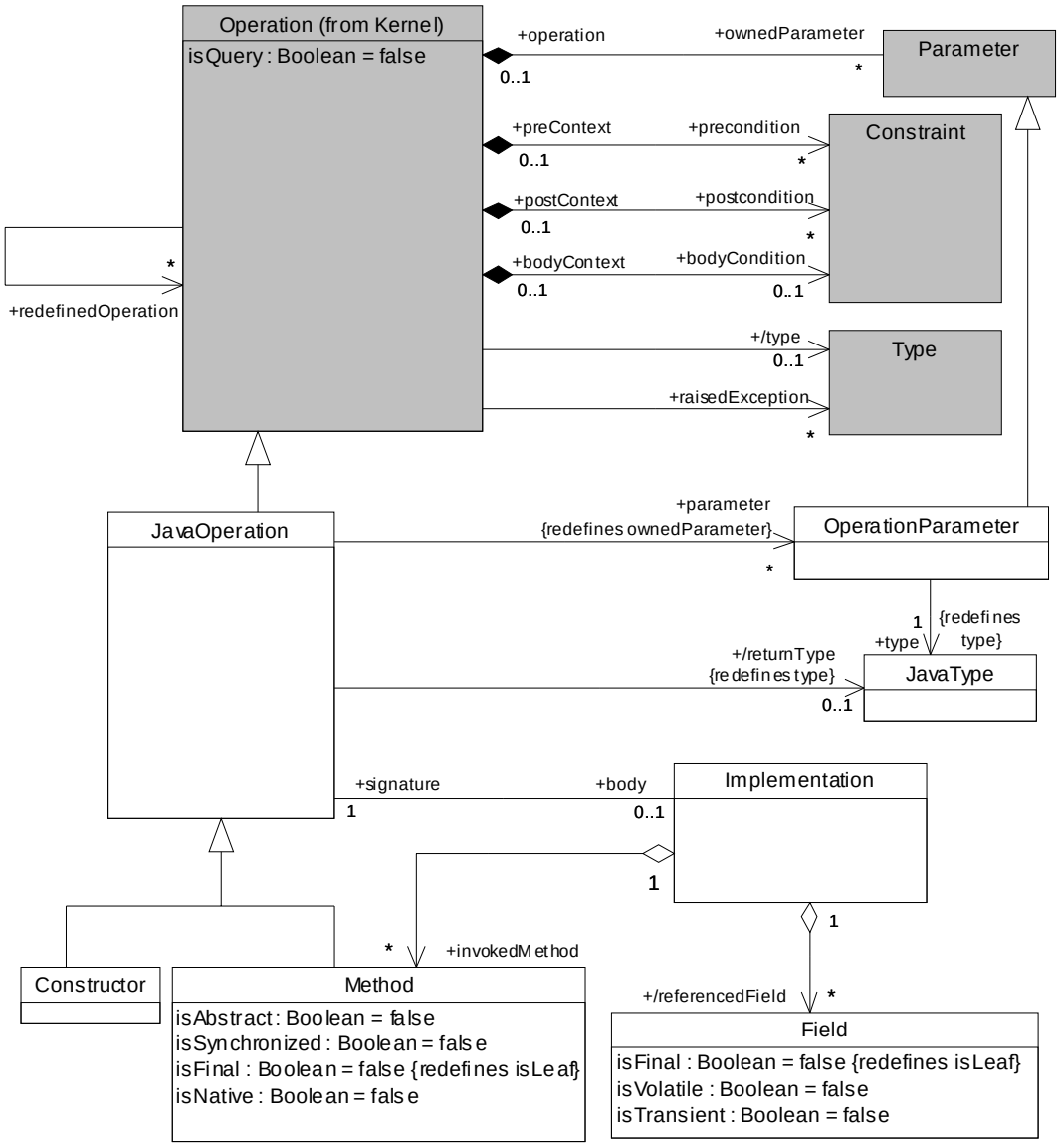


Diagrama de operaciones

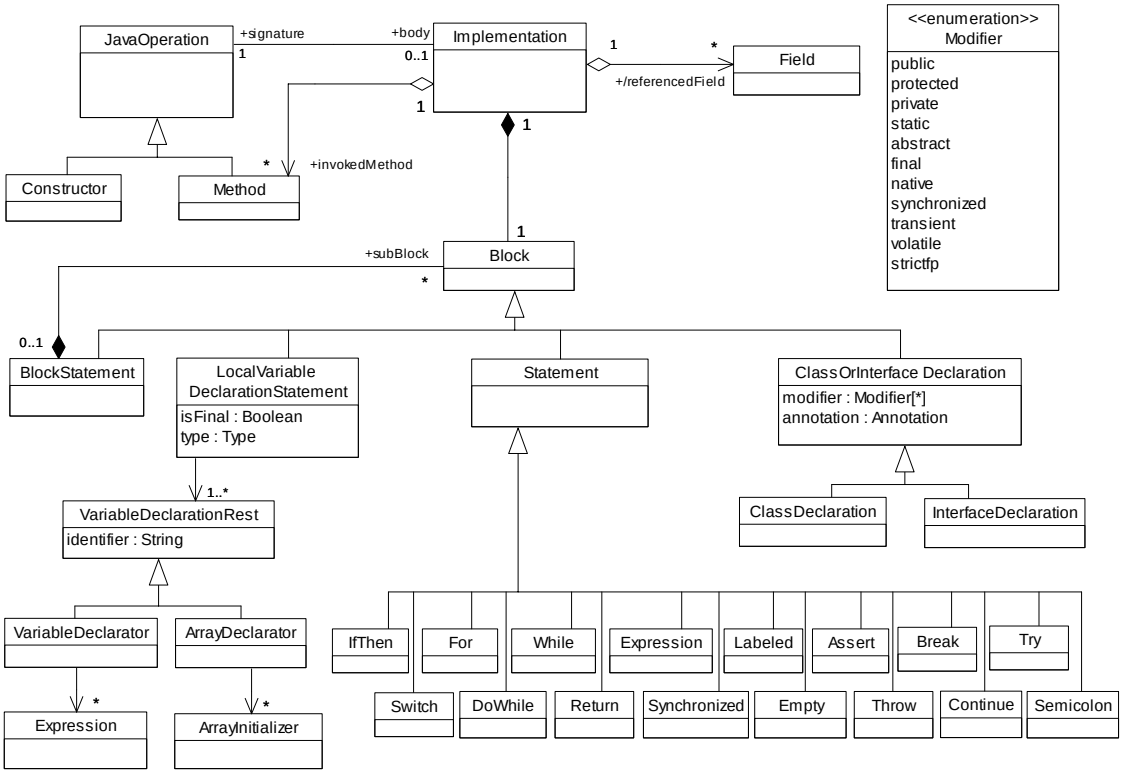


Diagrama de Implementación

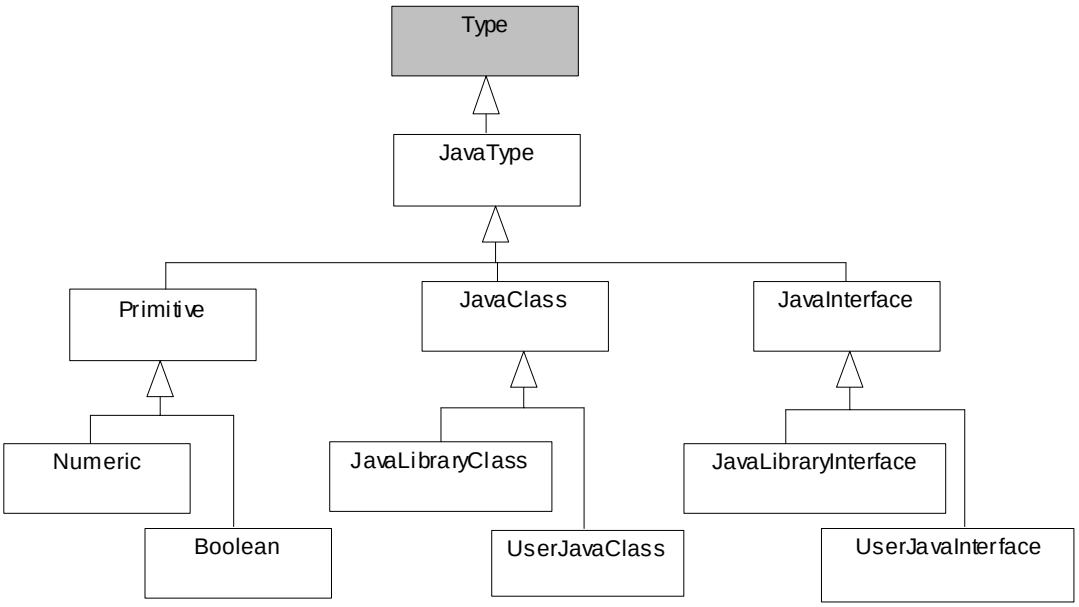


Diagrama de Tipos

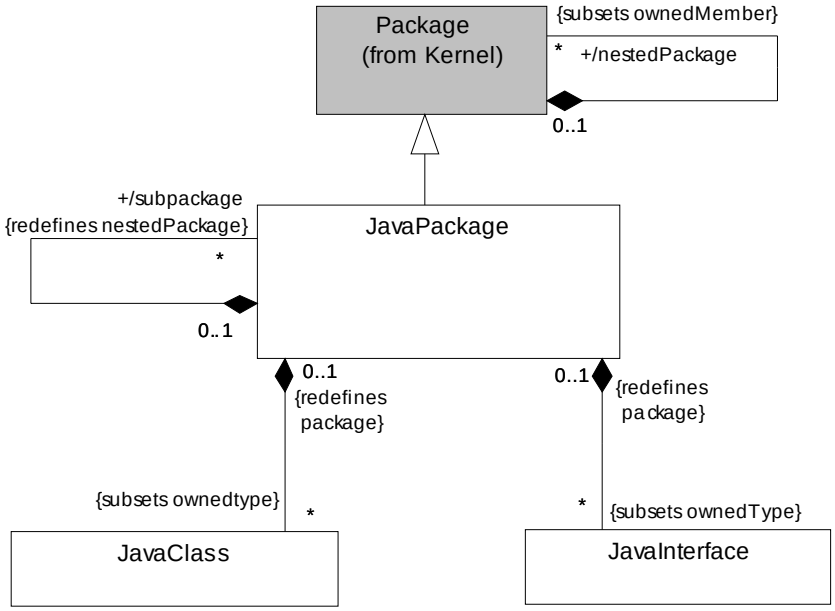


Diagrama de Paquetes

Descripción de las clases

A continuación se describen las principales metaclases del metamodelo a nivel de código Java. Se extiende el metamodelo PSM Java teniendo en cuenta la especificación del mismo (Especificación Java 3º Edición).

Block

Especifica el bloque de código que implementa una operación, tal como está definido en la Especificación del Lenguaje Java.

Generalizaciones

- Action (de Action)

Atributos

No tiene atributos adicionales

Asociaciones

- implementation: Implementation [1]

  Referencia la implementación de la cual ésta es parte.

- blockStatement: blockStatement [0..1]

  Referencia el bloque de sentencias de la cual ésta es parte.

Restricciones

No tiene restricciones adicionales.

Constructor

Un constructor, como está definido en la Especificación del Lenguaje Java.

Generalizaciones

- JavaOperation

Atributos

No tiene atributos adicionales.

Asociaciones

No tiene asociaciones adicionales.

Restricciones

[1] Un constructor no tiene tipo de retorno

```
self.returnType->isEmpty()
```

[2] El nombre de un constructor es el mismo nombre de la clase que contiene la declaración.

```
self.name = self.class.name
```

Field

Representa un atributo, como está definido en la Especificación del Lenguaje Java.

Generalizaciones

- Property (de Kernel)

Atributos

- isFinal: Boolean [1]

  Especifica si un atributo es final, es decir, si es una constante. Si es final debe tener un valor inicial obligatoriamente. Redefine *RedefinableElement::isLeaf*.

- isVolatile: Boolean [1]

  Especifica si un atributo es volátil, es decir, si es accedido de forma asincrónica.

- isTransient: Boolean [1]

  Especifica si un atributo es parte del estado persistente del objeto.

Asociaciones

- class: JavaClass [0..1]

  Referencia a la clase que declara este atributo. Redefine *Property::class*.

- javaType: JavaType [1]

  Referencia al tipo del atributo. Redefine *TypedElemnt::type*.

Restricciones

[1] Un atributo es una propiedad que es parte de una clase y no es miembro de ninguna asociación.

```
self.class->size() = 1 and self.association->isEmpty() and self.opposite->isEmpty()
```

Implementation

Especifica un procedimiento que lleva a cabo el resultado de la operación.

Generalizaciones

- Element (de Kernel)

Atributos

No tiene atributos adicionales

Asociaciones

- block: Block [1]

  Especifica el bloque de código de la implementación.

- signature: JavaOperation [1]

  Especifica la operación que ésta implementa.

- invokedMethod: Method [*]

  Referencia a los métodos invocados en el cuerpo de una operación.

- referencedField: Field [*]

  Referencia a las variables referenciadas en el cuerpo de una operación.

Restricciones

No hay restricciones adicionales.

JavaClass

Una clase Java, como está definida en la Especificación del Lenguaje Java.

Generalizaciones

- Class (de Kernel), Classifier (de Templates), BehavoiredClassifier (de Interfaces)

Atributos

- isFinal: Boolean

  Especifica si la clase puede tener subclases. Redefine *RedefinableElement::isLeaf*.

- isStatic: Boolean

  Especifica si la clase es estática.

- /isGeneric: Boolean

  Especifica si la clase es genérica. Es un atributo derivado.

Asociaciones

- field: Field [*]

  Referencia a las variables propias de la clase Java. Redefine *Class::ownedAttribute*.

- nestedClass: JavaClass [*]

  Referencia a las clases Java que están declaradas dentro del cuerpo de una JavaClass (clases anidadas).Subconjunto de *Class::nestedClassifier*

- nestedInterface: JavaInterface [*]

  Referencia a las interfaces Java que están declaradas dentro del cuerpo de una JavaClass (interfaces anidadas).Subconjunto de *Class::nestedClassifier*.

- /implement: JavaInterface [*]

  Referencia a las interfaces Java implementadas por esta clase. Es derivado.

- /superClass: JavaClass [1]

  Referencia a la superclase de una clase Java. Redefine *Class::superClass*. Es derivado.

- javaOperation: JavaOperation [*]

  Referencia las operaciones propias de la clase. Redefine *Class::ownedOperation*.

- javaPackage: JavaPackage [0..1]

  Referencia el paquete en el cual está declarada. Redefine *Type::package*.

- /parameters: JavaParametes [*]

  Referencia el conjunto de parámetros de una clase. Es derivado.

Restricciones

[1] Los clasificadores anidados de una clase o de una interfaz Java sólo pueden ser del tipo JavaClass o JavaInterface.

```
self.nestedClassifier->forAll(c |
   c.oclIsTypeOf(JavaClass) or c.oclIsTypeOf(JavaInterface))
```

[2] Las interfaces implementadas son aquellas referenciadas a través de la relación de realización de interfaz.

```
implement = self.interfaceRealization.contract
```

[3] Una clase que tiene algún método abstracto debe ser declarada abstracta.

```
self.javaOperation->select(op| op.oclIsTypeOf(Method) ) -> exists (m |
   m.oclAsType(Method).isAbstract)  implies self.isAbstract
```

[4] Una clase declarada abstracta no tiene constructor definido explícitamente

```
self.isAbstract implies
   self.javaOperation->select(op| op.oclIsTypeOf (Constructor) ) -> isEmpty()
```

[5] Una clase declarada final no puede tener subclases, es decir, ninguna clase en el paquete la tendrá como superclase.

```
self.isFinal  implies
    self.javaPackage.ownedMember ->select(m|
      m.oclIsTypeOf(JavaClass)) ->forAll (c| c.oclAsType(JavaClass).superClass <> self)
```

[6] Los modificadores protegido, privado o estático sólo pueden ser aplicados a las clases anidadas, es decir, a las declaradas dentro de la declaración de otra clase.

```
(self.visibility = #protected or self.visibility = #private or self.isStatic)   implies
    self.javaPackage.ownedMember->select(m| m.oclIsTypeOf(JavaClass)) ->
      exists ( c | c.oclAsType(JavaClass).nestedClass -> includes(self) )
```

[7] Los métodos privados de una clase no pueden ser declarados abstractos.

```
self.javaOperation->select(op| op.oclIsTypeOf(Method) ) ->
    forAll (m | m.visibility = #private implies not m.oclAsType(Method).isAbstract)
```

[8] Los métodos estáticos de una clase no pueden ser declarados abstractos.

```
self.javaOperation->select(op| op.oclIsTypeOf(Method) ) ->
    forAll (m | m.isStatic implies not m.oclAsType(Method).isAbstract)
```

[9] Los métodos finales de una clase no pueden ser declarados abstractos.

```
self.javaOperation->select(op| op.oclIsTypeOf(Method) ) ->
    forAll (m | m.oclAsType(Method).isFinal implies not m.oclAsType(Method).isAbstract)
```

[10] Una clase es genérica si tiene una signatura template

```
isGeneric = ( self.ownedTemplateSignature -> size ( ) =1 )
```

[11] Parameters se deriva a partir de los parámetros de la signatura template.

```
/parameters= self.ownedTemplateSignature.parameter
```

[12] Una clase es concreta, si todos sus métodos tienen una implementación asociada.

```
not self.isAbstract implies self.allMethod()-> forAll (m |
    self.allBody()-> exist (b|b.signature=m) )
```

[13] Los elementos que pueden ser parámetros actuales de un parámetro formal son tipos Java.

```
self.parameters.parameteredElement -> forAll (p | p.oclIsTypeOf (JavaType)  )
```

Operaciones adicionales

[1] allMethod es el conjunto de todos los métodos de una clase, es decir, los métodos propios, heredados y los métodos de las interfaces que implementa.

```
allMethod(): Set(Method)
allMethod ()= self.allClassMethod()-> union(self.implement.allInterfaceMethod())

allClassMethod(): Set(Method)
allClassMethod()=  self.javaOperation->select(o |o.oclIsType(Method)) ->
                        union(self.superClass.allClassMethod())
```

allInterfaceMethod (): Set(Method)

allInterfaceMethod()= self.method -> union(self.superInterface.allInterfaceMethod())

[2] allBody es el conjunto de todas las implementaciones de los métodos de una clase, es decir, tantos propios, como heredados.

allBody(): Set(Implementation)

allBody = self.allMethod().body

JavaInterface

Describe las características de una interfaz según la Especificación del Lenguaje Java.

Generalizaciones

- Interface (de Interfaces), Classifier (de Templates).

Atributos

No tiene atributos adicionales.

Asociaciones

- nestedClass: JavaClass [*]

  Referencia todas las clases que son declaradas dentro del cuerpo de una JavaInterface (clases anidadas). Subconjunto de *Interface::nestedClassifer*.

- nestedInterface: JavaInterface [*]

  Referencia todas las interfaces declaradas dentro del cuerpo de una JavaInterface (interfaces anidadas). Subconjunto de *Interface::nestedClassifer*.

- /superInterface: JavaInterface [*]

  Referencia las superinterfaces de una JavaInterface. Es derivada. Redefine *Classifier::general*.

- field: Field [*]

  Referencia los campos propios de una JavaInterface. Redefine *Interface::ownedAttribute*.

- method: Method [*]

  Referencia los métodos propios de una JavaInterface. Redefine *Interface::ownedOperation.*

- javaPackage: JavaPackage [0..1]

  Referencia el paquete en el cual está declarada. Subconjunto de *Type::package*.

- /parameter: JavaParameter [1]

  Referencia el conjunto de parámetros de una interfaz. Es derivado.

Restricciones

[1] Las interfaces en Java son implícitamente abstractas.

self.isAbstract

[2] Los miembros propios de una interfaz son implícitamente públicos.

```
self.ownerMember->forAll (m| m.visibility = #public)
```

[3] Los clasificadores anidados de una interfaz sólo pueden ser del tipo JavaClass o JavaInterface.

```
self.nestedClassifier->forAll(c |
   c.oclIsTypeOf(JavaClass) or c.oclIsTypeOf(JavaInterface))
```

[4] Una interfaz sólo puede ser declarada privada o protegida si está directamente anidada en la declaración de una clase.

```
(self.visibility= #protected or self.visibility = #private) implies
   self.package.ownedMember->select(m | m.oclIsTypeOf(JavaClass)) ->
    exists(c | c.oclAsType(JavaClass).nestedInterface->includes (self) )
```

[5] Una interfaz sólo puede ser declarada estática si está directamente anidada en la declaración de una clase o interfaz.

```
self.isStatic  implies
   self.package.ownedMember->select(m | m.oclIsTypeOf(JavaClass) ) ->
      exists(c | c.oclAsType(JavaClass).nestedInterface->includes (self) ) or
         self.package.ownedMember->select(m | m.oclIsTypeOf (JavaInterface) ) ->
            exists(i | i.oclAsType(JavaInterface).nestedInterface->includes (self) )
```

[6] Los métodos declarados en una interfaz son abstractos por lo tanto no tienen implementación.

```
self.method->forAll (m| m.isAbstract and m.body->isEmpty() )
```

[7] Los métodos de una interfaz no pueden ser declarados estáticos.

```
self.method->forAll (m| not m.isStatic)
```

[8] Los métodos de una interfaz no pueden ser sincronizados.

```
self.method->forAll (m| not m.isSynchronized)
```

[9] Los campos de una interfaz son implícitamente públicos, estáticos y finales.

```
self.field->forAll (f | f.visibility = #public and f.isStatic and f.siFinal)
```

[10] superInterface se deriva de la relación de generalización.

```
/superInterface = self.generalization.general
```

[11] parameters se deriva a partir de los parámetros de la signatura template.

```
/parameters= self.ownedTemplateSignature.parameter
```

[12] Los elementos que pueden ser parámetros actuales de un parámetro formal son tipos Java.

```
self.parameters.parameteredElement -> forAll (p | p.oclIsTypeOf (JavaType)  )
```

JavaOperation

Describe una operación según la Especificación del Lenguaje Java.

Generalizaciones

- Operation (de Kernel)

Asociaciones

- class: JavaClass [0..1]

  Referencia a la clase que declara esta operación. Redefine *Operation::class*.

- javaExceptions: JavaClass [*]

  Referencia a los tipos que representan las excepciones que pueden surgir durante una invocación de esta operación. Redefine *Operation::raisedException*.

- body: Implementation [0..1]

  Referencia a la implementación de la operación.

- parameter: OperationParameter [*]

  Especifica los parámetros de la operación. Redefine *Operation::ownedParameter*

- /returnType: JavaType [0..1]

  Especifica el tipo de retorno de la operación. Redefine *Operation::type*. Es derivado.

Restricciones

[1] Si una operación es abstracta no tiene implementación

```
self.isAbstract implies self.body->isEmpty()
```

JavaPackage

Es un paquete Java, como está definido en la Especificación del Lenguaje Java.

Generalizaciones

- Package (de Kernel)

Atributos

No tiene atributos adicionales.

Asociaciones

- javaClass: JavaClass [*]

  Referencia todas las clases que son miembros de este paquete. Subconjunto de Package::ownedType.

- javaInterface: JavaInterface [*]

  Referencia todas las interfaces que son miembros de este paquete. Subconjunto de Package::ownedType.

- /subpackage: JavaPackage [*]

  Referencia a los paquetes miembros de este paquete. Redefine Package::nestedPackage. Es derivado.

Restricciones

[1] Los miembros de un paquete sólo pueden ser clases, interfaces o subpaquetes.

```
self.ownedMember -> forAll (m | m.oclIsTypeOf ( JavaInterface) or
        m.oclIsTypeOf ( JavaClass) or m.oclIsTypeOf ( JavaPackage) )
```

Method

Describe un método según está definido en la Especificación del Lenguaje Java.

Generalizaciones

- JavaOperation

Atributos

- isAbstract: Boolean [1]

  Especifica si un método es abstracto. Es verdadero si no tiene implementación.

- isSyncronized: Boolean [1]

  Especifica si un método es sincronizado. Es verdadero si adquiere un lock antes de su ejecución.

- isFinal: Boolean [1]

  Especifica si un método es final. Si es verdadero no puede ser sobrescrito en una clase derivada. Redefine *RedefinableElement::isLeaf*.

- isNative: Boolean [1]

  Especifica si un método es Nativo.

Asociaciones

- interface: JavaInterface [0..1]

  Declara a la interfaz que declara este método. Redefine *Operation::interface*.

Restricciones

[1] Si un método es nativo no puede ser abstracto.

```
self.isNative implies not self.isAbstract
```

[2] Si un método tiene un tipo de retorno entonces debe tener una sentencia return.

```
self.type->size() = 1 implies
   self.body.block.oclIsTypeOf(Return) or
   self.body.block.oclIsKindOf(BlockStatement) and
   self.body.block.allStatement() -> exists (sent | sent.oclIsTypeOf(Return) )
```

Operaciones adicionales

[1] allStatement es el conjunto de todas las sentencias que conforman el cuerpo de un método.

```
allStatement(): Set(Statement)
allStatement()= self.subBlock->union(self.subBlock.allStatement())
```

OperationParameter

Especifica los parámetros de una operación según está definido en la Especificación del Lenguaje Java.

Generalizaciones

- Parameter (de Kernel)

Atributos

No tiene atributos adicionales.

Asociaciones

- type: JavaType [1]

  Referencia el tipo del parámetro. Redefine *TypedElement::type*.

Restricciones

No tiene restricciones adicionales.

ANEXO F

Formalización en NEREUS del Metamodelo Simplificado UML

CLASS Association
IS-SUBTYPE-OF Relationship, Classifier
ASSOCIATES <<Association-Type>>, <<Association-Property$_0$>>, <<Association-Property$_1$>>, <<Association-Property$_2$>>
GENERATED_BY create
TYPES Association
FUNCTIONS
create: ∗ x Boolean x _→ Association
get_isDerived: Association → Boolean
AXIOMS b: Boolean
get_isDerived (create(∗, b, _)) = b …
END-CLASS

CLASS BehavioralFeature
IS-SUBTYPE-OF Feature, Namespace
ASSOCIATES <<BehavioralFeature-Parameter >>, <<BehavioralFeature-Type>>
GENERATED_BY create
DEFERRED
TYPES BehavioralFeature
FUNCTIONS
create: ∗ x _ → BehavioralFeature …
END-CLASS

CLASS Class
IS-SUBTYPE-OF Classifier
ASSOCIATES <<Class-Property>>, <<Class-Class>>, <<Class-Classifier>>, <<Class-Operation>>
GENERATED_BY create
TYPES Class
FUNCTIONS
create: * x _ → Class
allDescendant: Class → Set(Class) …
END-CLASS

CLASS Classifier
IS-SUBTYPE-OF Namespace, RedefinableElement, Type
ASSOCIATES << Classifier-Generalization$_0$>>, <<Classifier-NamedElement>>,

<<Classifier-Property>>, <<Classifier-Classifier>>, <<Classifier-Feature>>, <<Classifier-Package>>
GENERATED_BY create
DEFFERRED
TYPES Classifier
FUNCTIONS
create: ∗ x Boolean x _ → Classifier
get_isAbstract: Classifier → Boolean
AXIOMS b: Boolean
get_isAbstract (create(∗, b, _)) = b …
END-CLASS

CLASS Element
ASSOCIATES <<Element-Element$_0$>>, <<Element-Element$_1$>>, <<Element-Comment>>
DEFERRED
TYPES Element
END-CLASS

CLASS Feature
IS-SUBTYPE-OF RedefinableElement
ASSOCIATES <<Classifier-Feature>>
GENERATED_BY create
DEFERRED
TYPES Feature
create: ∗ x Boolean x _ → Feature
get_isStatic: Feature → Boolean
AXIOMS b:Boolean
get_isStatic(create(∗, b, _)) = b
END-CLASS

CLASS Generalization
IS-SUBTYPE-OF DirectedRelationship
ASSOCIATES <<Classifier-Generalization$_0$>>, <<Classifier-Generalization$_1$>>
GENERATED_BY create
TYPES Generalization
FUNCTIONS
create: Boolean x _→ Generalization
get_isSubstitutable: Generalization → Boolean
AXIOMS b: Boolean
get_isSubstitutable(create(b,_)) = b
END-CLASS

CLASS Interface
IS-SUBTYPE-OF Classifier
ASSOCIATES <<Interface-Property>>, <<Interface-Interface>>, <<Classifier-Interface>>, <<Interface-Operation>>
GENERATED_BY create
TYPES Interface
FUNCTIONS
create: ∗ x _ → Interface
AXIOMS a:Classifier-Feature; i: Interface; p:Property
forAll$_f$ (get_feature(a, i), [equal(get_visibility(f), 'public')])…
END-CLASS

CLASS InterfaceRealization
IS-SUBTYPE-OF Realization
ASSOCIATES <<BehavioredClassifier-InterfaceRealization>>, <<Interface-InterfaceRealization>>
GENERATED_BY create
TYPES InterfaceRealization
FUNCTIONS
create: ∗ x _→ InterfaceRealization
END-CLASS

CLASS MultiplicityElement
IS-SUBTYPE-OF Element
ASSOCIATES <<MultiplicityElement-ValueEspecification$_0$>>,
<<MultiplicityElement-ValueEspecification$_1$>>
GENERATED_BY create
DEFERRED
TYPES MultiplicityElement
FUNCTIONS
create: Boolean x Boolean x Integer x UnlimitedNatural x_ → MultiplicityElement
get_isOrdered: MultiplicityElement → Boolean
get_isUnique: MultiplicityElement → Boolean
get_lower: MultiplicityElement → Integer
get_upper: MultiplicityElement → UnlimitedNatural
AXIOMS b1,b2: Boolean; i: Integer; un: UnlimitedNatural
get_isOrdered (create(b1,b2,i,un,_)) = b1
get_isUnique (create(b1,b2,i,un,_)) = b2
get_lower (create(b1,b2,i,un,_)) = i
get_upper (create(b1,b2,i,un,_)) = un …
END-CLASS

CLASS NamedElement
IS-SUBTYPE-OF Element
ASSOCIATES <<Dependency-NamedElement$_0$>>, << Dependency-NamedElement$_1$>>,
<< NamedElement-Namespace$_0$>>
GENERATED_BY create
DEFERRED
TYPES NamedElement
FUNCTIONS
create: String x VisibilityKind x _ → NamedElement
get_name: NamedElement → String
get_visibility: NamedElement → VisibilityKind
AXIOMS n:String; v:VisibilityKind
get_name (create(n,v,_)) = n
get_visibility (create(n,v,_)) = v …
END-CLASS

CLASS Namespace
IS-SUBTYPE-OF NamedElement
ASSOCIATES <<Namespace-PackageableElement>>, << NamedElement-Namespace$_0$>>,
<<NamedElement-Namespace$_1$>>, <<Constraint-Namespace<< <<ElementImport-Namespace>>,
<<NamespacePackageImport>>
GENERATED_BY create
DEFFERRED
TYPES Namespace
FUNCTIONS
create: ∗ x _ → Namespace …
END-CLASS

CLASS Operation
IS-SUBTYPE-OF BehavioralFeature
ASSOCIATES <<Operation-Parameter>>, <<Class-Operation>>,<<Operation-Type>>,
<<Constraint-Operation$_0$>>, <<Constraint-Operation$_1$>>, <<Constraint-Operation$_2$>>,
<<Operation-Operation>>
GENERATED_BY create
TYPES Operation
FUNCTIONS
create: ∗ x Boolean x _→ Operation
get_isQuery : Operation → Boolean
isEquivalentOperationTo: Operation x Operation → Boolean ...
AXIOMS b: Boolean
get_isQuery(create(∗, b, _)) = b ...
END-CLASS

CLASS Package
IS-SUBTYPE-OF Namespace, PackageableElemnt
ASSOCIATES << PackageableElement- Package>>, <<Package-Type>>, …
GENERATED_BY create
DEFFERRED
TYPES Package
FUNCTIONS
create: ∗ x _→ Package
AXIOMS ...
END-CLASS

CLASS PackageableElement
IS-SUBTYPE-OF NamedElement
ASSOCIATES …
GENERATED_BY create
DEFFERRED
TYPES PackageableElement
FUNCTIONS
create: ∗ x VisibilityKind x _ → PackageableElement …
END-CLASS

CLASS Property
IS-SUBTYPE-OF StructuralFeature
ASSOCIATES <<Property-Property$_0$>>, <<Property-Property$_1$>>, <<Property-Property$_2$>>,
<<Association-Property$_0$>>, <<Association-Property$_1$>>,
<<DataType-Property>>, <<Property-ValueSpecification>>, <<Class-Property>>
GENERATED_BY create
TYPES Property
FUNCTIONS
create: ∗ x Boolean x Boolean x String x AggregationKind x Boolean x _ → Property
get_isDerived: Property → Boolean
get_isDerivedUnion: Property → Boolean
get_Default: Property → String
get_aggregation: Property → AggregationKind
get_isComposite: Property → Boolean
is_Navigable: Property → Boolean
isEquivalentPropertyTo: Property x Property → Boolean
AXIOMS s: String; b1,b2,b3: Boolean, a:AggregationKind; p:Property
get_isDerived(create(∗, b1, b2,s,a,b3,b4,_)) = b1
get_isDerivedUnion(create(∗, b1, b2,s,a,b3,b4,_)) = b2

get_Default(create(∗, b1, b2,s,a,b3,b4,_)) = s
get_aggregation(create(∗, b1, b2,s,a,b3,b4,_)) = a
get_isComposite(create(∗, b1, b2,s,a,b3,b4,_)) = b3
get_isComposite(p) = isEqual(getAggregation(p), 'composite')
get_isReadOnly(p) => isNavigable(p)
isNavigable(p)….
END-CLASS

CLASS RedefinableElement
IS-SUBTYPE-OF NamedElement
ASSOCIATES <<Classiffier-RedefinableElement>>, <<RedefinableElement-RedefinableElement>>
GENERATED_BY create
DEFERRED
TYPES RedefinableElement
FUNCTIONS
create: ∗ x Boolean x _ → RedefinableElement
get_isLeaf: RedefinableElement → Boolean
AXIOMS b:Boolean
get_isLeaf(create(∗, b, _)) = b …
END-CLASS

CLASS Relationship
IS-SUBTYPE-OF Element
ASSOCIATES <<Relationhip-Element>>
GENERATED_BY create
DEFFERRED
TYPES Relationship
FUNCTIONS
create: _ → Relationship
END-CLASS

CLASS StructuralFeature
IS-SUBTYPE-OF Feature, MultiplicityElement, TypedElement
GENERATED_BY create
DEFERRED
TYPES StructuralFeature
FUNCTIONS
create: ∗ x Boolean x _ → StructuralFeature
get_isReadOnly: StructuralFeature → Boolean
AXIOMS b:Boolean
get_isReadOnly(create(∗,b,_)) = b
END-CLASS

CLASS Type
IS-SUBTYPE-OF PackageableElement
GENERATED_BY create
DEFFERRED
TYPES Type
FUNCTIONS
create: ∗ x _ → Type …
END-CLASS

CLASS TypedElement
IS-SUBTYPE-OF NamedElement

ASSOCIATES <<Type-TypedElement>>
GENERATED_BY create
DEFERRED
TYPES TypedElement
FUNCTIONS
create: ∗ x _ → TypedElement
END-CLASS