# COMPONENTES MDA PARA PATRONES DE DISEÑO

**Liliana Inés Martinez**

**Directora: Lic. Liliana 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

# Contenido

**1. Introducción** ........................................................................ 1
1.1. Motivación ........................................................................ 1
1.2. La propuesta ........................................................................ 2
1.3. Publicaciones ........................................................................ 3
1.4. Organización de esta tesis ........................................................................ 4

**2. Trabajos Relacionados** ........................................................................ 5

**3. Background** ........................................................................ 11
3.1. La arquitectura Model-Driven (MDA) ........................................................................ 11
3.1.1. El concepto de modelo en MDA ........................................................................ 11
3.1.2. Proceso de Desarrollo Model-Driven ........................................................................ 11
3.1.3. Lenguaje de modelado ........................................................................ 12
3.1.4. Metamodelos en MDA ........................................................................ 12
3.1.5. Relación entre modelos, lenguajes, metamodelos y metametamodelos ........................................................................ 12
3.1.6. Transformaciones, herramientas de transformación y definiciones de transformación ........................................................................ 13
3.1.7. El rol de MOF en MDA ........................................................................ 13
3.1.8. El rol de UML en MDA ........................................................................ 13
3.1.9. El rol de OCL en MDA ........................................................................ 14
3.1.10. El rol de QVT en MDA ........................................................................ 14
3.1.11. El framework de MDA ........................................................................ 15
3.1.12. Beneficios de MDA ........................................................................ 15
3.2. Patrones ........................................................................ 16
3.2.1. Introducción ........................................................................ 16
3.2.2. Patrones de diseño ........................................................................ 16
3.2.3. Clasificación de los patrones de diseño ........................................................................ 17
3.2.4. Beneficios de los patrones de diseño ........................................................................ 18
3.3. Componentes reusables ........................................................................ 18
3.3.1. Componentes reusables MDA ........................................................................ 19

3.4. El lenguaje NEREUS .................... 19
3.4.1. Introducción .................... 19
3.4.2. Especificaciones básicas .................... 19
3.4.3. Definición de asociaciones .................... 21
3.4.4. Especificaciones algebraicas modulares .................... 24

**4. Definiendo Componentes MDA** .................... 25
4.1. Introducción .................... 25
4.2. Especificando metamodelos de patrones de diseño .................... 27
4.2.1. Notación de los metamodelos de patrones de diseño .................... 28
4.2.2. El componente *Observer* .................... 28
4.2.2.1. Breve descripción de patrón de diseño *Observer* .................... 29
4.2.2.2. Metamodelo del patrón *Observer* a nivel PIM .................... 30
4.2.2.3. Metamodelo del patrón *Observer* a nivel PSM .................... 45
4.2.2.3.1. Metamodelo del patrón *Observer* para la plataforma Eiffel .................... 45
4.2.2.3.2. Metamodelo del patrón *Observer* para la plataforma Java .................... 55
4.2.2.4. Metamodelo del patrón *Observer* a nivel ISM .................... 70
4.2.2.4.1. Metamodelo del patrón *Observer* específico a la implementación Eiffel …..... 70
4.2.2.4.2. Metamodelo del patrón *Observer* específico a la implementación Java .......... 83
4.3. Especificando transformaciones de modelos basadas en metamodelos .................... 100
4.3.1. Sintaxis para la definición de transformaciones .................... 100
4.3.2 Transformación PIM-UML a PSM-EIFFEL .................... 101
4.3.3 Transformación PSM-EIFFEL A ISM-EIFFEL .................... 106

**5. Formalizando Componentes MDA** .................... 112
5.1. Introducción .................... 112
5.2. Formalizando metamodelos MOF .................... 113
5.2.1. Traducción de clases UML a NEREUS .................... 114
5.2.2. Traducción de asociaciones UML a NEREUS .................... 114
5.2.3. Traducción de restricciones OCL a NEREUS .................... 115
5.2.4. Traducción de la clase *Attach* a NEREUS .................... 116
5.2.5. Traducción de la asociación *AssocEndSubject-SubjectObserver* a NEREUS ........ 119
5.2.6. Formalizando el metamodelo del patrón de diseño *Observer* .................... 120
5.3. Formalizando refinamientos .................... 130

**6. Conclusiones** .................................................. 136
6.1. Contribuciones .................................................. 136
6.2. Futuros trabajos .................................................. 137

**Bibliografía** .................................................. 138

Anexo A: Metamodelo UML .................................................. 144
Anexo B: Especializaciones del metamodelo UML .................................................. 152
Anexo C: Especificación parcial del Metamodelo UML en NEREUS .................................................. 192

# 1. Introducción

## 1.1. Motivación

La arquitectura Model-Driven (Model-Driven Architecture o MDA) es un *framework* para el desarrollo de software definido por el Object Management Group (OMG) (MDA, 2007). Su propuesta es elevar el nivel de abstracción en el que se desarrollan sistemas complejos separando la especificación de la funcionalidad de un sistema de su implementación en una plataforma tecnológica específica. MDA promueve el uso de modelos y transformaciones de modelos para el desarrollo de sistemas de software.

El proceso de desarrollo MDA distingue cuatro clases de modelos:

- *Modelo independiente de la computación* (Computation Independent Model o CIM): describe los requerimientos del sistema y los procesos de negocio que debe resolver sin tener en cuenta aspectos computacionales.
- *Modelo independiente de la plataforma* (Platform Independent Model o PIM): es un modelo computacional independiente de las características específicas a una plataforma de desarrollo, como por ejemplo **.**NET, J2EE o relacional.
- *Modelo específico a la plataforma* (Platform Specific Model o PSM): describe un sistema en términos de una plataforma de implementación particular.
- *Modelo específico a la implementación* (Implementation Specific Model o ISM): se refiere a componentes y aplicaciones que usan lenguajes de programación específicos.

Los elementos esenciales de MDA son los modelos, los metamodelos y las transformaciones.

Un modelo es una descripción o especificación de un sistema y su ambiente para algún cierto propósito. Dentro de MDA un modelo debe estar escrito en un lenguaje de forma tal de ser interpretado por una computadora.

Dentro del contexto de MDA, los metamodelos son expresados usando MOF (Meta Object Facility) que define una forma común de capturar todos los estándares y construcciones de intercambio (MOF, 2006). Los metamodelos MOF se basan en los conceptos de entidades, interrelaciones y sistemas y se expresan como una combinación de diagramas de clases UML y restricciones OCL (UML-Infrastructure, 2007; UML-Superstructure, 2007; OCL, 2006).

La transformación de modelo a modelo es el proceso de convertir un modelo en otro modelo del mismo sistema. Para expresar las transformaciones, OMG está trabajando en la definición del QVT (Query\View\Transformation) para expresar transformaciones como una extensión de MOF (QVT, 2007).

El proceso de desarrollo Model-Driven (MDD) en el contexto de MDA es llevado a cabo como una secuencia de transformaciones de modelos que incluye al menos los siguientes pasos: construir un PIM, transformar el PIM en uno o más PSMs, y construir componentes ejecutables y aplicaciones directamente a partir de un PSM. Un alto grado de automatización de PIMs a PSMs, y de PSMs a ISMs es esencial en el proceso de desarrollo de MDA. Las herramientas que llevan a cabo la automatización se basan en la definición de las transformaciones, las cuales describen como generar un modelo a partir de otro.

El éxito de esta propuesta depende de la definición de las transformaciones entre modelos y de librerías de componentes que tengan un impacto significativo sobre las herramientas que proveen soporte a MDA.

Entre los posibles componentes MDA se pensó en definir componentes para patrones de diseño (Gamma y otros, 1995) dada su amplia difusión, aceptación y uso, debido a que describen soluciones a problemas de diseño recurrentes. Arnaut (2004) analiza los patrones de diseño de Gamma y otros (1995) para identificar cuales de ellos pueden ser transformados en componentes reusables en una librería Eiffel. Su hipótesis de trabajo es que "los patrones de diseño son buenos, pero los componentes son mejores". En este caso en particular la reusabilidad está dada en términos de código, pero nos inspiró a pensar en los patrones de diseño en términos de componentes MDA.

## 1.2. La propuesta

Como consecuencia de las ideas antes mencionadas se propone una técnica de metamodelado para alcanzar un alto nivel de reusabilidad y adaptabilidad en una perspectiva de MDA.

Se propone un "megamodelo" para definir familias de componentes de patrones de diseño alineados a MDA. Un "megamodelo" es un modelo cuyos elementos representan modelos, metamodelos, metametamodelos, servicios, herramientas, etc. (Bézivin y otros, 2004).

Los elementos que componen el "megamodelo" propuesto son metamodelos y refinamientos. Los componentes fueron definidos como una unidad que encapsula metamodelos y refinamientos. Los metamodelos fueron definidos en tres niveles de abstracción: PIM, PSM e ISM y se vinculan a través de refinamientos que transforman un modelo origen en un modelo destino, ambos en diferentes niveles de abstracción (PIM a PSM, PSM a ISM).

El desarrollo de componentes reusables requiere poner énfasis sobre la calidad del software, por lo cual las técnicas tradicionales para la verificación y validación son esenciales para lograr atributos de calidad en el software, tales como consistencia, corrección, robustez, eficiencia, confiabilidad etc. En el contexto de los procesos MDD, los formalismos pueden ser usados para detectar inconsistencias tanto en los modelos internos o entre el modelo origen y el modelo destino que ha sido obtenido a través de transformaciones de modelos. Una especificación formal clarifica el significado deseado de los metamodelos y de las transformaciones de modelos basadas en metamodelos ayudando a validarlos y proveyendo una referencia para la implementación.

En esta dirección, se propone especificar componentes MDA en NEREUS (Favre, 2005) a través de un sistema de reglas de transformación. NEREUS es un lenguaje algebraico que permite especificar metamodelos basados en los conceptos de entidad, relaciones y sistemas. NEREUS es un lenguaje alineado con los metamodelos MOF: cada construcción en MOF está asociada a una entidad de primera clase en NEREUS, por ejemplo las asociaciones. La mayoría de los conceptos de los metamodelos basados en MOF pueden ser traducidos a NEREUS en una forma directa. La necesidad de la formalización se debe a que MOF está basado en UML y OCL, los cuales son imprecisos y ambiguos cuando se trata de la simulación, verificación, validación y predicción de las propiedades del sistema. La falta de formalización de UML es referida en la Infraestructura de este lenguaje donde se cita: "It is important to note that the specification of UML as a metamodel does note preclude it from being specified via a mathematically formal language (e.g., Object-Z or VDM) at a later time" (UML-Infrastructure, 2007, p.11).

## 1.3. Publicaciones

Esta tesis fue desarrollada en el marco de un proyecto más amplio que tiene entre otros objetivos la integración de especificaciones formales y semiformales y donde surgió el lenguaje de metamodelado *NEREUS*. A continuación se mencionan publicaciones relacionadas a esta tesis que fueron el resultado de investigaciones dentro de dicho proyecto.

*Foundations for MDA Case Tools.*
Co-Autores: Favre Liliana, Pereira Claudia. Artículo aceptado para la Encyclopedia of Information Science and Technology, Second Edition, IGI Global, USA. A ser publicado en 2008.

*Formalizing MDA Components.*
Co-Autor: Favre Liliana. International Conference on Software Reuse (ICRS 2006). Torino. Italia. Lecture Notes in Computer Science 4039, Springer Verlag Berlin Heidelberg. ISNN 0302-9743. 2006. Páginas 326-339.

*MDA-Based Design Pattern Components.*
Co-Autor: Favre Liliana. Publicado en: Proceedings de 2006 Information Resources Management Association International Conferences (IRMA 2006). Washington, D.C. USA. ISBN 1-59904-019-0. 21-24 de Mayo, 2006. Páginas 259-263.

*Forward Engineering of UML Static Models.*
Co-Autores: Favre Liliana, Pereira Claudia. Artículo invitado en: Encyclopedia of Information Science and Technology, Volume I-V Mehdi Khosrow-Pour (Editor). IGP (Idea Group Publishing) .USA. ISBN 1-59140-553-X. 2005. Páginas 1212-1217.

*Integrating Design Patterns into Forward Engineering Processes.*
Co-autores: Favre Liliana, Pereira Claudia. Publicado en: Proceedings de 2004 Information Resources Management Association International Conference (IRMA 2004). New Orleans, USA. ISBN 1-59140-261-1. 23-26 de Mayo, 2004. Páginas 502-505.

*Una Integración de Patrones de Diseño en Procesos de Ingeniería Forward de Modelos Estáticos UML.*
Co-autor: Liliana Favre. Publicado en: VI Workshop de Investigadores en Ciencias de la Computación (WICC 2004). Neuquén, Argentina. Mayo 2004. Páginas 249-253.

*Forward Engineering and UML: From UML Static Models to Eiffel Code.*
Co-autores: Favre Liliana, Pereira Claudia. Publicado en: UML and the Unified Process (Liliana Favre editor). Capítulo IX. IRM Press, 2003. ISBN 1-931777-44-6, USA. Páginas 199-217.

*Forward Engineering and UML: From UML Static Models to Eiffel Code.*
Co-autores: Favre Liliana, Pereira Claudia. Publicado en: Proceedings de 2002 Information Resources Management Association (IRMA 2002). ISBN 1-930708-39-4. Seattle, USA. Mayo de 2002. Páginas 584-588.

*Una integración de modelos estáticos UML y Eiffel.*
Co-autores: Favre Liliana, Pereira Claudia. Publicado en: Proceedings del VII Congreso Argentino de Ciencias de la Computación (CACIC 2001). Calafate, Santa Cruz, Argentina. Octubre de 2001. Páginas 521-530.

*Transforming UML Static Models to Object Oriented Code.*
Co-autores: Favre Liliana, Pereira Claudia. Publicado en: Proceedings de 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.

## 1.4. Organización de esta tesis

Este trabajo está organizado como se detalla a continuación.

En el capítulo 2 se citan trabajos relacionados a esta tesis: patrones de diseño (automatización, especificación y herramientas que los soportan), componentes para el reuso, especificación de las transformaciones de modelos, componentes MDA y herramientas que soportan MDA.

En el capítulo 3 se presenta el marco teórico, el cual da la base necesaria para la comprensión de esta tesis. Se describe la arquitectura Model-Driven y conceptos relacionados a la misma (modelos, metamodelos, transformaciones), patrones (clasificación, ventajas), componentes reusables y el lenguaje NEREUS.

El capítulo 4 detalla como definir componentes MDA a través de la especificación de los metamodelos involucrados y la especificación de las transformaciones entre los mismos. La propuesta es ilustrada a través de la definición de un componente para el patrón de diseño *Observer*.

En el capítulo 5 se detalla como formalizar componentes MDA a través de la traducción de metamodelos y transformaciones a NEREUS.

El capítulo 6 considera las conclusiones, contribuciones y trabajos futuros.

El Anexo A hace un breve repaso del metamodelo UML, el Anexo B describe los metamodelos correspondientes a las plataformas Java y Eiffel y los metamodelos específicos a la implementación Java y Eiffel y el Anexo C presenta una especificación parcial del metamodelo UML en NEREUS.

# 2. Trabajos Relacionados

Desde la aparición de los patrones de diseño, se han desarrollado muchos trabajos y herramientas para asistir al programador en la aplicación de un patrón y en la generación automática de código a partir de éste. La desventaja en general de éstas es, por un lado, la falta de integración del código del patrón con el resto del código, y por otro lado, la falta de independencia del lenguaje ya que no son capaces de generar código en más de un lenguaje. Algunas herramientas generan el código en un *workspace* diferente de la especificación del software general y ambiente de código, por lo cual es necesario "cortar y pegar" manualmente el código del patrón. Algunos de los trabajos más relevantes son listados a continuación.

Budinsky y otros (1996) describen una herramienta que automatiza la implementación de patrones de diseño. El usuario aporta información específica a la aplicación para un patrón dado, a partir de la cual la herramienta genera el código del patrón automáticamente. La gran promesa de esta propuesta es que el diseñador puede fácilmente adaptar la implementación del patrón a las necesidades exactas de su problema. Sin embargo tiene dos problemas. Por un lado el usuario debe entender que cortar y donde pegar y esto no es obvio. Por otra parte, una vez que el usuario ha incorporado código generado en su aplicación, cualquier cambio que involucre regenerar el código, lo forzará a reincorporarlo en la aplicación. Además el usuario no puede ver cambios en el código generado a través de la herramienta.

Florijn y otros (1997) describen el prototipo de una herramienta que soporta patrones de diseño durante el desarrollo o mantenimiento de programas orientados a objetos. El objetivo fue introducir patrones como ciudadanos de primera clase en los ambientes de desarrollo orientado a objetos integrados.

En Albin-Amiot y Guéhéneuc (2001a) se presentan dos herramientas que ayudan a los desarrolladores a implementar grandes aplicaciones y grandes *frameworks*, usando patrones de diseño. Scriptor (Generación Pura) es un generador de aplicaciones fuertemente industrial. El principal interés de esta propuesta es ocultar el código generado. Los patrones de diseño se implementan directamente en diagramas UML estableciendo explícitamente que una clase dada corresponde, por ejemplo, al patrón Singleton. Una vez que el código es generado y entregado, no hay forma de localizar qué patrón de diseño ha sido aplicado y dónde ha sido aplicado. PatternsBox (Generación conservativa) es una herramienta para instanciar patrones de diseño. El principal interés de esta propuesta es conservar todos los atributos del código fuente. Los desarrolladores eligen explícitamente los artefactos en su código fuente que juegan un rol en un patrón de diseño, adaptan el patrón a esos artefactos por medio de parametrización y luego instancian el patrón de diseño en su código fuente. Se modifican o crean sólo aquellos artefactos de código necesarios (clases, interfaces, métodos) para implementar el patrón de diseño, manteniendo el resto del código sin tocar. La desventaja de esta propuesta es que los desarrolladores necesitan escribir la mayoría o gran parte del código a mano, mientras definen implementan y documentan las decisiones de diseño.

La propuesta de esta tesis, permitiría la generación del código correspondiente al patrón de diseño de forma integrada con el código que se genera para el resto del modelo que contiene al patrón. A si mismo, los componentes están enmarcados en MDA, lo que permitiría la

generación de código en más de un lenguaje a partir de modelos independientes a la plataforma y no solo dependientes de la misma.

Ha habido también innumerables trabajos sobre especificación de patrones de diseño usando distintas técnicas formales y semiformales. A continuación se describen algunos de estos trabajos relacionados.

Edén y otros (1997) describen un método preciso que especifica cómo aplicar un patrón de diseño expresándolo como un algoritmo en un lenguaje de meta-programación. Presentan un prototipo de una herramienta de que soporta la especificación de patrones de diseño y su realización en un programa dado. El prototipo permite la aplicación automática de un patrón de diseño sin obstruir el texto de código de fuente del programador quien puede revisarlo a voluntad. Muestran la especificación del modelo meta-programando, las técnicas y un resultado de la muestra de su aplicación. Concluyen que la aplicación más beneficiosa (y realista) de un patrón de diseño sería adaptar las estructuras existentes en un programa, una tarea más compleja y dependiente del contexto que la generación "bruta" de código.

Albin-Amiot y Guéhéneuc (2001b) presentan cómo un metamodelo puede ser usado para obtener una representación de patrones de diseño y cómo esta representación permite tanto la generación automática como la detección de patrones de diseño. La solución presentada se basa en un lenguaje de modelado, donde la generación de código y la *traceability* son direccionadas. Los patrones son formalizados usando un conjunto de bloques básicos llamados entidades y elementos. Esos bloques son definidos en el corazón del metamodelo dedicado a la representación de patrones. El metamodelo no fue definido como una extensión de otro ya existente (como UML). El metamodelo propuesto experimentado en Java, provee un medio de describir aspectos estructurales y de conducta de los patrones de diseño. A partir de esa descripción se obtiene la maquinaria necesaria para producir código y detectar patrones instanciados en el código. La contribución deseada de esta propuesta es la definición de patrones de diseño como entidades de modelado de primera clase. La limitación es la falta de integración del código generado con el código del usuario.

Kim y otros (2003a) presentan un lenguaje de metamodelado basado en roles (RBML) para la especificación de patrones de diseño. La especificación de un patrón define una familia de modelos UML en términos de roles, donde un rol es asociado con una metaclase UML como su base. Un rol especifica las propiedades que el elemento del modelo que juega dicho rol debe poseer, es decir, las propiedades determinan un subconjunto de instancias de esta base. RBML permite especificar distintas perspectivas de patrones tal como estructura estática, interacciones y la conducta basada en estados. Este lenguaje usa una notación visual basada en UML y restricciones textuales expresadas en OCL para definir las propiedades del patrón. Puede ser usado por desarrolladores de herramientas que necesiten descripciones precisas de patrones de diseño.

Kim y otros (2003b) y France y otros (2004) presentan una técnica para especificar soluciones de patrones de diseño expresadas en UML usando roles. La intención de la propuesta es facilitar el desarrollo de herramientas que soportan la aplicación rigurosa de patrones de diseño para modelos UML. Establecen que usar roles de modelo permite a los desarrolladores de herramientas extender el mecanismo usado para verificar que los modelos UML conforman las especificaciones de patrones que son metamodelos. El primer trabajo examina las características de los roles basados en objetos y los generalizan. A partir de esta generalización definen roles a nivel de metamodelo para especificar patrones de diseño donde cada elemento del modelo (por ejemplo, clases, asociaciones) juega un rol determinado. Tal rol se refiere a un rol del modelo. En el segundo trabajo se establece que las especificaciones

creadas por la técnica propuesta son metamodelos que caracterizan los modelos de diseño UML. La especificación de patrones consiste de una especificación estructural que especifica la vista del diagrama de clases de soluciones de patrones y un conjunto de especificaciones de interacción que especifica las interacciones en las soluciones del patrón.

Elaasar y otros (2006) presentan un framework de modelado de patrones (Pattern Modeling Framework), basado en el metamodelado para la especificación y detección de patrones. Provee un lenguaje de especificación de patrones llamado Epattern (definido como una extensión de MOF), capaz de especificar en forma precisa patrones en metamodelos alineados a MOF. Presentan también una estrategia para generar un algoritmo de detección de patrones a partir de una especificación Epattern y una herramienta que soporta la propuesta.

Debnath y otros (2006) presentan una forma de definir patrones a través de *profiles* UML. Proponen una arquitectura estructurada en niveles, una jerarquía entre niveles permite el reuso de definiciones. El *profile* es usado como una herramienta para documentar y definir patrones de diseño. Cada *profile* describe la semántica de un patrón de diseño en particular. Los desarrolladores pueden introducir estereotipos, *tags* y restricciones OCL en sus modelos lo que les permite ver claramente el patrón usado, mejorar la comunicación con sus colegas y establecer un vocabulario común. Además analizan cómo una herramienta CASE UML debería definir incorporar *profiles* en los modelos y cómo deberían chequear si el modelo es consistente con respecto a las restricciones del *profile*.

Estos trabajos presentan distintas formas de especificar patrones. El objetivo de algunos fue la documentación y definición, el de otros la aplicación automática de los patrones y la detección automática de patrones y/o la generación de código a partir de ellos. En esta tesis se propone englobar todos estos objetivos a través de la especificación de componentes para patrones de diseño enmarcados en MDA, en los cuales metamodelos de patrones fueron especificados en tres niveles de abstracción: PIM, PSM y código utilizando la notación MOF. Esta especificación permitiría tanto la detección de un patrón en los tres niveles de abstracción, como así también la generación de código a partir de un modelo en más de un lenguaje de programación.

A medida que los patrones de diseño han cobrado importancia, varios IDE (Integrated Development Environments) y ambientes de modelado de software UML han comenzado a introducir soporte para los mismos.

Bulka explora el estado del software de automatización de patrones en 2002 y discute las ventajas y desventajas de varias propuestas (Bulka 2002). Existen varios grados de automatización de patrones ofrecida por las herramientas de modelado UML. Ellas van desde mecanismos estáticos a dinámicos. Las propuestas estáticas simplemente insertan un grupo de clases relacionadas en un *workspace*, mientras las dinámicas tienden a integrar el nuevo patrón insertado con las clases existentes, renombrando clases y nombres de métodos según sea requerido. Niveles aceptables de automatización proveen *wizards* que permiten adaptar patrones a un problema y un contexto en particular. La gran promesa de los *wizard* es que el diseñador puede ajustar más fácilmente la implementación del patrón a las necesidades de su aplicación. Sin embargo hasta que se construya una gran librería de implementaciones alternativas de patrones, la mayoría de los diseñadores se sentirán restringidos a las elecciones de *templates* de patrones disponibles. Niveles de automatización avanzados van más allá, las clases involucradas en patrones responden automáticamente e inteligentemente a cambios

hechos por el diseñador en otras partes del modelo. De esta forma los patrones mantienen su integridad y no pueden ser dañados accidentalmente por el diseñador.

A continuación se mencionan algunas de las herramientas disponibles en el mercado que soportan la automatización de patrones de diseño.

UMLStudio (2007) es una propuesta de *template* simple para la automatización de patrones de diseño. El usuario de esta herramienta de modelado UML navega a través de un catálogo de patrones, viendo varios diseños. Selecciona el patrón deseado y éste aparece sobre el *workspace*. Las clases existentes son ignoradas por el *template* del patrón seleccionado, las nuevas clases el patrón seleccionado a partir de la librería no son integradas con las clases existentes. Por lo tanto, el usuario deberá hacer las modificaciones necesarias a las clases y nombres de métodos, y con frecuencia deberá mover las clases existentes a las posiciones ocupadas por las nuevas clases a partir del *template* de manera de integrar completamente las clases del patrón en el modelo UML existente. La necesidad de adaptar las clases extraídas de la librería de *templates* en forma manual es la mayor desventaja de esta técnica.

Objecteering/UML (2007) y ModelMaker (2007) son propuestas más inteligentes y dinámicas para la automatización de patrones. Estas herramientas permiten ligar patrones a las clases existentes y ofrecen la habilidad de personalizar aspectos de la aplicación usando cajas de diálogo *wizard*. Las cajas de diálogo preguntan por el nombre de clases y métodos antes de insertar un grupo de clases relacionadas en un *workspace*. El beneficio de estos *templates* de patrones parametrizados es que las nuevas clases y métodos son integradas con las clases y métodos existentes, ya que los objetos UML seleccionados y los parámetros ingresados en el *wizard* durante el proceso de automatización de parámetros da a la herramienta la información necesaria para hacer la integración. Las clases del nuevo patrón insertado son integradas con clases existentes. La desventaja de los *templates* de patrones parametrizados es que el usuario está trabajando en una caja de diálogo, tipear y recordar nombres de métodos y clases es tedioso y propenso a errores.

Entre los trabajos vinculados a componentes para reuso cabe mencionar a Meyer (2003) y Arnout (2004).

Meyer (2003) examina los principales conceptos que hay detrás de los “componentes confiables” en la industria del software. Por un lado establece las principales propiedades que debe tener un modelo de componente de alta calidad, de manera tal de proveer un servicio de certificación para los componentes existentes. Este modelo divide las propiedades de interés en cinco categorías de calidad de componentes (Aceptación, Comportamiento, Restricciones, Diseño y Extensión). Por otra parte Meyer apunta a la producción de componentes que garanticen dichas propiedades de calidad. El trabajo de tesis toma estas ideas y contribuye con una técnica de metamodelado para construir componentes de software reusables en una perspectiva MDA. Para lograr la confiabilidad de los componentes se propuso la formalización de los mismos.

Arnout (2004) analiza los patrones propuestos por Gamma para identificar cuáles pueden transformarse en componentes reusables y describe el componente de software correspondiente de aquellos que sí se pueden transformar. Su hipótesis de trabajo es que “los patrones de diseño son buenos, pero los componentes son mejores”. Señala que los componentes fueron escritos en Eiffel porque ofrece mecanismos orientados a objetos, útiles en esta tarea, y soporta Diseño por Contrato, clave para su trabajo. Sin embargo la propuesta no está limitada a dicho lenguaje. Alrededor del 65% de los patrones descriptos en Gamma

fueron transformados en componentes reusables de una librería. Presenta una herramienta llamada Pattern wizard que es capaz de generar esquemas de código para el resto de los patrones (que no pueden transformarse en componentes), de manera tal de liberar a los programadores de buena parte del trabajo de generar código. Si bien en este trabajo la reusabilidad está dada en términos de código, fue una inspiración para pensar en patrones de diseño en términos de componentes MDA, elevando de esta manera el nivel de abstracción de los mismos.

En la actualidad la propuesta MDA está tomando cada vez más fuerza en el desarrollo de software. Desde su aparición han surgido numerosos trabajos relacionados con tópicos vinculados con ella como son: lenguajes de modelado, especificación de las transformaciones entre modelos, herramientas de soporte, entre otros. En particular se mencionarán algunos trabajos relacionados a la especificación de transformaciones de modelos.

En Czarnecki y Helsen (2003) se analiza una taxonomía para la clasificación de varias propuestas de transformación de modelo a modelo existente. La taxonomía se describe con un modelo de características para compararlas.

Judson y otros (2003) describen una propuesta para el modelado riguroso de transformaciones basadas en patrones que involucra la especialización del metamodelo UML para caracterizar los modelos origen y destino.

Kuster y otros (2004) comparan y contrastan dos propuestas de transformaciones de modelos: una es una transformación de grafos y la otra una propuesta relacional.

Cariou y otros (2004) proponen especificar transformaciones de modelos independientemente de cualquier tecnología de transformación. Para lograr esta meta, proponen definir contratos. Estos contratos para la transformación de modelos son usados para la especificación, validación y testeo de las transformaciones. Discuten la relevancia y los límites de OCL para definir dichos contratos. Se enfocan sobre las transformaciones de diagramas de clases UML a diagramas de clases UML, pero establecen que dichos contratos pueden ser definidos para cualquier clase de modelos o diagramas.

Giandini y otros (2006) y Giandini y otros (2007) presentan un lenguaje puramente declarativo para expresar transformaciones entre modelos. Para implementar el metamodelo de transformaciones extienden la Infraestructura 2.0 a través de estereotipos y usan el lenguaje OCL para expresar patrones de transformación. En el primer trabajo el metamodelo propuesto pretende ser mínimo para poder expresar relaciones y *querys* de transformación entre modelos. En el segundo trabajo proponen un metamodelo que usa las metaclases OCL existentes y simplifica el metamodelo propuesto en el primero.

A diferencia de los trabajos mencionados, se propuso las transformaciones de modelos como contratos OCL basadas en los metamodelos origen y destino e integradas como parte de un componente.

La noción de componentes vinculados a MDA ha aparecido recientemente como una característica necesaria para el desarrollo exitoso de la misma. A continuación se mencionan algunos trabajos relacionados a los mismos que nos inspiraron a desarrollar componentes para patrones de diseño en el contexto MDA.

Bettin (2003) resume lecciones que surgen de varios proyectos relacionados al desarrollo basado en componentes y MDA y examina el uso pragmático de las herramientas MDA actuales.

Bézivin y otros (2003) definen los componentes MDA como "una unidad de encapsulamiento para cualquier artefacto usado o producido dentro de un proceso relacionado a MDA". Este trabajo introduce varios conceptos y entidades que están presentes en un contexto MDA y dan varios ejemplos de componentes MDA. Presentan a las transformaciones de modelos como el componente más importante.

Mottu y otros (2006) proponen encapsular las transformaciones de modelos como componentes MDA. Proponen un modelo para componentes confiables como también una metodología para diseñar e implementar tales componentes. Definen un componente MDA como un conjunto orgánico compuesto de tres facetas: casos de testeo, implementación y un contrato que define su especificación. Un componente confiable es considerado un "vigilante" en el sentido que contiene contratos suficientemente precisos como para detectar la mayoría de las situaciones erróneas a tiempo de ejecución. La confiabilidad es evaluada usando un proceso de testeo que refleja la consistencia entre la especificación y la implementación del componente.

El éxito de MDA depende de la existencia de herramientas CASE (Computer Arded Software Engineering) que tengan un impacto significativo sobre el proceso de desarrollo de software (CASE TOOLS, 2007). En la actualidad hay pocas herramientas CASE que proveen soporte a MDA. Algunas de ellas son OptimalJ, Objeteering/UML, ArcStyler, AndroMDA, entre otras. Las técnicas que existen actualmente en estas herramientas proveen poco soporte para analizar la consistencia de transformaciones de modelos. Favre y otros (2008) explican los desafíos más importantes para automatizar los procesos que deberían ser soportados por herramientas MDA. Existen trabajos que discuten distintas herramientas que soportan MDA, entre ellos pueden mencionarse Molina y otros (2004) y Bollati y otros (2007). El primero realiza un estudio comparativo entre las herramientas OptimalJ y ArcStyler Mientras el segundo analiza AndroMDA y ArgoUML. La integración de una librería de componentes para patrones de diseño podría mejorar el soporte de estas herramientas a MDA.

# 3. Background

## 3.1. La arquitectura Model-Driven (MDA)

La Arquitectura Model-Driven (MDA) es un framework para el desarrollo de software definido por el Object Management Group (OMG). La clave de MDA es la gran importancia de los *modelos*, los cuales guían el proceso de desarrollo de software. Esta sección describe conceptos relacionados a dicho framework (Kleppe y otros, 2003; MDA, 2003).

### 3.1.1. El concepto de modelo en MDA

Un modelo es una descripción o especificación de un sistema y su ambiente para algún cierto propósito. Los siguientes modelos son el centro de MDA:

- *Modelo independiente de la computación* (Computation Independent Model o CIM): también llamado modelo del dominio muestra el sistema en el ambiente en que operará y lo que se espera que haga. Es útil no sólo para entender el problema, sino también como fuente de un vocabulario común para usar en los otros modelos.
- *Modelo independiente de la plataforma* (Platform Independent Model o PIM): tiene un alto nivel de abstracción y es independiente de cualquier tecnología de implementación. Se centra en el funcionamiento de un sistema mientras oculta los detalles necesarios para una plataforma particular. Este modelo muestra aquella parte de la especificación completa que no cambia de una plataforma a otra, por ejemplo dotNET o J2EE.
- *Modelo específico a la plataforma* (Platform Specific Model o PSM): combina el punto de vista independiente de la plataforma con un enfoque adicional en el detalle del uso de una plataforma específica para un sistema. Es adecuado para especificar el sistema en términos de construcciones de implementación que son disponibles en una tecnología de implementación específica. Por ejemplo, un PSM EJB es un modelo del sistema en términos de estructuras EJB. Típicamente contendrá términos específicos EJB, como “home interface”, “entity bean”, etc.
- *Modelo específico a la implementación* (Implementation Specific Model o ISM): muestra el sistema a nivel de código, se refiere a componentes y aplicaciones escritos en lenguajes de programación tales como Java o Eiffel.

### 3.1.2. Proceso de desarrollo Model-Driven

MDA es una realización del proceso de desarrollo Model-Driven (Model-Driven Development o MDD), el cual es llevado a cabo como una secuencia de transformaciones de modelos que incluye al menos los siguientes pasos:

1. Construcción de un PIM.
2. Transformación del PIM en uno o más PSMs.

3. Transformación de cada PSM a código (componentes ejecutables y aplicaciones).

Un alto grado de automatización de PIMs a PSMs, y de PSMs a ISMs es esencial en el proceso de desarrollo de MDA. Existen herramientas anteriores a MDA que transforman PSM a código, ya que el PSM es más cercano al código y su transformación es más directa.

### 3.1.3. Lenguaje de modelado

Otro elemento del framework de MDA es el lenguaje de modelado. Un modelo es siempre escrito en un lenguaje. Debido a que los PIMs y PSMs son transformados automáticamente, deben ser escritos en un lenguaje de modelado capaz de ser interpretado por una computadora.

Los modelos usados en MDA pueden ser expresados usando el lenguaje UML, aunque no está restringido a éste. UML es el estándar definido por OMG para el análisis y diseño orientado a objetos.

### 3.1.4. Metamodelos en MDA

El metamodelo es un lenguaje para especificar modelos, que describe los elementos que pueden ser usados en dicho modelo. Cada elemento que un modelador puede usar en su modelo está definido por el metamodelo del lenguaje. En UML por ejemplo, se pueden usar clases, atributos, asociaciones, estados, acciones, etc., porque en el metamodelo de UML hay elementos que definen qué es una clase, qué es un atributo, etc.

Los metamodelos dentro de MDA son importantes por dos razones:

1. Un metamodelo define un lenguaje de modelado de forma tal que una herramienta de transformación pueda leer, escribir y entender los modelos.
2. La definición de transformación que describe como transformar un modelo en un lenguaje origen en un modelo en un lenguaje destino se basa en los metamodelos de ambos lenguajes.

### 3.1.5. Relación entre modelos, lenguajes, metamodelos y metametamodelos

El modelo describe información del dominio.

El metamodelo es un modelo de modelos, estableciendo que elementos pueden usarse en este último. Por ejemplo, UML define que en un modelo UML se pueden usar los conceptos “Class”, “State”, “Package”, etc.

El meta-metamodelo es el lenguaje para especificar metamodelos.

OMG define una arquitectura de cuatro capas para sus estándares, llamadas M0, M1, M2 y M3:

1. En el nivel M3 reside el *meta-metamodelo*. Este nivel forma la base de la arquitectura de metamodelado. Dentro de OMG, MOF es el lenguaje estándar M3. Lenguajes de modelado como UML, CWM (Common Warehouse Model) (CWM, 2003) son instancias del MOF.
2. En el nivel M2 reside el *metamodelo*. Éste es una instancia de un meta-metamodelo (elemento de M3) que a su vez especifica elementos de la capa M1.
3. El nivel M1 contiene modelos. Cada modelo es una instancia de un metamodelo (elemento de M2) y a su vez especifica elementos de la capa M0.

4. En el nivel M0 se encuentra un sistema en ejecución en el cual existen instancias reales, los elementos en este nivel son *instancias* de los elementos especificados por modelos a nivel M1.

### 3.1.6. Transformaciones, herramientas de transformación y definiciones de transformación

En el proceso de desarrollo de MDA es esencial un alto grado de automatización de las transformaciones. Una herramienta de transformación toma un PIM y lo transforma en un PSM. Una segunda herramienta (o la misma) transforma el PSM a código. Para llevar a cabo esta tarea las herramientas deben contar con una definición de transformación que describa cómo transformar el modelo.

Existe una distinción entre transformación en sí, que es el proceso de generar un modelo nuevo a partir de otro, y la definición de la transformación. La herramienta de transformación usa la misma definición de transformación para transformar cualquier modelo de entrada.

Una transformación es la generación automática de un modelo destino a partir de un modelo origen, de acuerdo a una definición de transformación.

Una definición de una transformación de modelos es una especificación de un mecanismo para convertir los elementos de un modelo, que es una instancia de un metamodelo particular, en elementos de otro modelo, que es instancia de otro metamodelo o posiblemente el mismo.

La característica más importante de una transformación es que debe preservar la semántica entre los modelos origen y destino. El significado de un modelo sólo puede preservarse si se puede expresar tanto en el modelo origen como en el destino.

### 3.1.7. El rol de MOF (Meta Object Facility) en MDA

MOF es un estándar de OMG que define un lenguaje común y abstracto para definir lenguajes de modelado como así también para construir herramientas para la definición de dichos lenguajes. Dentro de la arquitectura de cuatro capas definida por OMG, MOF reside en el nivel M3 (nivel de meta-metamodelado) y se autodefine usando MOF. Usa cinco construcciones básicas para definir un lenguaje de modelado: clases, generalización, asociaciones, atributos y operaciones.

Dentro de MDA, MOF provee conceptos y herramientas para razonar sobre lenguajes de modelado. Usando la definición de MOF de un lenguaje de modelado (es decir el metamodelo de la capa M2) se pueden definir transformaciones entre lenguajes de modelado. Sin un estándar que describa los metamodelos, las transformaciones no podrían ser definidas apropiadamente y la propuesta MDA sería muy difícil de realizar.

### 3.1.8. El rol de UML en MDA

El lenguaje de modelado unificado UML es el estándar de OMG para el análisis y diseño orientado a objetos. Es un lenguaje para especificar, visualizar, construir y documentar artefactos de sistemas de software, como también modelos de negocios y otros sistemas que no son de software.

Dentro de MDA, UML se puede utilizar para:

1. *Crear modelos del sistema a construir*. Para esto es necesario conocer cómo y dónde aplicar UML para desarrollar modelos lo suficientemente precisos y consistentes como para que puedan ser usados dentro de MDA.

2. *Definir transformaciones entre modelos*. Para llevar a cabo esta tarea además de tener un profundo conocimiento de UML y su uso, se necesita estar íntimamente familiarizado con el metamodelo UML, ya que en términos de este metamodelo, se definen las transformaciones en MDA.

### 3.1.9. El rol de OCL en MDA

El Lenguaje de Restricciones de Objetos (OCL) es un lenguaje de expresiones en el que se pueden escribir las expresiones sobre modelos. Estas expresiones especifican condiciones invariantes que deben ser cumplidas por el sistema que está siendo modelado o *queries* sobre objetos descriptos en el modelo. Son expresiones libres de efectos secundarios, cuando son evaluadas, simplemente retornan un valor.

En un comienzo, OCL se usó en UML a especificar restricciones: invariantes, precondiciones y postconditions. Recientemente una nueva versión de OCL (versión 2.0) permite además de restricciones definir *queries*, referenciar valores o establecer condiciones y reglas de negocios.

OCL tiene los siguientes beneficios dentro de MDA:

- *Da más precisión a los modelos*: un modelo UML de un sistema es más preciso y más completo aplicando OCL. En el contexto de MDA, esto significa que el modelo que es el punto de partida del proceso de ingeniería forward es más rico y hace posible generar una especificación más completa.
- *Ayuda a definir lenguajes*: dentro del framework de MDA los lenguajes deben tener una definición precisa. OCL puede ser usado tanto en modelos UML como en modelos MOF. OCL extiende el poder expresivo de UML/MOF permitiendo crear modelos más precisos. El primer uso de OCL con MOF fue en la definición del metamodelo UML. Cientos de invariantes, llamadas "reglas bien formadas", fueron escritas en OCL para completar los diagramas que describen el metamodelo UML. El metamodelo es así más preciso y constituye una mejor especificación con menos ambigüedad.
- *Ayuda a construir definiciones de transformación*: Una transformación mapea uno o más elementos de un modelo origen a uno o más elementos en un modelo destino. Existen transformaciones que sólo pueden ser aplicadas bajo ciertas condiciones. Éstas pueden especificarse en OCL, dando una condición sobre los elementos del modelo origen y una segunda condición sobre los elementos del modelo destino. Todas las expresiones OCL usadas en una definición de transformación se especifican sobre el metamodelo de los lenguajes origen y destino.

### 3.1.10. El rol de QVT (Query/View/Transformation) en MDA

Actualmente OMG está desarrollando el estándar QVT para la especificación de transformaciones. Direcciona la forma en que son alcanzadas las transformaciones entre los modelos cuyos lenguajes son definidos usando MOF. Está compuesto de tres sublenguajes:

- Un lenguaje para crear vistas sobre modelos
- Un lenguaje para modelos de *query*
- Un lenguaje para escribir definiciones de transformaciones

Siendo este último es el más relevante dentro de MDA.

### 3.1.11. El framework de MDA

Las secciones previas describen brevemente cada uno los elementos principales que forman parte del framework de MDA: modelos, metamodelos, transformaciones, definiciones de transformaciones y herramientas que realizan las transformaciones. Todos estos elementos juntos forman el framework básico de MDA, como se muestra en la Figura 3.1.

### 3.1.12. Beneficios de MDA

Los beneficios de MDA en términos de mejoras en el proceso de desarrollo de software son los siguientes:

- **Productividad:** La ganancia de productividad se puede alcanzar usando herramientas que automaticen completamente la generación de código a partir de PSM, y aún más cuando es automatizada también la generación de un PSM a partir de un PIM.
- **Portabilidad:** La portabilidad en MDA se logra en el nivel del PIM ya que al ser independiente de la plataforma puede ser usado para generar PSMs en diferentes plataformas. Todo lo que se especifique a nivel de PIM es completamente portable.
- **Interoperabilidad:** La interoperabilidad se puede alcanzar dentro de MDA a través de herramientas que no sólo generen PSMs, sino los puentes entre ellos.
- **Mantenimiento y Documentación:** Mucha información sobre la aplicación es incorporada al PIM. Este tiene un alto nivel de abstracción y es una exacta representación del código, por lo tanto completa la función de documentación de alto nivel de un sistema de software.

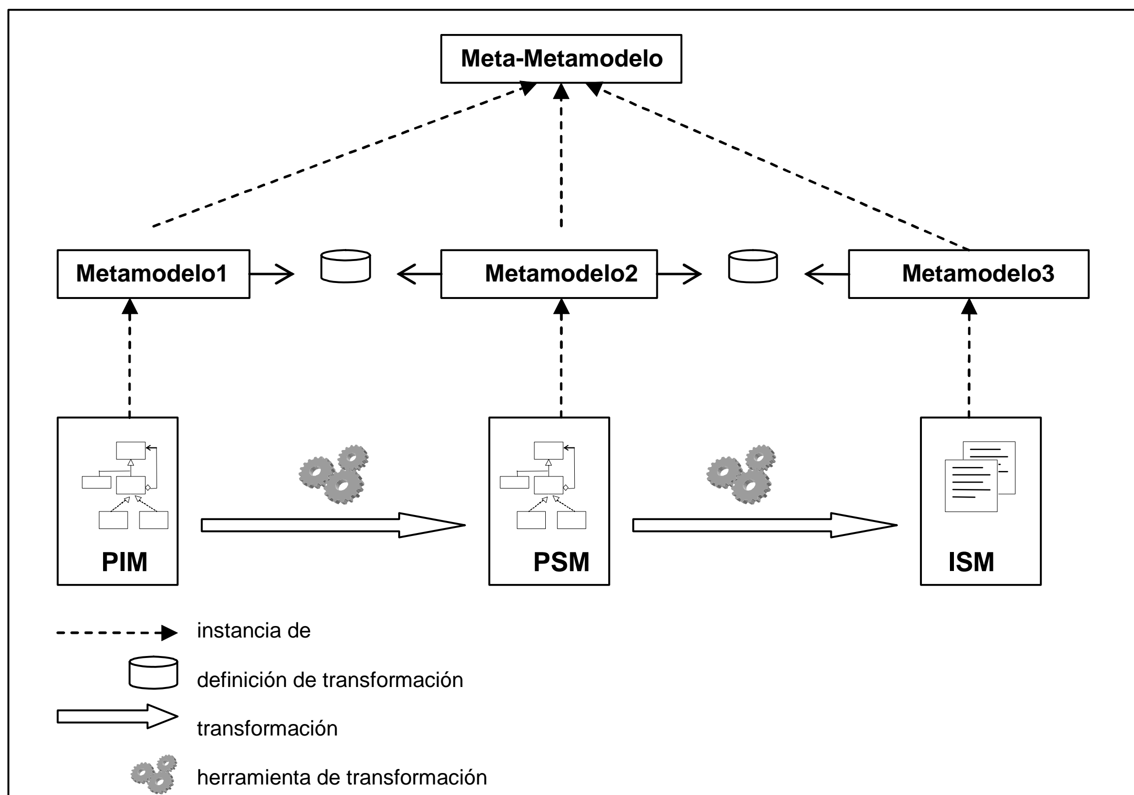



**Figura 3.1.** Framework de MDA

## 3.2. Patrones

### 3.2.1. Introducción

Dentro de la comunidad de orientación a objetos, el concepto de *patrón* representa uno de los enfoques más atractivos para la reutilización de conocimientos de diseño. Se utiliza el término patrón para describir abstracciones de software que son utilizadas por diseñadores y programadores expertos en el desarrollo de sus aplicaciones.

Un criterio de categorización general es el nivel de abstracción en el que el patrón reside. Hay tres grandes categorías de patrones que reflejan diferentes niveles de abstracción:

- En el nivel más alto de abstracción se encuentran los patrones arquitectónicos, los cuales caracterizan la organización global de un sistema. Definen la división de una aplicación en subsistemas y establecen como estos subsistemas colaboran (Shaw y Garlan, 1996; Buschmann, y otros, 1996; Risi y Rossi 2004).
- En un nivel intermedio, se encuentran los patrones de diseño los cuales describen soluciones a problemas de diseño específicos que surgen al refinar un diseño global. Estos patrones pueden ser dependientes o independientes del dominio de aplicación. En el primer caso los patrones de diseño se aplican a problemas de diseño en dominios específicos, tales como aplicaciones gráficas o de hipertexto. Los patrones independientes del dominio proveen soluciones para producir estructuras de diseño flexibles que pueden ser fácilmente adaptadas a nuevos requerimientos (Gamma y otros, 1995).
- En el nivel más bajo de abstracción se encuentran los idiomas que describen como materializar diseños específicos en un determinado lenguaje de programación (Coplien, 1992).

La motivación de usar patrones en sistemas de software es que son elegantes, genéricos, bien probados, simples y reusables. Los beneficios que provee el uso de patrones en el proceso de desarrollo de software incluyen: reutilización de diseño, reutilización potencial de código, mayor comprensión de la organización global de un sistema, y mejor interoperabilidad con otros sistemas por medio de la introducción de estándares. Los patrones permiten establecer un vocabulario común de diseño, cambiando el nivel de abstracción a colaboraciones entre clases y permitiendo comunicar experiencia sobre dichos problemas y soluciones. Son también un gran mecanismo de comunicación para transmitir la experiencia de los ingenieros y diseñadores experimentados a los novatos, convirtiéndose en unas de las vías para la gestión del conocimiento.

### 3.2.2. Patrones de diseño

Los patrones de diseño tuvieron su origen a mediados de la década del noventa, siendo *Design Patterns* (Gamma y otros, 1995) uno de los catálogos de patrones más difundidos. A partir de allí fueron ampliamente aceptados en el área de desarrollo de software tanto en empresas como en el mundo académico.

Gamma y otros (1995) da una definición precisa de lo que es un patrón de diseño y cómo debe ser usado. Es una visión aceptada en la comunidad de la ciencia de la computación. Define los patrones de diseño como:

*“descripciones de objetos que se comunican y clases que son personalizadas para resolver un problema de diseño general en un contexto particular”*.

Los patrones de diseño describen soluciones adecuadas a problemas que ocurren repetidamente, haciendo al diseño más flexible, elegante y principalmente reusable. Esta solución puede ser aplicada repetidamente para producir estructuras de diseño que lucen de forma similar al desarrollar aplicaciones diferentes.

Un patrón de diseño nombra, abstrae e identifica los aspectos claves de una estructura de diseño común que lo hacen útil para crear un diseño orientado a objetos reusable. El patrón de diseño identifica las clases que participan y las instancias, sus roles y colaboraciones, y la distribución de responsabilidades. Cada patrón se enfoca sobre un problema de diseño orientado a objeto particular.

Un patrón de diseño tiene cuatro elementos esenciales:

1. El **nombre**: identifica al patrón, permite describir, en una o dos palabras, un problema de diseño junto con sus soluciones y consecuencias. Dar un nombre a los patrones de diseño incrementa el vocabulario de diseño, lo cual mejora tanto la documentación como la comunicación entre miembros del equipo de desarrollo de software.
2. El **problema**: describe cuando aplicar el patrón, explica el problema y su contexto.
3. La **solución**: describe los elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño o implementación en concreto, sino que es más bien una plantilla que puede aplicarse en muchas situaciones diferentes.
4. Las **consecuencias**: son los resultados, así como ventajas e inconvenientes de aplicar el patrón.

### 3.2.3. Clasificación de patrones de diseño

En Gamma y otros (1995) se presentan 23 patrones de diseño, clasificados según dos criterios: el propósito y el ámbito. La Tabla 3.1 muestra esta clasificación.

El propósito refleja lo que hace el patrón, y los clasifica en:

- **De Creación**: Abstraen el proceso de creación de instancias de objetos. Ayudan a hacer a un sistema independiente de cómo se crean, se componen y se representan sus objetos.
- **Estructurales**: Tratan con la composición de clases u objetos. Se ocupan de cómo las clases y objetos se utilizan para componer estructuras de mayor tamaño.
- **De Comportamiento**: Caracterizan el modo en que las clases y objetos interactúan y se reparten la responsabilidad.

El segundo criterio, el ámbito, especifica si el patrón se aplica a clases u objetos:

- **De Clase**: tratan con relaciones entre clases y sus subclases, establecidas a través de la herencia, son estáticas, fijadas al momento de la compilación.
- **De Objeto**: tratan con relaciones entre objetos, que pueden ser cambiadas a tiempo de ejecución y son más dinámicas.

| | | Propósito | | |
|---|---|---|---|---|
| | | **De Creación** | **Estructurales** | **De Comportamiento** |
| **Ámbito** | **Clase** | Factory Method | Adapter (de clase) | Interpreter<br>Template Method |
| | **Objeto** | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter (de objetos)<br>Bridge<br>Composite<br>Decorator<br>Facade<br>Flyweight<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy<br>Visitor |

**Tabla 3.1.** Clasificación de patrones de diseño.

### 3.2.4. Beneficios de los patrones de diseño

En Beck y otros (1996) se describen distintas experiencias industriales con patrones de diseño y concluye con un conjunto de lecciones aprendidas. Entre otras figuran las siguientes:

- Los patrones de diseño sirven como un buen medio de comunicación entre los miembros de un equipo.
- Capturan las partes esenciales de un diseño en una forma compacta.
- Reducen los tiempos de desarrollo.
- Pueden usarse para registrar y alentar el reuso de buenas prácticas.
- Después de una fase inicial de aprender sobre patrones el esfuerzo de reconocerlos y escribirlos se va reduciendo.

## 3.3. Componentes reusables

Un componente reusable es un elemento de software que puede ser usado por muchas aplicaciones diferentes (Meyer, 1997).

Meyer (2003) pone énfasis en la importancia de la *calidad* en los componentes en el contexto del reuso de software. En este trabajo Meyer introduce el concepto de "*trusted component*" (componente confiable) y lo define como un componente con propiedades de calidad garantizadas y especificadas.

El término *trusted component* no se restringe a código solamente, existe un amplio rango de componentes que se extiende desde de clases orientadas a objetos hasta componentes muy grandes, con la condición que ellos reúnan criterios específicos:

1. Puede ser usado por otros elementos de software, sus "clientes". Puede ser usado por otro software o por personas o sistemas que no son de software. Puede ser usado tanto como un programa o como un componente.
2. Posee una descripción de su uso, que es suficiente para que un cliente lo use. Esto significa el ocultamiento de información y la necesidad de la especificación de la interfaz.

3. No está ligado a ningún conjunto fijo de clientes. Puede ser usado por cualquiera que respete las condiciones de la interfaz.

### 3.3.1. Componentes reusables MDA

Bézivin y otros (2003) definen un componente MDA como “una unidad de encapsulamiento para cualquier artefacto usado o producido dentro de un proceso relacionado con MDA, incluyendo el proceso en si”. Mottu y otros (2006) definen la noción de componente MDA como “compuesto de su especificación, una implementación y un conjunto de casos de test asociados”.

MDA es un framework que promete alto nivel de reuso para desarrollo de software. En lugar de reusar código, MDA propone reusar modelos para el diseño de software situándolos como entidades de primera clase. En este contexto, los modelos son elementos que pueden ser manipulados, almacenados y modificados por herramientas, por lo tanto pueden ser considerados componentes MDA.

Las transformaciones de modelos son entidades importantes para el reuso dentro de MDA. Encapsulan técnicas específicas para manipular y crear modelos, hacen referencia al metamodelo origen y al metamodelo destino y están escritas en un lenguaje dado por lo tanto tienen su propio metamodelo. Pueden considerarse también un componente MDA.

Otro componente MDA es el metamodelo. Éste tiene un nombre, un identificador único, una definición, un propósito, un número de versión, etc.

Otro componente MDA es el *profile* como está definido en UML 2.0 y MOF 2.0.

## 3.4. El lenguaje NEREUS

### 3.4.1. Introducción

NEREUS (Favre, 2005) es un lenguaje algebraico para especificar metamodelos basados en los conceptos de entidad, relaciones y sistemas. En particular, su definición fue inspirada en el contexto de MDA y en su concepción de metamodelado basada en el meta-metamodelo MOF. NEREUS es un lenguaje algebraico alineado con metamodelos MOF: cada construcción en MOF está asociada a una entidad de primera clase en NEREUS, por ejemplo las asociaciones.

La semántica de NEREUS (Favre, 2006; Favre, 2007) fue dada por traducción a CASL (Common Algebraic Specification Language) (Bidoit y Mosses, 2004; Mosses, 2004). CASL está basado en conceptos estándar de especificación algebraica y refleja la consolidación del trabajo de los últimos 20 años en el área. Fue diseñado por el grupo COFI (Common Framework Initiative for Algebraic Specification and Development) con el objetivo de brindar un *framework* unificado para especificación algebraica.

### 3.4.2. Especificaciones básicas

La especificación NEREUS más elemental es la clase que permite declarar *sorts*, operaciones y axiomas en lógica de primer orden. Las clases pueden estructurarse a partir de relaciones de importación, herencia y subtipo. La Figura 3.2 muestra su sintaxis.

Todas las cláusulas son opcionales y no existe un orden entre ellas exceptuando el impuesto por la visibilidad lineal: todo símbolo tiene que ser declarado antes de ser usado.

```
CLASS className
IMPORTS <importList>
IS-SUBTYPE-OF <subtypeList>
INHERITS <inheritList>
GENERATED-BY <constructorList>
DEFERRED
TYPES <sortList>
FUNCTIONS <functionList>
EFFECTIVE
TYPES <sortList>
FUNCTIONS <functionList>
AXIOMS <varList> <axiomList>
END-CLASS
```

**Figura 3.2.** Sintaxis de una clase NEREUS

La cláusula **IMPORTS** expresa relaciones de dependencia. La especificación de la nueva clase está basada en las especificaciones importadas declaradas en <*importList*>.

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 *sorts* y operaciones serán *sorts* y operaciones de la nueva clase.

La cláusula **IS-SUBTYPE-OF** expresa relaciones de herencia por comportamiento. Una noción relacionada a subtipo es la de polimorfismo que satisface que cada objeto de una subclase es también objeto de sus superclases.

NEREUS soporta especificaciones *generated*, la cláusula **GENERATED-BY** lista las operaciones constructoras básicas.

NEREUS distingue partes efectivas y diferidas. La cláusula **DEFERRED** declara *sorts* y operaciones que no están completamente definidos debido a que no hay suficientes ecuaciones para definir el comportamiento de las nuevas operaciones o no hay suficientes operaciones para generar todos los valores de un *sort*. La cláusula **EFFECTIVE** agrega *sorts* y operaciones completamente definidos o completa la definición de algún *sort* u operación definido en forma incompleta en alguna superclase.

En la cláusula **TYPES** se declaran *sorts* de la clase.

En la cláusula **FUNCTIONS** se declaran las funcionalidades de las operaciones con la sintaxis habitual, que admite *place-holders* que expresan donde deben aparecer los argumentos. Es posible definir operaciones en forma parcial. El dominio de definición de una función parcial puede hacerse explícito mediante el uso de aserciones, que deben suceder a la funcionalidad de la operación tras la palabra clave "pre:".

La visibilidad de una operación puede ser pública, protegida o privada y se denota siguiendo la sintaxis UML precediendo al nombre de la operación por los símbolos +, # y – respectivamente. Por defecto la visibilidad es pública. El nombre y la aridad de una operación son usados para determinar una declaración única de una operación.

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.

NEREUS permite definir clases parametrizadas. En el encabezamiento de la clase se declara el nombre de la clase seguido de una lista de parámetros cuyos elementos son de la forma C1: C2, donde C1 es el parámetro formal genérico restringido a una clase existente C2 o a sus subclases (sólo las subclases de C2 serán parámetros actuales válidos).

Pueden agregarse comentarios para dar más claridad a las especificaciones. Un comentario es una línea cuyos dos primeros caracteres son guiones --<texto>.

NEREUS provee un repertorio de tipos primitivos (Boolean, Integer, Char y String) y tipos enumerados.

**Incompletitud en la funcionalidad de operaciones: la notación "_"**

Es común en los primeros niveles de un diseño no poder especificar no sólo las propiedades sino también una funcionalidad completa. La notación *underscore* _ en una funcionalidad de operación indica que ésta es incompleta. Sin embargo a partir de especificaciones con funcionalidades incompletas es posible realizar validaciones o simulaciones del comportamiento de un sistema.

**La notación "*"**

La clase que hereda completa la constructora de sus padres. Para simplificar la notación, un asterisco "*" denota las propiedades heredadas, seguido por los atributos propios, seguido opcionalmente por un guión bajo para denotar la incompletitud.

### 3.4.3. Definición de asociaciones

NEREUS provee mecanismos de extensión que permiten definir nuevas asociaciones como si fuesen primitivas. Provee una jerarquía 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), su navegabilidad (unidireccional o bidireccional) y su conectividad (uno-a-uno, muchos-a-muchos, uno-a-muchos, etc.). La Figura 3.3 grafica parcialmente la jerarquía de constructores de tipos.

Para definir constructores de tipos se especificó un conjunto de esquemas reutilizables que permiten definir relaciones concretas por mecanismos de instanciación. A modo de ejemplo la Figura 3.4 muestra el esquema de la asociación binaria y la Figura 3.5 el esquema de la asociación binaria Bidireccional_1 (uno a uno), la cual corresponde a la asociación cuyos extremos tienen multiplicidad uno.

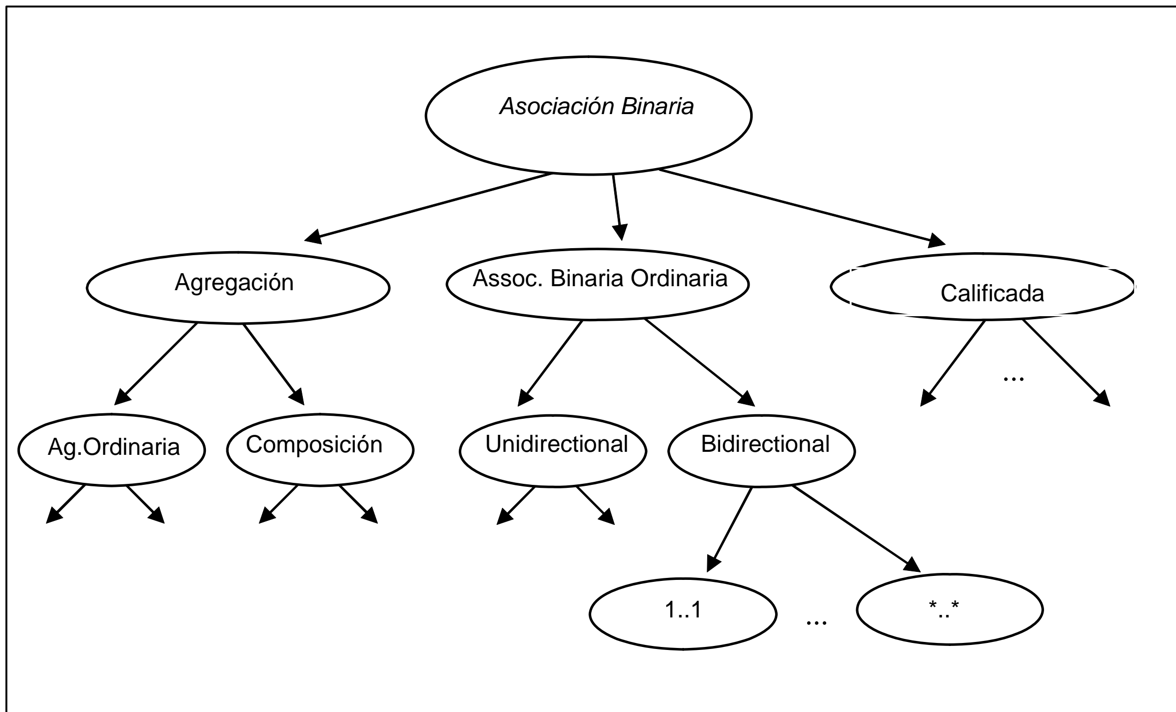



**Figura 3.3.** Jerarquía de constructores de tipos para la asociación binaria

**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**

**Figura 3.4.** Esquema NEREUS de la asociación binaria

```
RELATION SCHEME Bidirectional-1     --Bidirectional one-to-one
IMPORTS Natural
IS-SUBTYPE-OF BinaryAssociation
GENERATED-BY create, addLink
EFFECTIVE
TYPES Bidirectional-1
FUNCTIONS name, getRole1, getRole2, getMult1, getMult2, getVisibility1, getVisibility2,
frozen, changeable, addOnly
create: Typename→ Bidirectional-1
addLink: Bidirectional-1 (b) x Class1 (c1) x Class2 (c2)  → Bidirectional-1
     pre: rightCardinality(b,c1) < 1 and leftCardinality (b,c2) <1 and not isRelated (b,c1,c2)
isRightLinked: Bidirectional-1 x Class1 → Boolean
isLeftLinked: Bidirectional-1 x Class2 → Boolean
isRelated: Bidirectional-1 x Class1  x Class2  → Boolean
isEmpty: Bidirectional-1-> Boolean
rightCardinality: Bidirectional-1 x Class1 → Natural
leftCardinality: Bidirectional-1  x Class2 → Natural
getClass2: Bidirectional-1(a) x Class1(c1) → Class2
     pre: isRightLinked (a,c1)
getClass1: Bidirectional-1(a) x Class2 (c2) →  Class1
     pre: isLeftLinked (a,c2)
remove: Bidirectional-1 (a)  x Class1 (c1)  x Class2 (c2) →  Bidirectional-1
     pre: isRelated (a,c1,c2)
AXIOMS  a: Bidirectional-1 ;  c1, cc1: Class1 ;  c2, cc2: Class2 ;  t: TypeName
name (create(t)) = t
name (add(a,c1,c2)) = name(a)
isEmpty (create(t)) = True
isEmpty (addLink(a,c1,c2)) = False
frozen (a) = <True or False>
changeable (a) = <True or False>
addOnly (a) = <True or False>
getRole1(a) = <role name>
getRole2 (a)  = <role name>
getMult1 (a) = <multiplicity>
getMult2 (a) = <multiplicity>
getVisibility1 (a) = <visibility>
getVisibility2 (a) = <visibility>
isRelated (create(t),c1,c2) = False
isRelated(addLink(a,c1,c2),cc1,cc2) = (c1=cc1 and c2=cc2) or isRelated(a,cc1,cc2)
isRightLinked(create(t),c1)= False
isRightLinked(addLink(a,c1,c2),cc1)= if c1=cc1 then True else isRightLinked(a,cc1)
isLeftLinked(create(t),c2)= False
isLeftLinked(addLink(a,c1,c2),cc2)= if c2=cc2 then True  else isLeftLinked(a,cc2)
rightCardinality(create(t),c1)= 0
rightCardinality(addLink(a,c1,c2),cc1)= if c1=cc1 then 1  else rightCardinality(a,cc1)
leftCardinality(create(t),c1) = 0
leftCardinality(addLink(a,c1,c2),cc1)= if c1=cc1 then 1   else leftCardinality(a,cc1)
getClass1(addLink(a,c1,c2),cc1)= if c1=cc1 then  c2 else getClass1(a,cc1)
getClass2(addLink(a,c1,c2),cc2)=  if c2=cc2 then  c2 else getClass2(a,cc2)
remove(addLink(a,c1,c2),cc1,cc2)=  if (c1=cc1 and c2=cc2)then a else remove(a,cc1,cc2)
END-RELATION
```

**Figura 3.5.** Esquema NEREUS de la asociación bidireccional uno a uno

La siguiente sintaxis permite definir asociaciones:

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

La cláusula **IS** expresa una relación de instanciación del constructor de tipo <*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 *sort* de interés de <*typeConstructorName*> es renombrado por <*relationName*>. Los esquemas están parametrizados en las clases que intervienen, el rol, la visibilidad y multiplicidad de cada *association-end.* La sintaxis para los mismos sigue la de UML: la multiplicidad se instancia por una cadena de la forma *lower-bound..upper-bound* y la visibilidad puede instanciarse por +, - ó #.

La cláusula **CONSTRAINED-BY** posibilita especificar *constraints* en lógica de primer orden que pueden ser aplicados a la asociación.

### 3.4.4. Especificaciones algebraicas modulares

El mecanismo provisto por NEREUS para agrupar clases y relaciones es el PACKAGE:

**PACKAGE** packageName
**IMPORTS** < importList>
**INHERITS** < inheritList>
<elements>
**END-PACKAGE**

*packageName* es el nombre del package, <*importsList*> lista los packages importados, <*inheritList*> lista los packages heredados y <*elements*> son clases, asociaciones y packages.

Existen dos tipos de relaciones que se pueden definir entre packages: importación y generalización. La primera permite importar de un package elementos exportados por otros packages y la relación de herencia permite definir familias de packages. Los packages involucrados en las relaciones de generalización respetan el principio de sustitución como en las relaciones de generalización entre clases.

Un elemento puede pertenecer sólo a un package. Es posible definir la visibilidad de un elemento prefijando su nombre con alguno de los símbolos +, -, #.

Un package permitiría especificar un diagrama de clases. Por ejemplo, es posible agrupar clases y sus correspondientes relaciones a partir del diseño del sistema en una serie de packages.

# 4. Definiendo Componentes MDA

## 4.1. Introducción

MDA promueve el uso de modelos y transformaciones de modelos para el desarrollo de sistemas de software. Un proceso de desarrollo MDA distingue al menos tres clases de modelos: PIM, PSM e ISM. La idea central es la transformación automática de PIMs a PSMs, y de PSMs a ISMs. Las herramientas que llevan a cabo esta automatización se basan en la definición de las transformaciones, las cuales describen como generar un modelo a partir de otro. El éxito de esta propuesta depende de la definición de dichas transformaciones de modelos y de librerías de componentes que tengan un impacto significativo sobre las herramientas que proveen un soporte para MDA. Como consecuencia de estas ideas se diseñó una técnica de metamodelado para alcanzar un alto nivel de reusabilidad y adaptabilidad de componentes MDA (Martinez y Favre, 2006). En particular, en el contexto de esta tesis se describen componentes para patrones de diseño.

La Figura 4.1 muestra un megamodelo que describe la propuesta. Éste define familias de componentes por medio de metamodelos y refinamientos:

- Los metamodelos están definidos en tres niveles de abstracción: PIM, PSM e ISM y se vinculan a través de refinamientos.
- Un refinamiento vertical transforma un modelo origen en un modelo destino, ambos en diferentes niveles de abstracción (PIM a PSM, PSM a ISM). En este contexto un refinamiento es una especificación detallada de un modelo que conforma a otro más abstracto. Está asociado a un modelo origen y a un modelo destino y está compuesto de parámetros, precondiciones y postcondiciones. La precondición establece las condiciones que deben ser cumplidas por el modelo origen para que la transformación sea aplicada. La postcondición establece propiedades en el modelo destino que la transformación garantiza cuando es aplicada.

Las clases *Metamodelo-PIM*, *Metamodelo-PSM* y *Metamodelo-ISM* describen familias de PIMs, PSMs e ISMs respectivamente, mientras que las clases *Refinamiento-PIM-PSM* y *Refinamiento-PSM-ISM* describen una familia de refinamientos entre los metamodelos PIM y PSM y una familia de refinamientos entre los metamodelos PSM e ISM respectivamente. El megamodelo establece que:

- Cada metamodelo origen, instancia de la clase *Metamodelo-PIM*, puede participar en cero o más refinamientos, instancias de la clase *Refinamiento-PIM-PSM*, cada uno de los cuales lo vinculan a un metamodelo destino, instancia de *Metamodelo-PSM*.
- Cada metamodelo instancia de la clase *Metamodelo-PSM*, puede participar en cero o más refinamientos, instancias de la clase *Refinamiento-PSM-ISM*, cada uno de los cuales lo vinculan a un metamodelo destino, instancia de la clase *Metamodelo-ISM*.

La Figura 4.2 muestra una instancia del megamodelo, donde pueden observarse instancias concretas (aparecen subrayadas, siguiendo la notación UML) de los metamodelos del patrón de diseño *Observer*, refinamientos y links entre ellos. Este podría ser visto como un megacomponente que define una familia de componentes reusables que integra instancias de

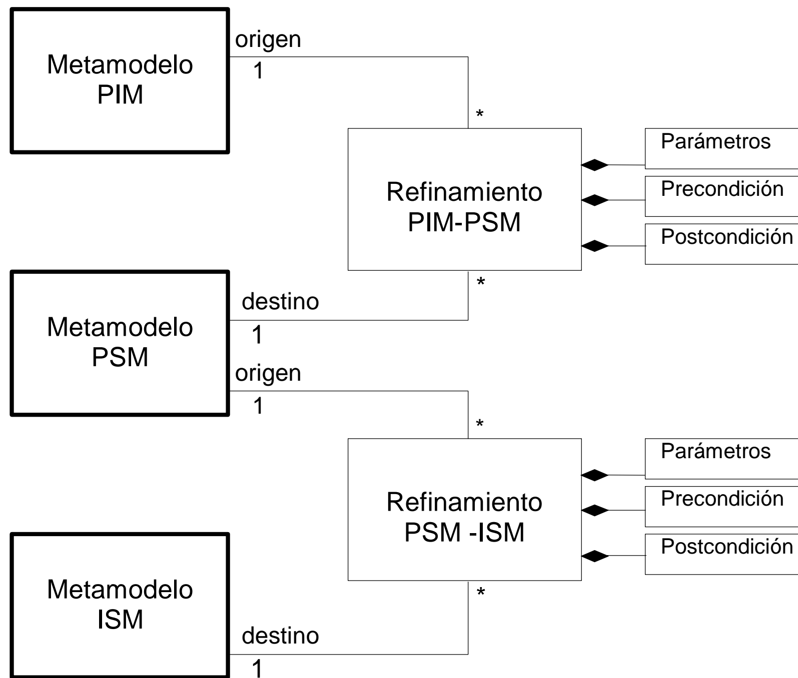



**Figura 4.1.** Un megamodelo para componentes MDA

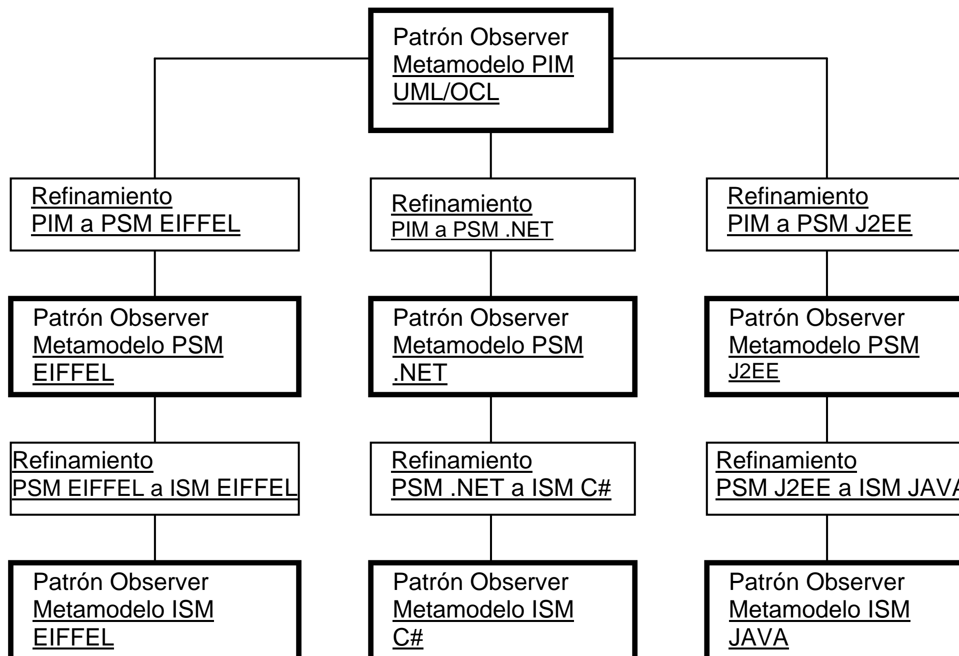



**Figura 4.2.** Una instancia del megamodelo. El Componente Patrón *Observer*

PIM, PSM e ISM. En el primer nivel el metamodelo origen *PIM UML/OCL* del patrón *Observer*, el cual es una instancia de la clase *Metamodelo PIM* del megamodelo, está vinculado a través de los refinamientos *PIM a PSM-EIFFEL*, *PIM a PSM .NET* y *PIM a PSM J2EE,* a tres metamodelos destinos, *PSM-EIFFEL*, *PSM-.NET* y *PSM-J2EE* respectivamente. Estos últimos son instancias de la clase *Metamodelo PSM* del megamodelo. De la misma forma cada metamodelo instancia del *Metamodelo PSM* está vinculado a través de una instancia del *Refinamiento PSM-ISM* a una instancia particular del *Metamodelo ISM*. Por

ejemplo el *Metamodelo PSM-EIFFEL* a través del *Refinamiento-PSM-EIFFEL a ISM EIFFEL*, está vinculado al *Metamodelo ISM-EIFFEL.*

Se describe en este capítulo cómo definir componentes MDA a través de:

- la especificación de metamodelos para patrones de diseño en tres niveles de abstracción (PIM, PSM e ISM) y
- la especificación de transformaciones de modelo a modelo (Refinamientos) basadas en dichos metamodelos.

Si bien se analizaron distintos patrones de diseño, a fin de ilustrar la propuesta se seleccionó el patrón *Observer*.

## 4.2. Especificando metamodelos de patrones de diseño

Para especificar un patrón de diseño en distintos niveles de abstracción, fue necesario construir metamodelos tanto para cada una de las plataformas como para cada lenguaje de programación utilizado. Todos estos metamodelos fueron especificados como una especialización del metamodelo UML mostrado parcialmente el Anexo A. En el Anexo B se muestran en particular los metamodelos correspondientes a las plataformas Eiffel y Java y los metamodelos correspondientes a los lenguajes de programación Eiffel y Java.

Los metamodelos de patrones se definen en tres niveles:

1. *Nivel de modelos independientes de la plataforma*: En este nivel se encuentran los metamodelos de los patrones de diseño, definidos de manera independiente de cualquier plataforma o tecnología específicas. Fueron especificados como una especialización del metamodelo UML.
2. *Nivel específico a la plataforma*: para cada patrón de diseño existen distintos metamodelos, cada uno de ellos corresponde a una plataforma específica. En esta tesis se detallan en particular los metamodelos del patrón de diseño *Observer* para las plataformas Eiffel y Java. Se especificaron como una especialización de los metamodelos específicos a sus plataformas.
3. *Nivel de código*: cada metamodelo en este nivel depende de un lenguaje específico. Se detallan en particular metamodelos para el patrón de diseño *Observer* vinculados a los lenguajes de programación Eiffel y Java. Se especificaron como una especialización de los metamodelos específicos a sus lenguajes.

La definición de los metamodelos de cada patrón de diseño se realizó en base a un análisis de las descripciones de los patrones hechas en distintos catálogos: Gamma y otros (1995), Alpert y otros (1998), Grand (1998), cada uno de los cuales ejemplifica patrones para distintos lenguajes, C++, Smalltalk y Java respectivamente.

Los metamodelos en el nivel independiente de la plataforma se especificaron teniendo en cuenta los siguientes elementos:

- *La estructura*: Se analizaron distintas alternativas para la representación gráfica de las clases en el patrón usando una notación UML. Un metamodelo debe especificar todo el espacio de soluciones de un patrón, es decir las posibles formas en las que el patrón de diseño puede presentarse a nivel de modelos PIM.
- *Los participantes*: Se estudiaron las clases y/u objetos que participan en el patrón de diseño, sus responsabilidades y como se relacionan. Cada elemento, presente en el modelo, debe estar especificado en el metamodelo.

- *Colaboraciones*: Se analizó cómo los participantes colaboran para llevar a cabo sus responsabilidades, lo que permitió determinar qué operaciones deben estar presentes en el patrón y por lo tanto deben estar especificadas en el metamodelo.
- *Ejemplos*: se analizaron distintas aplicaciones de cada patrón y sus distintas variantes. Permitieron chequear los metamodelos.

Cada metamodelo en el nivel específico a la plataforma fue definido teniendo en cuenta:

- *El metamodelo definido en el nivel previo (PIM).*
- *Las características de la plataforma*. Los metamodelos en este nivel fueron definidos teniendo en cuenta una tecnología de implementación específica. Por ejemplo un metamodelo para la plataforma Java restringe la herencia, ya que Java no permite herencia múltiple, mientras que un metamodelo para la plataforma Eiffel no tendrá esta restricción.

Los metamodelos en el nivel de código fueron definidos teniendo en cuenta la gramática de cada lenguaje en particular.

### 4.2.1. Notación de los metamodelos de patrones de diseño

Un modelo UML consiste de un número de diagramas, cada uno de los cuales describe una vista del diseño. En esta tesis, el metamodelo de un patrón es descripto desde la perspectiva estructural (diagrama de clases) y es especificado utilizando la notación UML, de manera semi-formal usando la combinación de notación gráfica, lenguaje natural (español) y lenguaje formal:

- *Sintaxis abstracta*: consiste de uno o más diagramas de clases UML que muestran las metaclases que definen las construcciones y sus relaciones. Las metaclases que aparecen en color gris oscuro corresponden a metaclases propias del metamodelo UML. Las metaclases que aparecen en color gris claro corresponden a las metaclases de los metamodelos específicos a las plataformas utilizadas y a las metaclases correspondientes a los metamodelos específicos a un lenguaje de programación en particular.
- *Descripción de las metaclases*: siguiendo la notación de la superestructura de UML (versión 2.0) se utiliza el lenguaje natural para describir cada metaclase, sus generalizaciones y sus asociaciones y lenguaje formal (OCL) para escribir sus restricciones. Las metaclases de cada uno de los metamodelos descriptos en esta tesis son presentadas en orden alfabético.

### 4.2.2. El componente *Observer*

A continuación se define el componente *Observer* a través de la especificación de los metamodelos de dicho patrón en los tres niveles de abstracción (PIM, PSM e ISM) y refinamientos entre los mismos, como se mencionó previamente. Se detalla el metamodelo de dicho patrón a nivel PIM, dos metamodelos a nivel PSM correspondientes a las plataformas Eiffel y Java y dos metamodelos para el nivel ISM, en particular para los lenguajes de programación Eiffel y Java. Luego se definen las transformaciones PIM-UML a PSM-EIFFEL y PSM-EIFFEL a ISM-EIFFEL. Se comienza con una breve descripción del patrón de diseño *Observer*.

#### 4.2.2.1. Breve descripción del patrón de diseño *Observer*

El patrón de diseño *Observer* "*define una dependencia uno a muchos entre objetos de manera tal que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente*" (Gamma y otros, 1995).

El patrón *Observer* involucra los siguientes participantes claves:

- El *Sujeto*: Puede contener cualquier número de observadores. Mantiene una colección de éstos y tiene la responsabilidad de agregar y eliminar observadores de dicha colección (*attach* y *detach*). Cuando su estado cambia (típicamente los valores de alguno de sus atributos cambia), el *sujeto* notificará a sus observadores (*notify*).
- El *Observador*: Puede observar a uno o más sujetos. Tiene la responsabilidad de actualizarse cuando recibe una notificación de cambio por parte del *Sujeto* (*update*)
- El *Sujeto Concreto*: Guarda el estado de interés para los observadores y envía una notificación de cambio de estado a los mismos.
- El *Observador Concreto*: Mantiene una referencia al *Sujeto Concreto* y mantiene su estado consistente con él (*update*).

**Colaboraciones**

- El sujeto concreto notifica a sus observadores cuando ocurre un cambio que podría hacer que el estado de éstos esté inconsistente con él.
- Después de esto, un objeto observador concreto puede preguntar al sujeto por información, y luego usar esta información para volver a tener su estado consistente con el del sujeto.

**Consecuencias**

El patrón *Observer* permite variar sujetos y observadores independientemente. Se pueden reusar sujetos sin reusar observadores, y viceversa. Permite agregar observadores sin modificar el sujeto u otros observadores.

La Figura 4.4 muestra el diagrama de clases que modela dicho patrón.

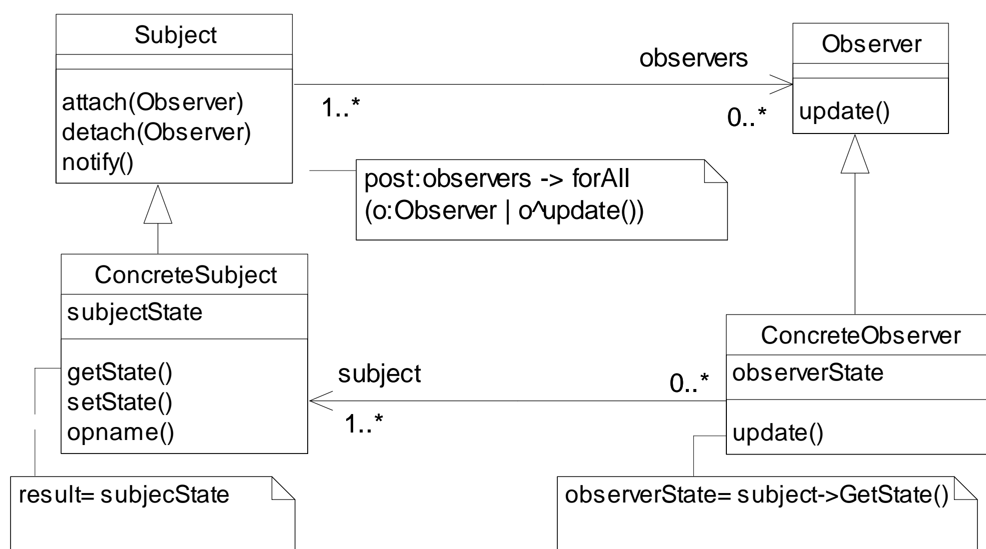



**Figura 4.4.** Modelo del patrón de diseño *Observer*

#### 4.2.2.2. Metamodelo del patrón Observer a nivel PIM

Este metamodelo especifica cada uno de los elementos que participan en un modelo correspondiente al patrón de diseño *Observer*. Es decir, especifica las clases que deben aparecer, sus atributos, sus responsabilidades y cómo se relacionan entre sí. Según la descripción propuesta por Gamma y Otros (1995), en dicho patrón existen cuatro participantes indispensables: el sujeto, el observador, el sujeto concreto y el observador concreto, por lo tanto deben estar especificados en el metamodelo.

Las figuras 4.5.a, 4.5.b, 4.5.c y 4.5.d muestran el metamodelo UML especializado del patrón *Observer*. En éste aparecen los elementos que especifican la vista estructural correspondiente a los diagramas de clases. La Figura 4.5.a muestra las metaclases principales *Subject*, *Observer*, *ConcreteSubject* y *ConcreteObserver* que especifican al sujeto, al observador, al sujeto concreto y al observador concreto respectivamente. También aparecen las metaclases que especifican cómo éstos participantes se relacionan entre sí. Las restantes figuras completan esta vista con la especificación de las operaciones correspondientes a los sujetos y observadores.

El metamodelo establece que el sujeto puede ser una clase o una interfaz. En el caso de ser una clase estará vinculado a cada sujeto concreto a través de una relación de generalización donde el sujeto cumple el rol de padre y el sujeto concreto el rol de hijo en dicha relación. En caso contrario, si el sujeto es una interfaz estará vinculado a un sujeto concreto a través de una relación de realización donde este último implementará el contrato definido por el sujeto abstracto. Las mismas relaciones se establecen para el observador y el observador concreto.

Un sujeto está relacionado con un observador a través de una asociación binaria a la cual se conectan mediante extremos de asociación. En el caso en que el sujeto sea una interfaz esta asociación puede no estar. El sujeto concreto y el observador concreto están vinculados de la misma manera a través de una asociación binaria.

Un Sujeto tendrá como mínimo tres operaciones instancias de las operaciones Attach, Detach y Notify.

Un Observador tendrá como mínimo una instancia de la operación Update.

Un sujeto concreto deberá tener un estado conformado por un atributo o un conjunto de atributos, los cuales serán objeto de observación, y como mínimo operaciones que permitan obtener y modificar sus valores. Tanto los atributos como las operaciones pueden ser propios o heredados.

Las metaclases de color gris, corresponden a clases del metamodelo UML, mientras que las otras corresponden a la especialización del metamodelo UML. Estas últimas se detallan a continuación.

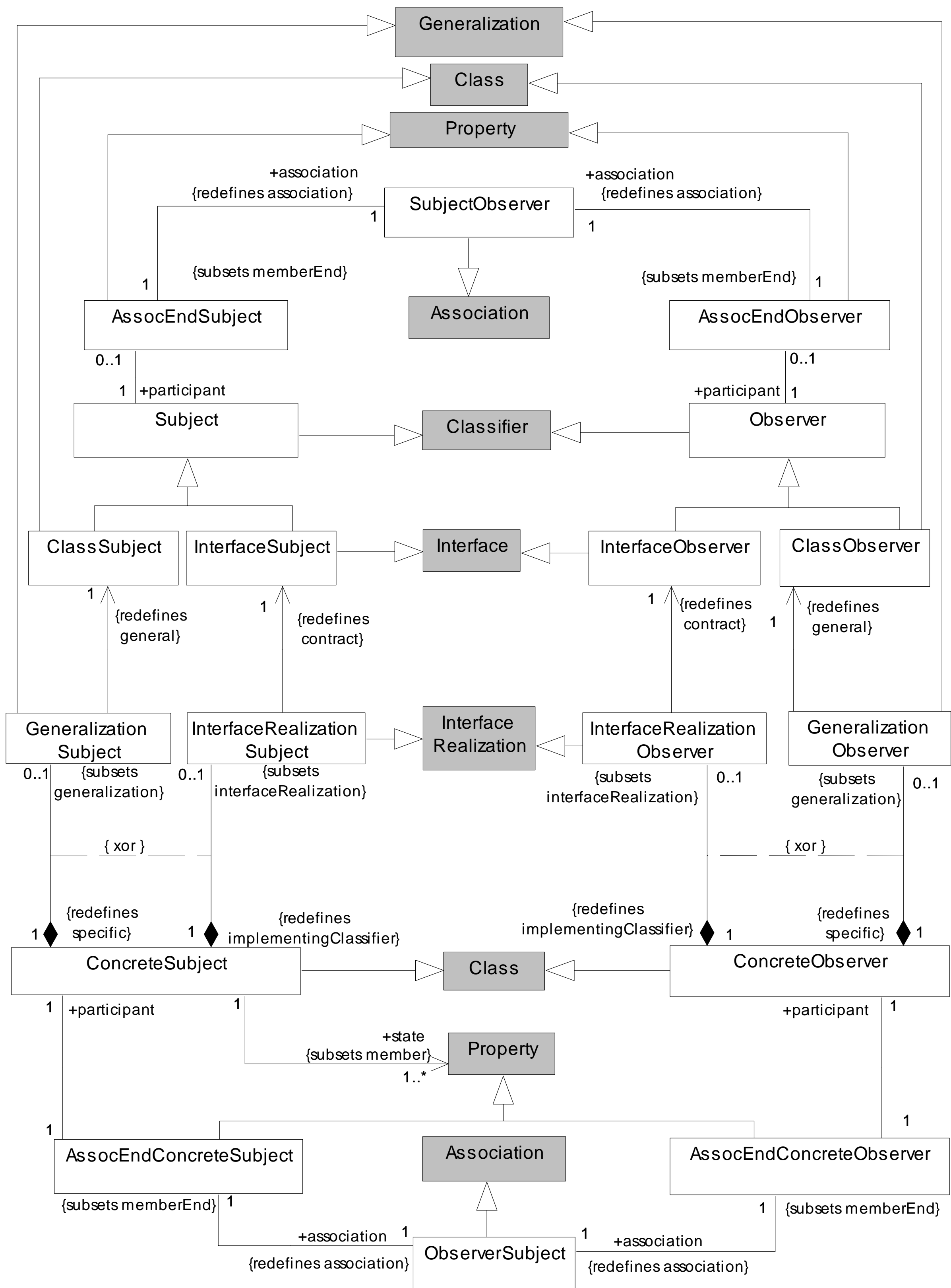
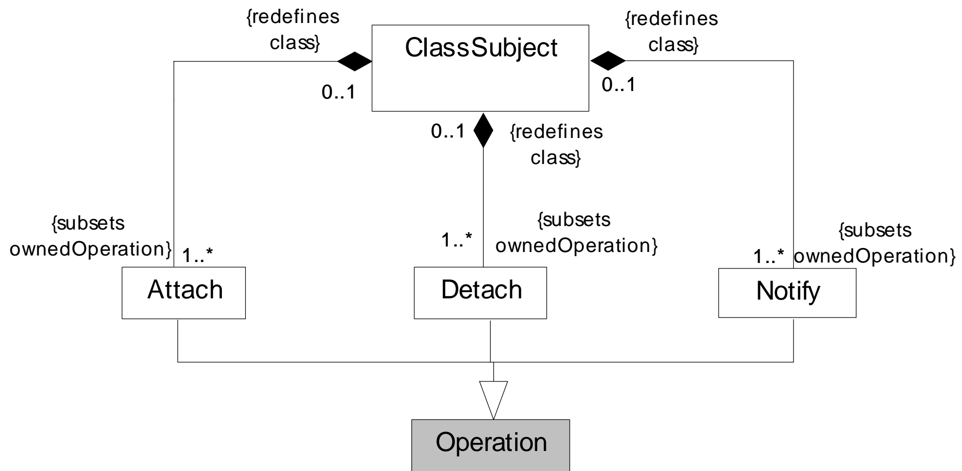



**Figura 4.5.a.** Metamodelo *Observer* - Diagrama de clases



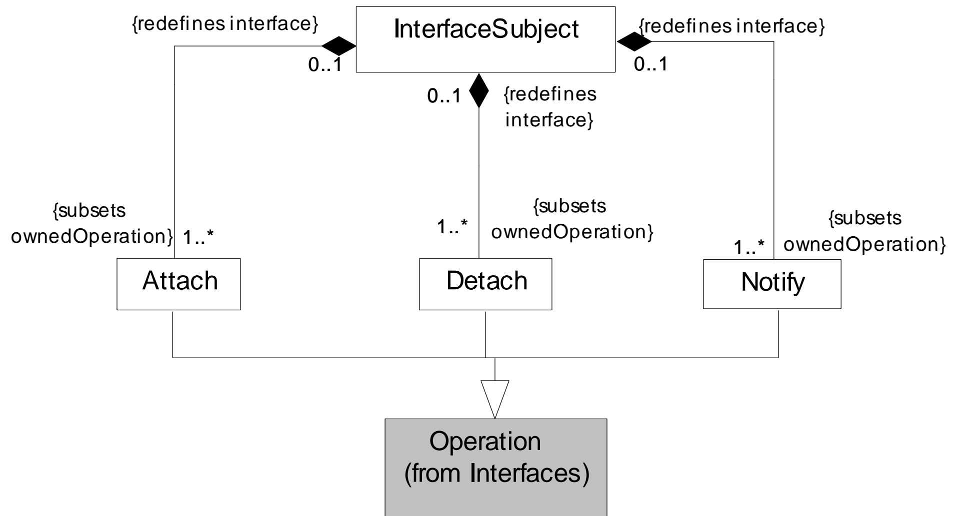




**Figura 4.5.b.** Sujeto abstracto: Operaciones

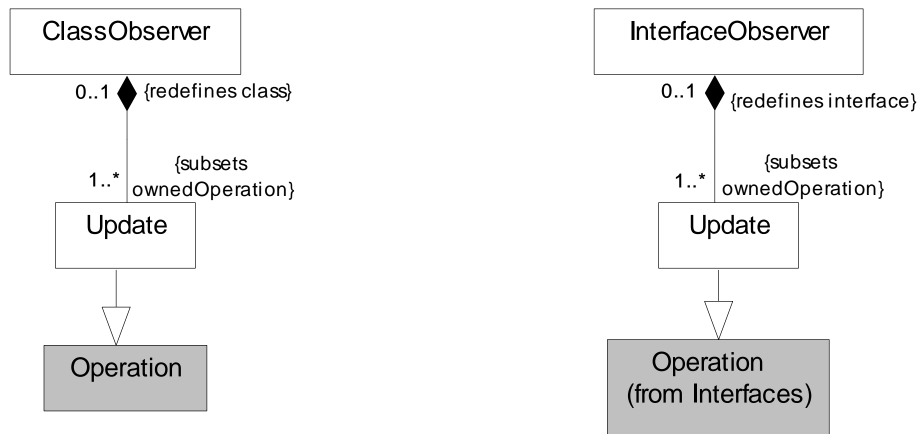



**Figura4.5.c.** Observador abstracto: Operaciones

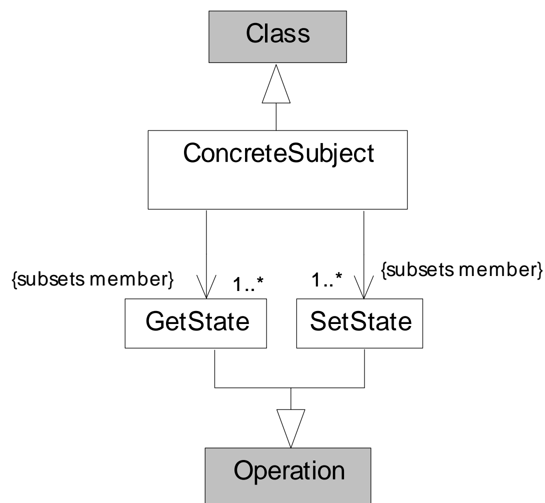



**Figura 4.5.d.** Sujeto concreto: Operaciones

### Descripción de las metaclases

A continuación se presentan en orden alfabético las metaclases del metamodelo del patrón *Observer* a nivel PIM. Para cada una se provee una breve descripción y se detallan las generalizaciones, las asociaciones y las restricciones, estas últimas escritas en OCL y acompañadas de una breve explicación en lenguaje natural.

## AssocEndConcreteObserver

### Generalizaciones

- Property (de Kernel)

### Descripción

Esta propiedad representa el extremo de asociación que conecta una asociación ObserverSubject, de la cual es miembro, con un ConcreteObserver.

### Asociaciones

- participant: ConcreteObserver [1] Designa el clasificador que participa en la asociación.
- association: ObserverSubject [1] Designa la asociación de la cual este extremo de asociación es miembro. Redefine Property::association

### Restricciones

[1] Este extremo de asociación tiene una multiplicidad cuyo rango será un subconjunto de los enteros no negativos. El límite inferior será cero o mayor que cero, y el límite superior será un entero mayor que cero.

```
self.lower >= 0 and self.upper >= 1
```

## AssocEndConcreteSubject

### Generalizaciones

- Property (de Kernel)

### Descripción

Esta propiedad representa el extremo de asociación que conecta una asociación ObserverSubject, de la cual es miembro, con una clase ConcreteSubject.

### Asociaciones

- participant: ConcreteSubject [1] Designa el clasificador que participa en la asociación.
- association: ObserverSubject [1] Designa la asociación de la cual este extremo de asociación es miembro. Redefine Property::association.

### Restricciones

[1] Este extremo de asociación tiene una multiplicidad cuyo rango será un subconjunto de los enteros no negativos.

```
self.lower >= 0  and  self.upper >= 1
```

[2] Deberá ser navegable.

```
self.isNavigable()
```

### Operaciones Adicionales

[1] La operación observadora isNavigable indica si este extremo de asociación es navegable. Es miembro de una asociación binaria, por lo tanto para ser navegable debe ser un extremo propio de una clase.

```
isNavigable = not self.class ->isEmpty()
```

## AssocEndObserver

### Generalizaciones

- Property (de Kernel)

### Descripción

Esta propiedad representa el extremo de asociación que conecta una asociación SubjectObserver, de la cual es miembro, con una clase Observer.

### Asociaciones

- participant: Observer [1] Designa el clasificador que participa en la asociación.
- association: SubjectObserver [1] Designa la asociación de la cual este extremo de asociación es miembro. Redefine Property::association.

### Restricciones

[1] Este extremo de asociación tiene una multiplicidad cuyo rango será un subconjunto de los enteros no negativos.

```
self.lower >= 0  and self.upper >= 1
```

[2] Deberá ser navegable.

```
self.isNavigable()
```

## AssocEndSubject

### Generalizaciones

- Property (de Kernel)

### Descripción

Esta propiedad representa el extremo de asociación que conecta una asociación SubjectObserver, de la cual es miembro, con una clase Subject.

### Asociaciones

- participant: Subject [1] — Designa el clasificador que participa en la asociación.
- association: SubjectObserver [1] — Designa la asociación de la cual este extremo de asociación es miembro. Redefine Property::association.

### Restricciones

[1] Este extremo de asociación tiene una multiplicidad cuyo rango será un subconjunto de los enteros no negativos.

```
self.lower >= 0 and self.upper >= 1
```

## Attach

### Generalizaciones

- Operation (de Kernel, de Interfaces)

### Descripción

Define una operación declarada por un Sujeto.

### Asociaciones

- classSubject: ClassSubject [0..1] — Designa la clase que declara esta operación. Redefine Operation::class.
- interfaceSubject: InterfaceSubject [0..1] — Designa la interfaz que declara esta operación. Redefine Operation::interface.

### Restricciones

[1] Esta operación cambia el estado del sujeto.

```
not self.isQuery
```

[2] Tiene un conjunto no vacío de parámetros y uno de ellos debe ser de entrada (direction= #in) y del tipo Observer.

```
self.ownedParameter ->notEmpty( ) and
self.ownedParameter ->select ( param |   param.direction= #in and
      param.type= oclIsKindOf(Observer)) -> size( ) = 1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## ClassObserver

### Generalizaciones

- Observer, Class (de Kernel)

### Descripción

Una metaclase ClassObserver especifica las características que debe tener una clase que cumpla el rol de observador en el modelo de un patrón *Observer*.

### Asociaciones

- update: Update [1..*] Toda instancia de ClassObserver debe tener al menos una operación instancia de Update. Subconjunto de Class::ownedOperation.

### Restricciones

No hay restricciones adicionales.

## ClassSubject

### Generalizaciones

- Subject, Class (de Kernel)

### Descripción

Esta metaclase especifica las características que debe tener una clase que cumpla el rol de sujeto en el modelo de un patrón *Observer*.

### Asociaciones

- attach: Attach [1..*] Toda instancia de ClassSubject tiene por lo menos una operación instancia de Attach. Subconjunto de Class::ownedOperation.
- detach: Detach [1..*] Toda instancia de ClassSubject tiene por lo menos una operación instancia de Detach. Subconjunto de Class::ownedOperation.
- notify: Notify [1..*] Toda instancia de ClassSubject tiene por lo menos una operación instancia de Notify. Subconjunto de Class::ownedOperation.

### Restricciones

No hay restricciones adicionales.

## ConcreteObserver

### Generalizaciones

- Class (de Kernel)

### Descripción

Esta metaclase especifica las características que debe tener una clase con el comportamiento de un observador concreto en el modelo de un patrón *Observer*.

### Asociaciones

| | |
|---|---|
| • assocEndConcreteObserver: AssocEndConcreteObserver[1] | Denota el extremo de asociación de la asociación ObserverSubject en la cual este clasificador participa. |
| • generalizationObserver: GeneralizationObserver [0..1] | Designa una relación de generalización donde ConcreteObserver cumple el rol de hijo (specific). Subconjunto de Classifier::generalization. |
| • interfaceRealizationObserver: InterfaceRealizationObserver [0..1] | Designa una relación de realización de interfaz donde ConcreteObserver cumple el rol del clasificador que implementa el contrato (implementingClassifier). Subconjunto de BehavioredClassifier::interfaceRealization. |

### Restricciones

[1] Una instancia de un observador concreto debe ser una clase no abstracta.

```
not self.isAbstract
```

[2] Si una instancia de un observador concreto participa en una realización de interfaz, entonces debe ser un BehavioredClassifier.

```
self.interfaceRealizationObserver -> notEmpty ()
   implies  self.oclIsKindOf (BehavioredClassifier) )
```

## ConcreteSubject

### Generalizaciones

- Class (de Kernel)

### Descripción

Esta metaclase especifica las características que debe tener una clase que cumpla el rol de sujeto concreto en el modelo de un patrón *Observer*.

### Asociaciones

| | |
|---|---|
| • assocEndConcreteSubject: AssocEndConcreteSubject [1] | Denota el extremo de asociación de la asociación ObserverSubject en la cual este clasificador participa. |
| • generalizationSubject: GeneralizationSubject [0..1] | Designa una relación de generalización donde ConcreteSubject cumple el rol de hijo (specific). Subconjunto de Classifier::generalización. |
| • getState: GetState [1..*] | Toda instancia de ConcreteSubject debe tener una o más operaciones instancias de GetState. Pueden ser propias o heredadas. Subconjunto de NameSpace::member. |
| • interfaceRealizationSubject: InterfaceRealization [0..1] | Designa una relación de realización de interfaz donde ConcreteSubject cumple el rol del clasificador que implementa el contrato (implementingClassifier). Subconjunto de BehavioredClassifier::InterfaceRealization. |
| • setState: SetState [1..*] | Toda instancia de ConcreteSubject debe tener una o más operaciones instancias de SetState. Pueden ser propias o heredadas. Subconjunto de NameSpace::member. |

- state: Property [1..*] — Especifica un conjunto no vacío de todos los atributos de ConcreteSubject. Pueden ser propios o heredados. Subconjunto de NameSpace::member.

**Restricciones**

[1] Una instancia de un sujeto concreto no debe ser una clase abstracta.

```
not self.isAbstract
```

[2] Si una instancia de un sujeto concreto participa en una realización de interfaz, entonces debe ser un BehavioredClassifier.

```
self.interfaceRealizationSubject -> notEmpty ()
    implies  self.oclIsKindOf ( BehavioredClassifier)
```

[3] state es un conjunto de propiedades que son atributos y no extremos de asociación.

```
self.state->forAll( p | p.association-> isEmpty() )
```

## Detach

**Generalizaciones**

- Operation (de Kernel, de Interfaces)

**Descripción**

Define una operación declarada por un sujeto.

**Asociaciones**

- classSubject: ClassSubject [0..1] — Designa la clase que declara esta operación. Redefine Operation::class.
- interfaceSubject: InterfaceSubject [0..1] — Designa la interfaz que declara esta operación. Redefine Operation::interface.

**Restricciones**

[1] Esta operación cambia el estado del sujeto.

```
not self.isQuery
```

[2] Tiene un conjunto no vacío de parámetros y uno de ellos debe ser de entrada y del tipo *Observer*.

```
self.ownedParameter ->notEmpty( )  and
self.ownedParameter ->select ( param |    param.direction= #in and
                              param.type= oclIsKindOf(Observer)) -> size( )  = 1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## GeneralizationObserver

**Generalizaciones**

- Generalization (de Kernel)

### Descripción

Esta metaclase especifica una relación de generalización entre un observador (ClassObserver) y un observador concreto (ConcreteObserver) en el modelo de un patrón *Observer*.

### Asociaciones

- classObserver: ClassObserver [1] — Designa el elemento general de esta relación. Redefine Generalization::general.
- concreteObserver: ConcreteObserver [1] — Designa el elemento específico de esta relación. Redefine Generalization::specific.

### Restricciones

No hay restricciones adicionales.

## GeneralizationSubject

### Generalizaciones

- Generalization (de Kernel)

### Descripción

Esta metaclase especifica una relación de generalización entre un sujeto (ClassSubject) y un sujeto concreto (ConcreteSubject) en el modelo de un patrón *Observer*.

### Asociaciones

- classSubject: ClassSubject [1] — Designa el elemento general de esta relación. Redefine Generalization::general.
- concreteSubject: ConcreteSubject [1] — Designa el elemento específico de esta relación. Redefine Generalization::specific.

### Restricciones

No hay restricciones adicionales.

## GetState

### Generalizaciones

- Operation (de Kernel)

### Descripción

Define una operación miembro de ConcreteSubject. Especifica un servicio que puede ser requerido desde otro objeto.

### Asociaciones

No tiene asociaciones adicionales.

### Restricciones

[1] Es una operación observadora.

```
self.isQuery
```

[2] Como debe retornar el estado del sujeto, el conjunto de parámetros no debe ser vacío, debe haber por lo menos uno cuya dirección sea *out* o *return*.

```
self.ownedParameter -> notEmpty( ) and
   self.ownedParameter ->select (par |
                          par.direction = #return or par.direction = #out) ->size ( ) >=1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## InterfaceObserver

### Generalizaciones

- Observer, Interface (de Interfaces)

### Descripción

Un InterfaceObserver especifica las características que debe tener una interfaz que cumpla el rol de observador abstracto en el modelo de un patrón *Observer*.

### Asociaciones

- update: Update [1..*] Toda instancia de InterfaceObserver debe tener al menos una operación instancia de Update. Subconjunto de Interface::ownedOperation.

### Restricciones

No hay restricciones adicionales.

## InterfaceSubject

### Generalizaciones

- Subject, Interface (de Interfaces)

### Descripción

Esta metaclase especifica las características que debe tener una interfaz que cumpla el rol de sujeto abstracto en el modelo de un patrón *Observer*.

### Asociaciones

- attach: Attach [1..*] Toda instancia de InterfaceSubject debe tener por lo menos una operación instancia de Attach. Subconjunto de Interface::ownedOperation.
- detach: Detach [1..*] Toda instancia de InterfaceSubject debe tener por lo menos una operación instancia de Detach. Subconjunto de Interface::ownedOperation.
- notify: Notify[1..*] Toda instancia de InterfaceSubject debe tener por lo menos una operación instancia de Notify. Subconjunto de Interface::ownedOperation.

### Restricciones

No hay restricciones adicionales.

## InterfaceRealizationObserver

### Generalizaciones

- InterfaceRealization (de Kernel)

### Descripción

Esta metaclase especifica una relación de realización de interfaz entre un observador abstracto (InterfaceObserver) y un observador concreto (ConcreteObserver) en el modelo de un patrón *Observer*.

### Asociaciones

- concreteObserver: ConcreteObserver [1] — Designa el elemento que implementa el contrato en esta relación. Redefine InterfaceRealization::implementingClassifier.
- interfaceObserver: InterfaceObserver [1] — Designa el elemento que define el contrato en esta relación. Redefine InterfaceRealization::contract.

### Restricciones

No hay restricciones adicionales.

## InterfaceRealizationSubject

### Generalizaciones

- InterfaceRealization (de Kernel)

### Descripción

Esta metaclase especifica una relación de realización de interfaz entre un sujeto abstracto (InterfaceSubject) y un sujeto concreto (ConcreteSubject) en el modelo de un patrón *Observer*.

### Asociaciones

- concreteSubject: ConcreteSubject [1] — Designa el elemento que implementa el contrato en esta relación. Redefine InterfaceRealization::implementingClassifier.
- interfaceSubject: InterfaceSubject [1] — Designa el elemento que define el contrato en esta relación. Redefine InterfaceRealization::contract.

### Restricciones

No hay restricciones adicionales.

## Notify

### Generalizaciones

- Operation (de Kernel, de Interfaces)

### Descripción

Define una operación declarada por un sujeto.

### Asociaciones

- classSubject: ClassSubject [0..1] — Designa la clase que declara esta operación. Redefine Operation::class.
- interfaceSubject: InterfaceSubject [0..1] — Designa la interfaz que declara esta operación. Redefine Operation::interface.

### Restricciones

[1] Es una operación que no cambia el estado del sujeto.

```
self.isQuery
```

[2] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## Observer

### Generalizaciones

- Classifier (de Kernel)

### Descripción

Un Observer es un clasificador especializado que especifica las características que debe tener aquel clasificador que cumpla el rol de observador en el modelo de un patrón *Observer*. Es una metaclase abstracta.

### Asociaciones

- assocEndObserver: AssocEndObserver [0..1] — Denota el extremo de asociación de la asociación SubjectObserver en la cual este clasificador participa.

### Restricciones

No hay restricciones adicionales

## ObserverSubject

### Generalizaciones

- Association (de Kernel)

### Descripción

Esta metaclase especifica una asociación binaria entre dos instancias de Observer y Subject.

### Asociaciones

- assocEndConcreteObserver: AssocEndConcreteObserver [1] — Representa una conexión con el clasificador ConcreteObserver. Subconjunto de Association::memberEnd.
- assocEndConcreteSubject: AssocEndConcreteSubject [1] — Representa una conexión con el clasificador ConcreteSubject. Subconjunto de Association: :memberEnd.

### Restricciones

[1] El número de extremos de asociación miembros es dos, por ser una asociación binaria.

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

## SetState

### Generalizaciones

- Operation (de Kernel)

### Descripción

Define una operación miembro de ConcreteSubject. Especifica un servicio que puede ser requerido desde otro objeto.

### Asociaciones

No tiene asociaciones adicionales.

### Restricciones

[1] No es una operación observadora.

```
not self.isQuery
```

[2] El conjunto de parámetros es no vacío y por lo menos debe haber uno de entrada.

```
self.ownedParameter->notEmpty( ) and
   self.ownedParameter ->select (param | param.direction= #in) ->size( ) >=1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## Subject

### Generalizaciones

- Classifier (de Kernel)

### Descripción

Esta metaclase es un clasificador especializado que especifica las características que debe tener aquella instancia que cumpla el rol de sujeto en el modelo de un patrón *Observer*. Es una metaclase abstracta.

### Asociaciones

- assocEndSubject: AssocEndSubject [0..1] — Denota el extremo de asociación de la asociación SubjectObserver en la cual este clasificador participa.

### Restricciones

No hay restricciones adicionales.

## SubjectObserver

### Generalizaciones

- Association (de Kernel)

### Descripción

Esta metaclase especifica una asociación binaria entre dos clasificadores: Subject y Observer.

### Asociaciones

- assocEndObserver: AssocEndObserver [1] — Representa una conexión con el clasificador Observer. Subconjunto de Association::memberEnd.
- assocEndSubject: AssocEndSubject [1] — Representa una conexión con el clasificador Subject. Subconjunto de Association::memberEnd.

### Restricciones

[1] El número de extremos de asociación miembros es dos por ser una asociación binaria.

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

## Update

### Generalizaciones

- Operation (de Kernel, de Interfaces)

### Descripción

Define una operación declarada por Observer que especifica un servicio que puede ser requerido por otro objeto.

### Asociaciones

- classObserver: ClassObserver [0..1] — Designa la clase que declara esta operación. Redefine Operation::ownedOperation.
- interfaceObserver: InterfaceObserver [0..1] — Designa la interfaz que declara esta operación. Redefine Operation::ownedOperation.

### Restricciones

[1] Es una operación que no cambia el estado del observador.

```
self.isQuery
```

[2] Su visibilidad debe ser pública.

```
self.visibility = #public
```

#### 4.2.2.3. Metamodelo del patrón *Observer* a nivel PSM

El metamodelo del patrón de diseño *Observer* en el nivel PIM está relacionado con distintos metamodelos en el nivel PSM, cada uno correspondiente a una plataforma específica. En particular en esta tesis se especificaron los metamodelos del patrón *Observer* en las plataformas Eiffel y Java, descriptos a continuación.

##### 4.2.2.3.1. Metamodelo del patrón *Observer* para la plataforma Eiffel

La Figura 4.6 muestra el metamodelo del patrón *Observer* para la plataforma Eiffel. En éste aparecen los elementos que especifican la vista estructural correspondiente a los diagramas de clases.

Este metamodelo fue construido especializando el metamodelo PSM-Eiffel (Anexo B). Las metaclases en color gris claro corresponden a este último mientras que las de color gris oscuro corresponden a las metaclases del metamodelo UML.

La principal diferencia con el metamodelo a nivel PIM radica en que en Eiffel no existen las interfaces, estas pueden modelarse como clases abstractas, por lo tanto, las instancias de las metaclases Observer y Subject son clases Eiffel abstractas, y como consecuencia de esto, las relaciones entre las instancias correspondientes a Observer y EffectiveObserver y entre las instancias correspondientes a Subject y EffectiveSubject son generalizaciones.

El metamodelo establece que un sujeto está relacionado con un observador a través de una asociación binaria a la cual se conectan mediante extremos de asociación. De la misma manera el observador efectivo está vinculado a través de una asociación binaria al sujeto efectivo.

Un sujeto tendrá como mínimo tres rutinas instancias de las Attach, Detach y Notify.

Un observador tendrá como mínimo una rutina instancia de Update.

Un sujeto efectivo deberá tener un estado conformado por un atributo o un conjunto de atributos, los cuales serán objeto de observación, y como mínimo operaciones que permitan obtener y modificar sus valores. Tanto los atributos como las rutinas pueden ser propios o heredados.

**Descripción de las metaclases**

## AssocEndEffectiveObserver

### Generalizaciones

- AssociationEnd (de PSM-Eiffel)

### Descripción

Este extremo de asociación conecta una asociación ObserverSubject, de la cual es miembro, con un EffectiveObserver.

### Asociaciones

- association: ObserverSubject [1] — Designa la asociación de la cual este extremo es miembro. Redefine Property::association.
- participant: EffectiveObserver [1] — Designa la clase que participa en la asociación.

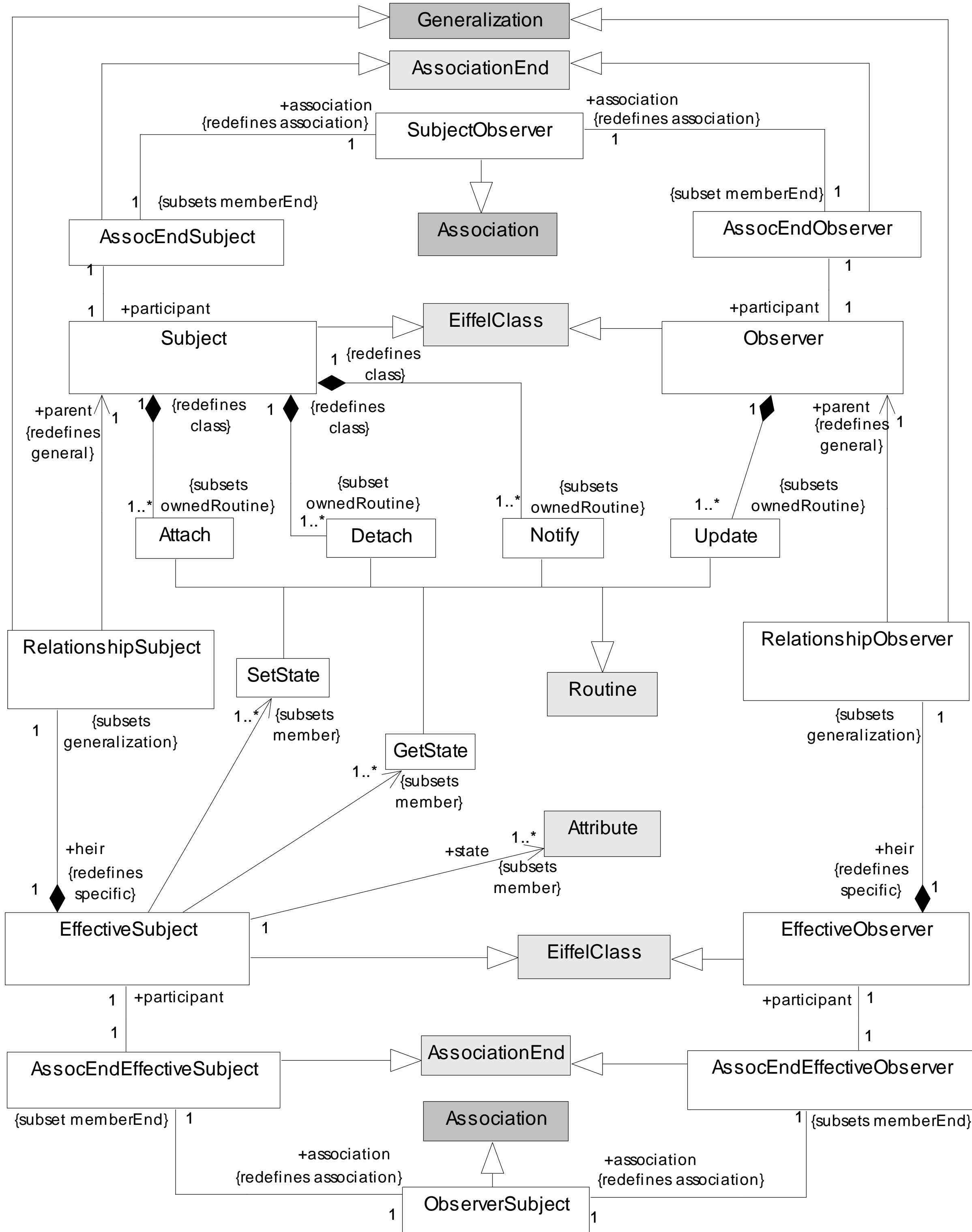



**Figura 4.6.** Metamodelo del patrón *Observer* - Plataforma Eiffel

## Restricciones

[1] Tiene una multiplicidad cuyo rango es un subconjunto de los enteros no negativos. El límite inferior será cero o mayor que cero, y el límite superior será un entero mayor que cero.

self.lower >= 0 **and** self.upper >= 1

## AssocEndEffectiveSubject

### Generalizaciones

- AssociationEnd (de PSM-Eiffel)

### Descripción

Conecta una asociación ObserverSubject, de la cual es miembro, con una clase EffectiveSubject.

### Asociaciones

- association: ObserverSubject [1] Designa la asociación de la cual este extremo es miembro. Redefine Property::association.
- participant: EffectiveSubject [1] Designa la clase que participa en la asociación.

### Restricciones

[1] Tiene una multiplicidad cuyo rango es un subconjunto de los enteros no negativos.

```
self.lower >= 0 and self.upper >=1
```

[2] Deberá ser navegable (los observadores necesitan acceder al sujeto que observan).

```
self.isNavigable()
```

### Operaciones Adicionales

[2] isNavigable indica si este extremo de asociación es navegable. Es miembro de una asociación binaria, entonces para ser navegable debe ser un extremo propio de una clase.

```
isNavigable(): Boolean
isNavigable() = not self.class ->isEmpty()
```

## AssocEndObserver

### Generalizaciones

- AssociationEnd (de PSM-Eiffel)

### Descripción

Conecta una asociación SubjectObserver, de la cual es miembro, con una clase Observer.

### Asociaciones

- association: SubjectObserver [1] Designa la asociación de la cual este extremo es miembro. Redefine Property::association.
- participant: Observer [1] Designa la clase que participa en la asociación.

### Restricciones

[1] Tiene una multiplicidad cuyo rango es un subconjunto de los enteros no negativos.

```
self.lower >= 0 and self.upper >= 1
```

[2] Deberá ser navegable.

```
self.isNavigable()
```

## AssocEndSubject

### Generalizaciones

- AssociationEnd (de PSM-Eiffel)

### Descripción

Conecta una asociación SubjectObserver, de la cual es miembro, con una clase Subject.

### Asociaciones

- association: SubjectObserver [1] Designa la asociación de la cual este extremo es miembro. Redefine Property::association.
- participant: Subject [1] Designa la clase que participa en la asociación.

### Restricciones

[1] Tiene una multiplicidad cuyo rango es un subconjunto de los enteros no negativos.

```
self.lower >= 0 and  self.upper >=1
```

## Attach

### Generalizaciones

- Routine  (de PSM-Eiffel)

### Descripción

Define una rutina declarada por Subject.

### Asociaciones

- subject: Subject [1] Designa la clase que declara esta rutina. Redefine Routine::class.

### Restricciones

[1] Esta rutina cambia el estado del sujeto.

```
not self.isQuery
```

[2] Esta rutina tiene un conjunto no vacío de parámetros y uno de ellos debe ser de entrada (direction= #in) y del tipo  Observer.

```
self.ownedParameter->notEmpty( )  and
self.ownedParameter ->select ( par |    par.direction= #in and
        par.type= oclIsKindOf(Observer)) -> size( ) = 1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## Detach

### Generalizaciones

- Routine  (de PSM-Eiffel)

### Descripción

Define una rutina declarada por Subject.

### Asociaciones

- subject: Subject [1] Designa la clase que declara esta rutina. Redefine Routine::class.

### Restricciones

[1] Esta rutina cambia el estado del sujeto.

```
not self.isQuery
```

[2] Esta tiene un conjunto no vacío de parámetros y uno de ellos debe ser de entrada y del tipo Observer.

```
self.ownedParameter ->notEmpty( ) and
self.ownedParameter ->select (par | par.direction= #in and
                        par.type= oclIsKindOf(Observer)) -> size( ) = 1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## EffectiveObserver

### Generalizaciones

- EiffelClass (de PSM-Eiffel)

### Descripción

Esta metaclase especifica las características que debe tener una clase con el comportamiento de un observador efectivo en el modelo de un patrón *Observer*.

### Asociaciones

- assocEndEffectiveObserver: AssocEndEffectiveObserver[1] Denota el extremo de asociación de la asociación ObserverSubject en la cual este clasificador participa.
- relationshipObserver: RelationshipObserver [1] Designa una relación de generalización donde EffectiveObserver cumple el rol de heredero (heir). Subconjunto de Classifier::generalization.

### Restricciones

[1] Una instancia de un observador efectivo no debe ser una clase diferida.

```
not self.isDeferred
```

## EffectiveSubject

### Generalizaciones

- EiffelClass (de PSM-Eiffel)

### Descripción

Esta metaclase especifica las características que debe tener una clase que cumpla el rol de sujeto efectivo en el modelo de un patrón *Observer*.

### Asociaciones

- assocEndEffectiveSubject: AssocEndEffectiveSubject [1] — Denota el extremo de asociación de la asociación ObserverSubject en la cual este clasificador participa.
- getState: GetState [1..*] — Toda instancia de EffectiveSubject debe tener una o más operaciones instancias de GetState. Pueden ser propias o heredadas. Subconjunto de NameSpace::member.
- relationshipSubject: RelationshipSubject [1] — Designa una relación de generalización donde EffectiveSubject cumple el rol de heredero (heir). Subconjunto de Classifier::generalization.
- setState: SetState [1..*] — Toda instancia de EffectiveSubject debe tener una o más operaciones instancias de SetState. Pueden ser propias o heredadas. Subconjunto de NameSpace::member.
- state: Attribute [1..*] — Especifica un conjunto no vacío de todos los atributos de EffectiveSubject. Pueden ser propios o heredados. Subconjunto de NameSpace::member.

### Restricciones

[1] Una instancia de un sujeto efectivo no debe ser una clase diferida.

**not** self.isDeferred

## GetState

### Generalizaciones

- Routine (de PSM-Eiffel)

### Descripción

Define una rutina miembro de EffectiveSubject. Especifica un servicio que puede ser requerido desde otro objeto.

### Asociaciones

No tiene asociaciones adicionales.

### Restricciones

[1] No es una rutina diferida y no cambia el estado del sujeto.

**not** self.isDeferred **and** self.isQuery

[2] Debe retornar el estado del sujeto, por lo tanto dentro del conjunto de argumentos debe haber por lo menos un parámetro cuya dirección sea *out* o *return*.

self.ownedParameter -> notEmpty( ) **and**
self.ownedParameter ->select (par | par.direction = #return **or** par.direction = #out) ->size()>=1

[3] Su visibilidad debe ser pública.

self.visibility = #public

## Notify

### Generalizaciones

- Routine (de PSM-Eiffel)

### Descripción

Define una rutina declarada por Subject.

### Asociaciones

- Subject: Subject[1] Designa la clase que declara esta rutina. Redefine Routine::class.

### Restricciones

[1] Esta rutina no cambia el estado del sujeto.

self.isQuery

[2] Su visibilidad debe ser pública.

self.visibility = #public

## Observer

### Generalizaciones

- EiffelClass (de PSM-Eiffel)

### Descripción

Esta metaclase especifica las características que debe tener toda clase que cumpla el rol de observador en el modelo de un patrón *Observer* en la plataforma Eiffel.

### Asociaciones

- assocEndObserver: AssocEndObserver [1] Denota el extremo de asociación de la asociación SubjectObserver en la cual esta clase participa.
- update: Update [1..*] Toda instancia de Observer debe tener al menos una rutina instancia de Update. Subconjunto de EiffelClass::ownedRoutine.

### Restricciones

No tiene restricciones adicionales.

## ObserverSubject

### Generalizaciones

- Association (de Kernel)

### Descripción

Esta metaclase especifica una asociación binaria entre instancias de EffectiveObserver y EffectiveSubject.

### Asociaciones

- assocEndEffectiveObserver: AssocEndEffectiveObserver [1] — Representa una conexión con la clase EffectiveObserver. Subconjunto de Association::memberEnd.
- assocEndEffectiveSubject: AssocEndEffectiveSubject [1] — Representa una conexión con la clase EffectiveSubject. Subconjunto de Association::memberEnd.

### Restricciones

[1] El número de extremos de asociación miembros es dos.

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

## RelationshipObserver

### Generalizaciones

- Generalization (de Kernel)

### Descripción

Esta clase especifica la relación de herencia (Generalization) entre un observador (Observer) y un observador efectivo (EffectiveObserver) en el modelo de un patrón *Observer*.

### Asociaciones

- heir: EffectiveObserver [1] — Designa el elemento que cumple el rol de heredero en la relación. Redefine Generalization::specific.
- parent: Observer [1] — Designa el elemento que cumple el rol de padre en la relación. Redefine Generalization::general.

### Restricciones

No hay restricciones adicionales

## RelationshipSubject

### Generalizaciones

- Generalization (de Kernel)

### Descripción

Esta metaclase especifica la relación de herencia (Generalization) entre un sujeto (Subject) y un sujeto efectivo (EffectiveSubject) en el modelo de un patrón *Observer*.

### Asociaciones

- heir: EffectiveSubject [1] — Designa el elemento que cumple el rol de heredero en la relación. Redefine Generalization::specific.
- parent: Subject [1] — Designa el elemento que cumple el rol de padre en la relación. Redefine Generalization::general.

### Restricciones

No hay restricciones adicionales

## SetState

### Generalizaciones

- Routine (de PSM-Eiffel)

### Descripción

Define una rutina miembro de EffectiveSubject. Especifica un servicio que puede ser requerido desde otro objeto.

### Asociaciones

No tiene asociaciones adicionales.

### Restricciones

[1] No es una rutina diferida y modifica el estado del sujeto.

**not** self.isDeferred **and not** self.isQuery

[2] El conjunto de argumentos no es vacío y por lo menos debe haber uno de entrada.

self.ownedParameter ->notEmpty( ) **and**
self.ownedParameter ->select (par | par.direction= #in) ->size( ) >=1

[3] Su visibilidad debe ser pública.

self.visibility = #public

## Subject

### Generalizaciones

- EiffelClass (de PSM-Eiffel)

### Descripción

Esta metaclase especifica las características que debe tener una clase Eiffel cuyo rol sea ser sujeto en el modelo de un patrón *Observer* en la plataforma Eiffel.

### Asociaciones

- assocEndSubject: AssocEndSubject [1] — Denota el extremo de asociación de la asociación SubjectObserver en la cual esta clase participa.
- attach: Attach [1..*] — Toda instancia de Subject debe tener por lo menos una rutina instancia de Attach. Subconjunto de EiffelClass::ownedRoutine.
- detach: Detach [1..*] — Toda instancia de Subject debe tener por lo menos una rutina instancia de Detach. Subconjunto de EiffelClass::ownedRoutine.
- notify: Notify[1..*] — Toda instancia de Subject debe tener por lo menos una rutina instancia de Notify. Subconjunto de EiffelClass::ownedRoutine.

### Restricciones

No tiene restricciones adicionales

## SubjectObserver

### Generalizaciones

- Association (de Kernel)

### Descripción

Esta metaclase especifica una asociación binaria entre instancias de Subject y Observer.

### Asociaciones

- assocEndObserver: AssocEndObserver [1] — Representa una conexión con la clase Observer. Subconjunto de Association::memberEnd.
- assocEndSubject: AssocEndSubject [1] — Representa una conexión con clase Subject. Subconjunto de Association::memberEnd.

### Restricciones

[1] El número de extremos de asociación miembros es dos.

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

## Update

### Generalizaciones

- Routine (de PSM-Eiffel)

### Descripción

Define la rutina declarada por Observer que especifica un servicio que puede ser requerido por otro objeto.

### Asociaciones

- observer: Observer [1] — Designa la clase que declara esta operación. Subconjunto de Routine::ownedRoutine.

### Restricciones

[1] Es una rutina que no cambia el estado del observador.

```
self.isQuery
```

[2] Su visibilidad debe ser pública.

```
self.visibility = #public
```

#### 4.2.2.3.2. Metamodelo del patrón *Observer* para la plataforma Java

Las Figuras 4.7.a, 4.7.b, 4.7.c y 4.7.d muestran el metamodelo UML especializado del patrón *Observer* en la plataforma Java. La Figura 4.7.a muestra las metaclases principales y como se relacionan entre sí. Las restantes figuras completan esta vista con los métodos correspondientes a los sujetos y observadores.

Este metamodelo fue construido especializando el metamodelo PSM-Java (Anexo B). Las metaclases en color gris claro corresponden a este último y las metaclases en color gris oscuro corresponden al metamodelo UML.

La principal diferencia con el metamodelo a nivel PIM radica en que las clases Java no pueden tener herencia múltiple.

El metamodelo establece que el Sujeto puede ser una clase Java o una interfaz Java. En el caso de ser una clase Java estará vinculado a cada sujeto concreto a través de una relación de generalización donde el sujeto cumple el rol de padre y el sujeto concreto el rol de hijo en dicha relación. En caso contrario, si el sujeto es una interfaz estará vinculado a un sujeto concreto a través de una relación de realización donde este último implementará el contrato definido por el sujeto abstracto. Las mismas relaciones se establecen para el observador y el observador concreto.

Un sujeto está relacionado con un observador a través de una asociación binaria a la cual se conectan mediante extremos de asociación. En el caso en que el sujeto sea una interfaz esta asociación puede no estar. El sujeto concreto y el observador concreto están vinculados de la misma manera a través de una asociación binaria.

Un Sujeto tendrá como mínimo tres métodos instancias de Attach, Detach y Notify.

Un Observador tendrá como mínimo una instancia del método Update.

Un sujeto concreto deberá tener un estado conformado por uno o más “field”, objeto de observación, y como mínimo métodos que permitan obtener y modificar sus valores. Tanto los “fields” como los métodos pueden ser propios o heredados.

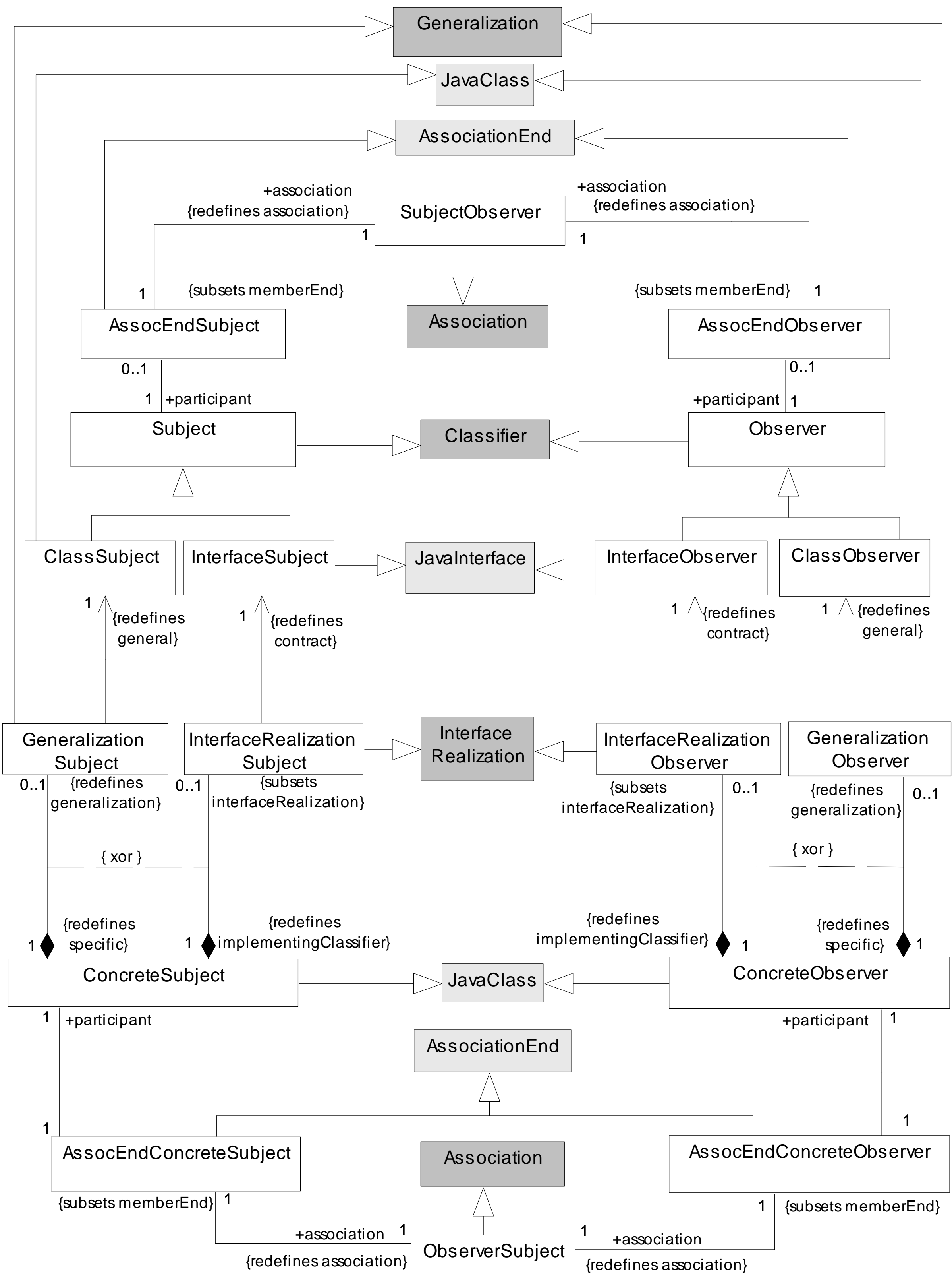
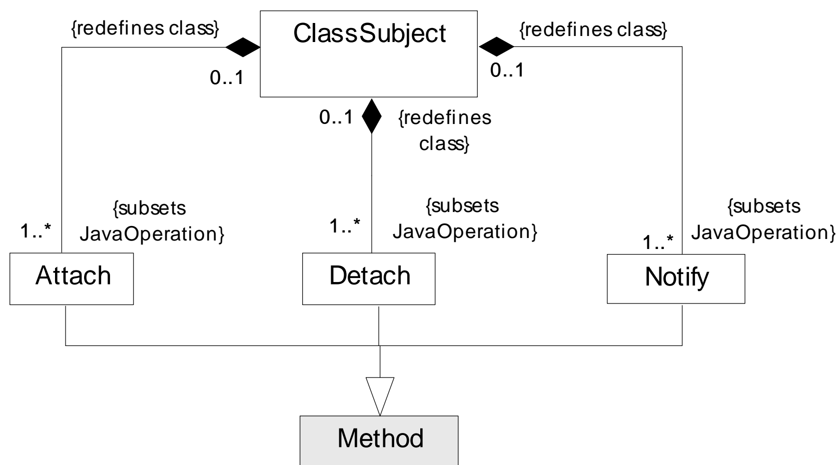



**Figura 4.7.a** Metamodelo del patrón *Observer* - Plataforma Java



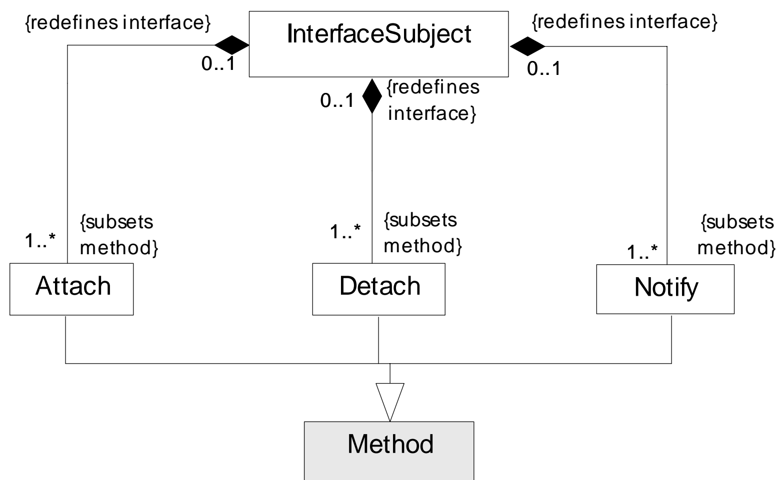




**Figura 4.7.b.** Sujeto Abstracto: Operaciones

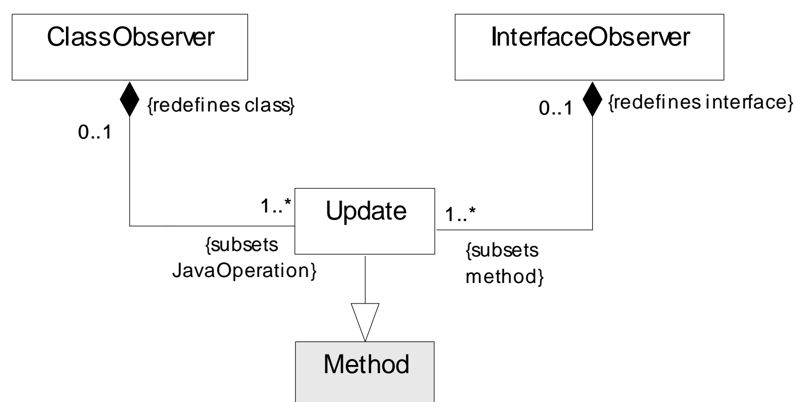



**Figura 4.7.c.** Observador Abstracto: Operaciones

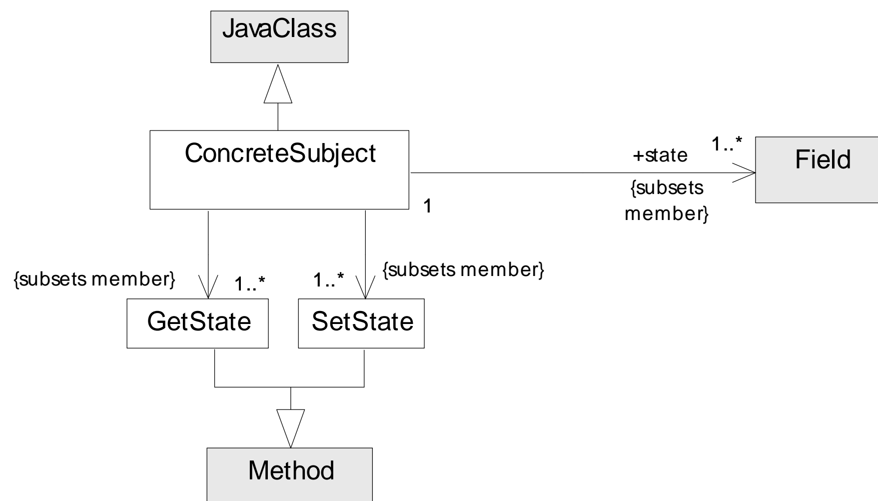



**Figura 4.7.d.** Sujeto concreto: Operaciones y Atributos

### Descripción de las metaclases

## AssocEndConcreteObserver

### Generalizaciones

- AssociationEnd (de PSM-Java)

### Descripción

Este extremo de asociación conecta una asociación ObserverSubject, de la cual es miembro, con un ConcreteObserver.

### Asociaciones

- association: ObserverSubject [1] Designa la asociación de la cual este extremo de asociación es miembro. Redefine Property::association
- participant: ConcreteObserver [1] Designa el clasificador que participa en la asociación.

### Restricciones

[1] Tiene una multiplicidad cuyo rango es un subconjunto de los enteros no negativos.

```
self.lower >= 0 and self.upper > 0
```

## AssocEndConcreteSubject

### Generalizaciones

- AssociationEnd (de PSM-Java)

### Descripción

Este extremo de asociación conecta una asociación ObserverSubject, de la cual es miembro, con una clase ConcreteSubject.

### Asociaciones

- association: ObserverSubject [1] Designa la asociación de la cual este extremo de asociación es miembro. Redefine Property::association.
- participant: Concretesubject [1] Designa el clasificador que participa en la asociación.

### Restricciones

[1] Tiene una multiplicidad cuyo rango es un subconjunto de los enteros no negativos.

```
self.lower >= 0  and self.upper > 0
```

[2] Deberá ser navegable.

```
self.isNavigable()
```

### Operaciones Adicionales

[3] La operación observadora isNavigable indica si este extremo de asociación es navegable. Es miembro de una asociación binaria, por lo tanto para ser navegable debe ser un extremo propio de una clase.

```
isNavigable(): Boolean
isNavigable() = not self.class ->isEmpty()
```

## AssocEndObserver

### Generalizaciones

- AssociationEnd (de PSM-Java)

### Descripción

Conecta una asociación SubjectObserver, de la cual es miembro, con una clase Observer.

### Asociaciones

- association: SubjectObserver [1] Designa la asociación de la cual este extremo de asociación es miembro. Redefine Property::association.
- participant: Observer [1] Designa el clasificador que participa en la asociación.

### Restricciones

[1] Tiene una multiplicidad cuyo rango es un subconjunto de los enteros no negativos.

```
self.lower >= 0  and  self.upper > 0
```

[2] Deberá ser navegable.

```
self.isNavigable()
```

## AssocEndSubject

### Generalizaciones

- AssociationEnd (de PSM-Java)

### Descripción

Conecta una asociación SubjectObserver, de la cual es miembro, con una clase Subject.

### Asociaciones

- association: SubjectObserver [1] — Designa la asociación de la cual este extremo de asociación es miembro. Redefine Property::association.
- participant: Subject [1] — Designa el clasificador que participa en la asociación.

### Restricciones

[1] Tiene una multiplicidad cuyo rango es un subconjunto de los enteros no negativos.

```
self.lower >= 0 and self.upper > 0
```

## Attach

### Generalizaciones

- Method (de PSM-Java)

### Descripción

Define un método declarado por un Sujeto.

### Asociaciones

- classSubject: ClassSubject [0..1] — Designa la clase que declara este método. Redefine JavaOperation::class.
- interfaceSubject: InterfaceSubject [0..1] — Designa la interfaz que declara este método. Redefine Method::interface.

### Restricciones

[1] Este método cambia el estado del sujeto.

```
not self.isQuery
```

[2] Este método tiene un conjunto no vacío de parámetros y uno de ellos debe ser de entrada (direction= #in) y del tipo Observer.

```
self.ownedParameter->notEmpty( ) and
self. ownedParameter ->select ( param |    param.direction= #in and
      param.type= oclIsKindOf(Observer)) -> size( ) = 1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## ClassObserver

### Generalizaciones

- Observer, JavaClass (de PSM-Java)

### Descripción

Una metaclase ClassObserver especifica las características que debe tener una clase Java que cumpla el rol de observador en el modelo de un patrón *Observer*.

### Asociaciones

- update: Update [1..*] — Toda instancia de ClassObserver debe tener al menos un método instancia de Update. Subconjunto de JavaClass::javaOperation.

### Restricciones

No hay restricciones adicionales.

## ClassSubject

### Generalizaciones

- Subject, JavaClass (de PSM-Java)

### Descripción

Esta metaclase especifica las características que debe tener una clase Java que cumpla el rol de sujeto en el modelo de un patrón *Observer*.

### Asociaciones

- attach: Attach [1..*] — Toda instancia de ClassSubject tiene por lo menos un método instancia de Attach. Subconjunto de JavaClass:: javaOperation.
- detach: Detach [1..*] — Toda instancia de ClassSubject tiene por lo menos un método instancia de Detach. Subconjunto de JavaClass:: javaOperation.
- notify: Notify[1..*] — Toda instancia de ClassSubject tiene por lo menos un método instancia de Notify. Subconjunto de JavaClass:: javaOperation.

### Restricciones

No hay restricciones adicionales.

## ConcreteObserver

### Generalizaciones

- JavaClass (de PSM-Java)

### Descripción

Esta metaclase especifica las características que debe tener una clase Java con el comportamiento de un observador concreto en el modelo de un patrón *Observer*.

### Asociaciones

- assocEndConcreteObserver: AssocEndConcreteObserver[1] — Denota el extremo de asociación de la asociación ObserverSubject en la cual este clasificador participa.
- generalizationObserver: GeneralizationObserver [0..1] — Designa una relación de generalización donde ConcreteObserver cumple el rol de hijo (specific). Redefine Classifier::generalization.
- interfaceRealizationObserver: InterfaceRealizationObserver [0..1] — Designa una relación de realización de interfaz donde ConcreteObserver cumple el rol del clasificador que implementa el contrato (implementingClassifier). Subconjunto de BehavioredClassifier::interfaceRealization.

### Restricciones

[1] Una instancia de un observador concreto no puede ser una clase abstracta.

```
not self.isAbstract
```

[2] Si una instancia de un observador concreto participa en una realización de interfaz, entonces debe ser un BehavioredClassifier.

```
self.interfaceRealizationObserver -> notEmpty () implies self.oclIsKindOf (BehavioredClassifier) )
```

## ConcreteSubject

### Generalizaciones

- JavaClass (de PSM-Java)

### Descripción

Esta metaclase especifica las características que debe tener una clase que cumpla el rol de sujeto concreto en el modelo de un patrón *Observer*.

### Asociaciones

- assocEndConcreteSubject: AssocEndConcreteSubject [1] — Denota el extremo de asociación de la asociación ObserverSubject en la cual este clasificador participa.
- generalizationSubject: GeneralizationSubject [0..1] — Designa una relación de generalización donde ConcreteSubject cumple el rol de hijo (specific). Redefine Classifier::generalization.
- getState: GetState [1..*] — Toda instancia de ConcreteSubject debe tener uno o más métodos instancias de GetState. Pueden ser propios o heredados. Subconjunto de NameSpace::member.
- interfaceRealizationSubject: InterfaceRealizationSubject [0..1] — Designa una relación de realización de interfaz donde ConcreteSubject cumple el rol del clasificador que implementa el contrato (implementingClassifier). Subconjunto de BehavioredClassifier::interfaceRealization.
- setState: GetState [1..*] — Toda instancia de ConcreteSubject debe tener uno o más métodos instancias de SetState. Pueden ser propios o heredados. Subconjunto de NameSpace::member.
- state: Field [1..*] — Especifica un conjunto no vacío de todos los atributos de ConcreteSubject. Pueden ser propios o heredados. Subconjunto de NameSpace::member.

### Restricciones

[1] Una instancia de un sujeto concreto no puede ser una clase abstracta.

```
not self.isAbstract
```

[2] Si una instancia de un sujeto concreto participa en una realización de interfaz, entonces debe ser un BehavioredClassifier.

```
self.interfaceRealizationSubject -> notEmpty () implies self.oclIsKindOf (BehavioredClassifier) )
```

## Detach

### Generalizaciones

- Method (de PSM-Java)

### Descripción

Define un método declarado por un sujeto.

### Asociaciones

- classSubject: ClassSubject [0..1] — Designa la clase que declara este método. Redefine JavaOperation::class.
- interfaceSubject: InterfaceSubject [0..1] — Designa la interfaz que declara este método. Redefine Method::interface.

### Restricciones

[1] Este método cambia el estado del sujeto.

```
not self.isQuery
```

[2] Este método tiene un conjunto no vacío de parámetros y uno de ellos debe ser de entrada y del tipo Observer.

```
self.ownedParameter ->notEmpty( )  and
self.ownedParameter ->select ( param |    param.direction= #in and
                                param.type= oclIsKindOf(Observer)) -> size( )  = 1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## GeneralizationObserver

### Generalizaciones

- Generalization (de Kernel)

### Descripción

Esta metaclase especifica una relación de generalización entre un observador (ClassObserver) y un observador concreto (ConcreteObserver) en el modelo de un patrón *Observer*.

### Asociaciones

- classObserver: ClassObserver [1] — Designa el elemento general de esta relación. Redefine Generalization::general.
- concreteObserver: ConcreteObserver [1] — Designa el elemento específico de esta relación. Redefine Generalization::specific.

### Restricciones

No hay restricciones adicionales.

## GeneralizationSubject

### Generalizaciones

- Generalization (de Kernel)

### Descripción

Esta metaclase especifica una relación de generalización entre un sujeto (ClassSubject) y un sujeto concreto (ConcreteSubject) en el modelo de un patrón *Observer*.

### Asociaciones

- classSubject: ClassSubject [1] — Designa el elemento general de esta relación. Redefine Generalization::general.
- concreteSubject: ConcreteSubject [1] — Designa el elemento específico de esta relación. Redefine Generalization::specific.

### Restricciones

No hay restricciones adicionales.

## GetState

### Generalizaciones

- Method (de PSM-Java)

### Descripción

Define un método miembro de ConcretetSubject. Especifica un servicio que puede ser requerido desde otro objeto.

### Asociaciones

No tiene asociaciones adicionales.

### Restricciones

[1] Es un método observador y no abstracto.

```
self.isQuery and not self.isAbstract
```

[2] Como debe retornar el estado del sujeto, el conjunto de parámetros no debe ser vacío, debe haber por lo menos uno cuya dirección sea *out* o *return*.

```
self.ownedParameter -> notEmpty( ) and
   self.ownedParameter->select (par | par.direction= #return or par.direction = #out) ->size () >=1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## InterfaceObserver

### Generalizaciones

- Observer, JavaInterface (de PSM-Java)

### Descripción

La metaclase InterfaceObserver especifica las características que debe tener una interfaz Java que cumpla el rol de observador abstracto en el modelo de un patrón *Observer*.

### Asociaciones

- update: Update [1..*] Toda instancia de InterfaceObserver debe tener al menos una operación instancia de Update. Subconjunto de JavaInterface::method.

### Restricciones

No hay restricciones adicionales.

## InterfaceSubject

### Generalizaciones

- Subject, JavaInterface (de PSM-Java)

### Descripción

Esta metaclase especifica las características que debe tener una interfaz Java que cumpla el rol de sujeto abstracto en el modelo de un patrón *Observer*.

### Asociaciones

- attach: Attach [1..*] Toda instancia de InterfaceSubject debe tener por lo menos un método instancia de Attach. Subconjunto de JavaInterface::method.
- detach: Detach [1..*] Toda instancia de InterfaceSubject debe tener por lo menos un método instancia de Detach. Subconjunto de JavaInterface::method.
- notify: Notify [1..*] Toda instancia de InterfaceSubject debe tener por lo menos un método instancia de Notify. Subconjunto de JavaInterface::method.

### Restricciones

No hay restricciones adicionales.

## InterfaceRealizationObserver

### Generalizaciones

- InterfaceRealization (de Kernel)

### Descripción

Esta metaclase especifica una relación de realización de interfaz entre un observador abstracto (InterfaceObserver) y un observador concreto (ConcreteObserver) en el modelo de un patrón *Observer*.

### Asociaciones

- concreteObserver: ConcreteObserver [1] — Designa el elemento que implementa el contrato en esta relación. Redefine InterfaceRealization::implementingClassifier.
- interfaceObserver: InterfaceObserver [1] — Designa el elemento que define el contrato en esta relación. Redefine InterfaceRealization::contract.

### Restricciones

No hay restricciones adicionales.

## InterfaceRealizationSubject

### Generalizaciones

- InterfaceRealization (de Kernel)

### Descripción

Esta metaclase especifica una relación de realización de interfaz entre un sujeto abstracto (InterfaceSubject) y un sujeto concreto (ConcreteSubject) en el modelo de un patrón *Observer*.

### Asociaciones

- concreteSubject: ConcreteSubject [1] — Designa el elemento que implementa el contrato en esta relación. Redefine InterfaceRealization::implementingClassifier.
- interfaceSubject: InterfaceSubject [1] — Designa el elemento que define el contrato en esta relación. Redefine InterfaceRealization::contract.

### Restricciones

No hay restricciones adicionales.

## Notify

### Generalizaciones

- Method (de PSM-Java)

### Descripción

Define un método declarado por un sujeto.

### Asociaciones

- classSubject: ClassSubject [0..1] — Designa la clase que declara este método. Redefine JavaOperation::class.
- interfaceSubject: InterfaceSubject [0..1] — Designa la interfaz que declara este método. Redefine Method::interface.

### Restricciones

[1] Es un método que no cambia el estado del sujeto.

```
self.isQuery
```

[2] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## Observer

### Generalizaciones

- Classifier (de Kernel)

### Descripción

Un Observer es un clasificador especializado que especifica las características que debe tener aquel clasificador que cumpla el rol de observador en el modelo de un patrón *Observer*. Es una metaclase abstracta.

### Asociaciones

- assocEndObserver: AssocEndObserver [0..1] — Denota el extremo de asociación de la asociación SubjectObserver en la cual este clasificador participa.

### Restricciones

No hay restricciones adicionales

## ObserverSubject

### Generalizaciones

- Association (de Kernel)

### Descripción

Esta metaclase especifica una asociación binaria entre dos instancias de Observer y Subject.

### Asociaciones

- assocEndConcreteObserver: AssocEndConcreteObserver [1] — Representa una conexión con el clasificador ConcreteObserver. Subconjunto de Association::memberEnd.
- assocEndConcreteSubject: AssocEndConcreteSubject [1] — Representa una conexión con el clasificador ConcreteSubject. Subconjunto de Association::memberEnd.

### Restricciones

[1] El número de extremos de asociación miembros es dos.

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

## SetState

### Generalizaciones

- Method (de PSM-Java)

### Descripción

Define una operación miembro de ConcreteSubject. Especifica un servicio que puede ser requerido desde otro objeto.

### Asociaciones

No tiene asociaciones adicionales.

### Restricciones

[1] Es un método que no es abstracto y que modifica el estado del sujeto.

```
not self.isAbstract and not self.isQuery
```

[2] El conjunto de parámetros es no vacío y por lo menos debe haber uno de entrada.

```
self.OwnedParameter->notEmpty( ) and
   self.OwnedParameter ->select (param | param.direction= #in) ->size( ) >=1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## Subject

### Generalizaciones

- Classifier (de Kernel)

### Descripción

Esta metaclase es un clasificador especializado que especifica las características que debe tener aquella instancia que cumpla el rol de sujeto en el modelo de un patrón *Observer*. Es una metaclase abstracta.

### Asociaciones

- assocEndSubject: AssocEndSubject [0..1] Denota el extremo de asociación de la asociación SubjectObserver en la cual este clasificador participa.

### Restricciones

No hay restricciones adicionales.

## SubjectObserver

### Generalizaciones

- Association (de Kernel)

### Descripción

Esta metaclase especifica una asociación binaria entre dos clasificadores: Subject y Observer.

**Asociaciones**

- assocEndObserver: AssocEndObserver [1] — Representa una conexión con el clasificador Observer. Subconjunto de Association::memberEnd.
- assocEndSubject: AssocEndSubject [1] — Representa una conexión con el clasificador Subject. Subconjunto de Association::memberEnd.

**Restricciones**

[1] El número de extremos de asociación miembros es dos.

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

## Update

**Generalizaciones**

- Method (de PSM-Java)

**Descripción**

Define un método declarado por un Observer que especifica un servicio que puede ser requerido por otro objeto.

**Asociaciones**

- classObserver: ClassObserver [0..1] — Designa la clase que declara esta operación. Redefine JavaOperation::class.
- interfaceObserver: InterfaceObserver [0..1] — Designa la interfaz que declara esta operación. Redefine Method::interface..

**Restricciones**

[1] Es un método que no cambia el estado del observador.

```
self.isQuery
```

[2] Su visibilidad debe ser pública.

```
self.visibility = #public
```

#### 4.2.2.4. Metamodelo del patrón *Observer* a nivel ISM

En este nivel se describen los metamodelos del patrón de diseño *Observer* específicos a las implementaciones Eiffel y Java.

##### 4.2.2.4.1. Metamodelo del patrón *Observer* específico a la implementación Eiffel

Las Figuras 4.8.a y 4.8.b muestran el metamodelo del patrón de diseño *Observer* a nivel ISM para Eiffel. Fue construido en base al metamodelo ISM-Eiffel (Anexo B) teniendo en cuenta las ideas mencionadas en Gamma y otros (1995) relacionadas a la implementación de este patrón de diseño. Las metaclases en color gris oscuro corresponden a las metaclases del metamodelo UML mientras que las de color gris claro corresponden a las del metamodelo ISM-Eiffel.

La principal diferencia con el metamodelo del patrón *Observer* a nivel PSM específico a la plataforma Eiffel es que a nivel de código no existen los extremos de asociación y aparecen detalles de implementación.

La metaclase Subject especifica una clase Eiffel que además de tener al menos tres rutinas, instancias de Attach, Detach y Notify, contendrá una referencia a sus observadores a través de un atributo, el cual puede ser una instancia de ObserverReference o una instancia de SubjectObserverReference. En el primer caso, el atributo referencia a la colección de observadores, en el segundo caso referencia a la clase intermedia encargada de mantener la relación entre sujetos y observadores en el caso que dicha relación sea compleja.

La metaclase Observer especifica una clase Eiffel que tendrá al menos una rutina instancia de Update.

EffectiveSubject especifica una clase Eiffel que tiene un estado conformado por un atributo o un conjunto de atributos, los cuales serán objeto de observación, y como mínimo rutinas que permitan obtener y modificar sus valores. Tanto los atributos como las rutinas pueden ser propios o heredados. Toda instancia de EffectiveSubject será subclase de una instancia de Subject.

EffectiveObserver especifica una clase Eiffel que tendrá una referencia al sujeto o sujetos que observa a través de un atributo instancia de SubjectReference. Toda instancia de EffectiveObserver será subclase de una instancia de Observer.

ObserverReference especifica una referencia a una colección de observadores de un sujeto.

SubjectReference especifica una referencia a un sujeto o a una colección de sujetos.

SubjectObserverAssociation especifica una clase Eiffel encargada de mantener la relación entre un sujeto y sus observadores. Esta clase tendrá un atributo instancia de SubjectObserverMapping que almacenará los links entre sujetos y observadores y tendrá rutinas para adherir y remover dichos links y para notificar a los observadores de los cambios producidos en el sujeto (ver Figura 4.8.b).

**Figura 4.8.a.** Metamodelo ISM-Eiffel del patrón *Observer*

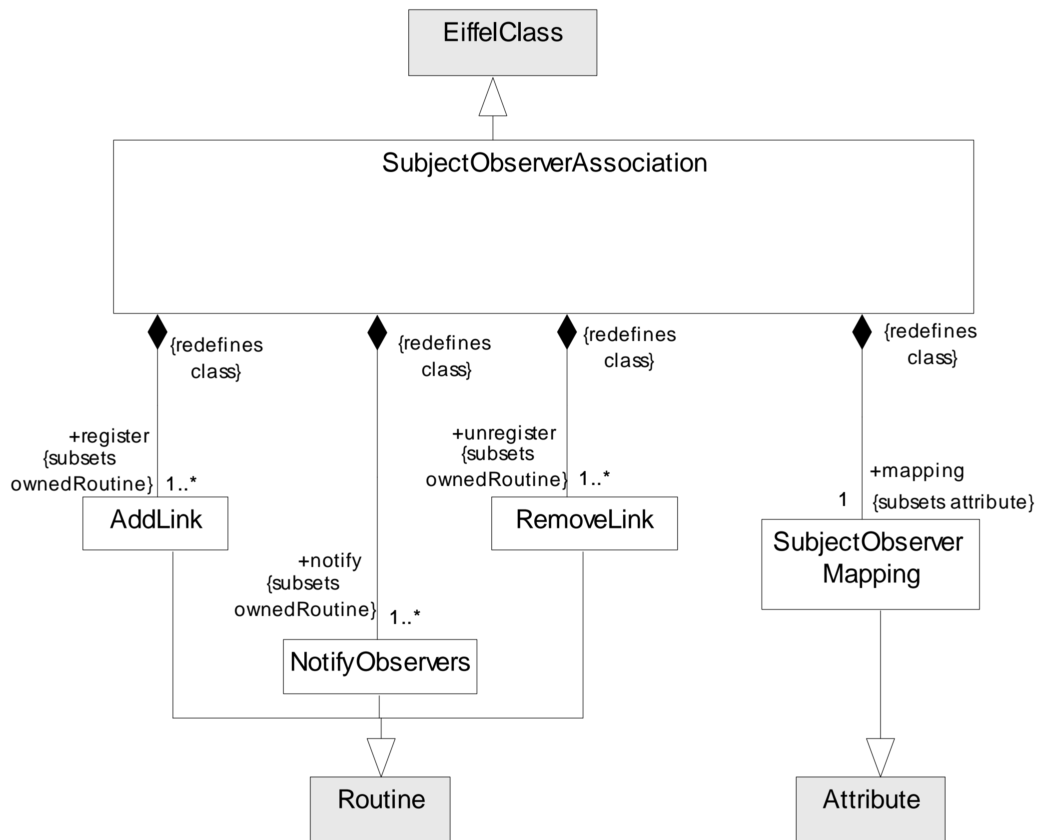



**Figura 4.8.b.** Metamodelo ISM-Eiffel: SubjectObserverAssociation

### Descripción de las metaclases

## AddLink

### Generalizaciones

- Routine (de ISM-Eiffel)

### Descripción

Define una rutina declarada por una instancia de la metaclase SubjectObserverAssociation.

### Asociaciones

- subjectObserverAssociation: SubjectObserverAssociation [1] — Designa la clase que declara esta rutina. Redefine Routine::class.

### Restricciones

[1] Esta rutina cambia el estado de la instancia que la define.

**not** self.isQuery

[2] Esta rutina tiene un conjunto no vacío de argumentos formales y uno de ellos debe ser de entrada (direction= #in) y del tipo Observer.

```
self.arguments->notEmpty( ) and
self.arguments ->select (arg | arg.direction= #in and arg.type= oclIsKindOf(Observer))-> size()= 1
```

[3] Su accesibilidad debe ser disponible o selectivamente disponible, en este último caso, una instancia de la metaclase Subject deberá existir entre sus clientes.

```
self.availability = #available or
    (self.availability = #selectively_available implies
            (self.clients->exists(c | c.oclIsKindOf (Subject)  )  ) )
```

## Attach

### Generalizaciones

- Routine (de ISM-Eiffel)

### Descripción

Define una rutina declarada por un sujeto.

### Asociaciones

- subject: Subject [1] Designa la clase que declara esta rutina. Redefine Routine::class.

### Restricciones

[1] Esta rutina cambia el estado del sujeto.

```
not self.isQuery
```

[2] Esta rutina tiene un conjunto no vacío de argumentos formales y uno de ellos debe ser de entrada (direction= #in) y del tipo Observer.

```
self.arguments->notEmpty( ) and
self.arguments ->select (arg | arg.direction= #in and arg.type= oclIsKindOf(Observer))-> size()= 1
```

[3] Su accesibilidad debe ser disponible o selectivamente disponible, en este último caso, una instancia de la metaclase Observer deberá existir entre sus clientes.

```
self.availability = #available or
    (self.availability = #selectively_available implies
            (self.clients->exists(c | c.oclIsKindOf (Observer)  )  ) )
```

[4] Si el sujeto que declara esta rutina tiene una referencia a una clase SubjectObserverAssociation, esta rutina delegará su tarea a esta clase, invocando a una rutina instancia de AddLink.

```
not self.subject.subjectObserverReference -> isEmpty implies
    self.invokedRoutine->exists (r | r.oclIsTypeOf(AddLink))
```

## Detach

### Generalizaciones

- Routine (de ISM-Eiffel)

### Descripción

Define una rutina declarada por un sujeto.

### Asociaciones

- subject: Subject [1] — Designa la clase que declara esta rutina. Redefine Routine::class.

### Restricciones

[1] Esta rutina cambia el estado del sujeto.

```
not self.isQuery
```

[2] Tiene un conjunto no vacío de argumentos y uno de ellos debe ser de entrada y del tipo Observer.

```
self.arguments →notEmpty( ) implies
self.arguments → select (arg | arg.direction= #in
        and arg.type = oclIsKindOf(Observer)) → size( )=1
```

[3] Su accesibilidad debe ser disponible o selectivamente disponible, en este último caso, una instancia de la metaclase Observer deberá existir entre sus clientes.

```
self.availability = #available or
   (self.availability = #selectively_available implies
        (self.clients->exists(c | c.oclIsKindOf (Observer) ) ) )
```

[4] Si el sujeto que declara esta rutina tiene una referencia a una clase SubjectObserverAssociation, esta rutina delegará su tarea a esta clase invocando a una rutina instancia de RemoveLink.

```
not self.subject.subjectObserverReference -> isEmpty implies
   self.invokedRoutine->exists (oclIsTypeOf(RemoveLink))
```

## EfectiveObserver

### Generalizaciones

- EiffelClass (de ISM-Eiffel)

### Descripción

Esta metaclase especifica las características que debe tener una clase con el comportamiento de un observador concreto en el código Eiffel de un patrón *Observer*.

### Asociaciones

- relationshipObserver: RelationshipObserver [1] — Designa una relación de Generalización donde EffectiveObserver cumple el rol de heredero. Subconjunto de EiffelClass::generalization.
- subjectReference: SubjectReference [1] — Denota un atributo propio, el cual es una referencia al sujeto o sujetos que el observador está observando. Subconjunto de EiffelClass::attribute.

### Restricciones

[1] Una instancia de un sujeto concreto no debe ser una clase diferida.

```
not self.isDeferred
```

## EffectiveSubject

### Generalizaciones

- EiffelClass (de ISM-Eiffel)

### Descripción

Esta clase especifica las características que debe tener una clase sujeto concreto en el código Eiffel de un patrón *Observer*.

### Asociaciones

- getState: GetState [1..*] — Toda instancia de ConcreteSubject debe tener una o más rutinas instancias de GetState. Puede ser propia o heredada. Subconjunto de NameSpace::member.
- relationshipSubject: RelationshipSubject [1] — Designa una relación de Generalización donde EffectiveSubject cumple el rol de heredero. Subconjunto de EiffelClass::generalization.
- setState: GetState [1..*] — Toda instancia de ConcreteSubject debe tener una o más rutinas instancias de SetState Puede ser propia o heredada. Subconjunto de NameSpace::member.
- state: Attribute [1..*] — Especifica el conjunto de todos los atributos miembros de ConcreteSubject. Pueden ser propios o heredados. Subconjunto de NameSpace::member.

### Restricciones

[1] Una instancia de un sujeto efectivo no debe ser una clase diferida.

```
not self.isDeferred
```

## GetState

### Generalizaciones

- Routine (de ISM-Eiffel)

### Descripción

Define una rutina miembro de EffectiveSubject. Especifica un servicio que puede ser requerido desde otro objeto.

### Asociaciones

No tiene asociaciones adicionales.

### Restricciones

[1] No es una rutina diferida y no cambia el estado del sujeto.

```
not self.isDeferred and self.isQuery
```

[2] Debe retornar el estado del sujeto, por lo tanto dentro del conjunto de argumentos debe haber uno de tipo out o return.

```
self.arguments -> notEmpty( ) and
   self.arguments ->select (arg | arg.direction = #return or arg.direction = #out)->size() >=1
```

[3] Su accesibilidad debe ser disponible o selectivamente disponible, en este último caso, una instancia de clase Observer deberá existir entre sus clientes.

```
self.availability = #available or
    (self.availability = #selectively_available implies
            (self.clients->exists(c | c.oclIsKindOf (Observer)  ) ) )
```

## Notify

### Generalizaciones

- Routine (de ISM-Eiffel)

### Descripción

Define una rutina declarada por un sujeto.

### Asociaciones

- subject: Subject[1] Designa la clase que declara esta rutina. Redefine Routine::class.

### Restricciones

[1] Esta rutina no cambia el estado del sujeto.

```
self.isQuery
```

[2] Su accesibilidad debe ser disponible o selectivamente disponible, en este último caso, una instancia de clase Observer deberá existir entre sus clientes.

```
self.availability = #available or  (self.availability = #selectively_available implies
            (self.clients->exists(c | c.oclIsKindOf (Observer) ) ) )
```

[3] Si el sujeto que declara esta rutina tiene una referencia a una clase SubjectObserverAssociation, esta rutina delegará su tarea a esta clase invocando a una rutina instancia de NotifyObserver.

```
not self.subject.subjectObserverReference -> isEmpty implies
    self.invokedRoutine->exists (oclIsTypeOf(NotifyObserver))
```

## NotifyObservers

### Generalizaciones

- Routine (de ISM-Eiffel)

### Descripción

Define una rutina declarada por una instancia de SubjectObserverAssociation.

### Asociaciones

- subjectObserverAssociation: SubjectObserverAssociation [1] Designa la clase que declara esta rutina. Redefine Routine::class.

### Restricciones

[1] Esta rutina no cambia el estado de la instancia que la define.

```
self.isQuery
```

[2] Su accesibilidad debe ser disponible o selectivamente disponible, en este último caso, una instancia de metaclase Subject deberá existir entre sus clientes.

```
self.availability = #available or (self.availability = #selectively_available implies
        (self.clients->exists(c | c.oclIsKindOf (Subject) ) ) )
```

## Observer

### Generalizaciones

- EiffelClass (de ISM-Eiffel)

### Descripción

Esta metaclase especifica las características que debe tener toda clase que cumpla el rol de observador en el modelo de un patrón *Observer* a nivel de código Eiffel.

### Asociaciones

- update: Update [1..*] — Toda instancia de Observer debe tener al menos una rutina, instancia de Update. Especifica una rutina propia. Subconjunto de EiffelClass::ownedRoutine.

### Restricciones

No tiene restricciones adicionales.

## ObserverReference

### Generalizaciones

- Attribute (de ISM-Eiffel)

### Descripción

Esa metaclase especifica un atributo, el cual es una referencia a los observadores de un sujeto dado.

### Asociaciones

- subject: Subject [1] — Designa la clase que declara este atributo. Redefine Attribute::class.

### Restricciones

[1] La accesibilidad de este atributo es privada.

```
self.availability = #secret
```

[2] El tipo de este atributo deberá corresponder a alguna de las colecciones de la librería Eiffel, y el tipo de elementos que almacene deberá ser del tipo Observer.

```
self.type.oclIsKindOf(EiffelCollection) and
    self.type.parameters->size() =1 and
    self.type.parameters.ownedParameteredElement.oclIsTypeOf(Observer)
```

## RelationshipObserver

### Generalizaciones

- Generalization (de Kernel)

### Descripción

Esta clase especifica la relación de herencia (Generalization) entre un observador (Observer) y un observador concreto (EffectiveObserver).

### Asociaciones

- heir: EffectiveObserver [1] — Designa el elemento que cumple el rol de heredero en la relación. Redefine Generalization::specific.
- parent: Observer [1] — Designa el elemento que cumple el rol de padre en la relación. Redefine Generalization::general.

### Restricciones

No hay restricciones adicionales

## RelationshipSubject

### Generalizaciones

- Generalization (de Kernel)

### Descripción

Esta clase especifica la relación de herencia (Generalization) entre un sujeto abstracto (Subject) y un sujeto concreto (EffectiveSubject) en el modelo de un patrón *Observer*.

### Asociaciones

- heir: EffectiveSubject [1] — Designa el elemento que cumple el rol de heredero en la relación. Redefine Generalization::specific.
- parent: Subject [1] — Designa el elemento que cumple el rol de padre en la relación. Redefine Generalization::general.

### Restricciones

No hay restricciones adicionales

## RemoveLink

### Generalizaciones

- Routine (de ISM-Eiffel)

### Descripción

Define una rutina declarada por una instancia de SubjectObserverAssociation.

### Asociaciones

- subjectObserverAssociation: SubjectObserverAssociation [1] — Designa la clase que declara esta rutina. Redefine Routine::class.

### Restricciones

[1] Esta rutina cambia el estado de la instancia que la define.

```
not self.isQuery
```

[2] Tiene un conjunto no vacío de argumentos y uno de ellos debe ser de entrada y del tipo Observer.

```
self.arguments →notEmpty( )  implies
self.arguments → select (arg | arg.direction= #in
            and arg.type = oclIsKindOf(Observer)) → size( )=1
```

[3] Su accesibilidad debe ser disponible o selectivamente disponible, en este último caso, una instancia de la metaclase Subject deberá existir entre sus clientes.

```
self.availability = #available or
    (self.availability = #selectively_available implies
            (self.clients->exists(c | c.oclIsKindOf (Subject) )  )  )
```

## SetState

### Generalizaciones

- Routine (de ISM-Eiffel)

### Descripción

Define una rutina miembro de EffectiveSubject. Especifica un servicio que puede ser requerido desde otro objeto.

### Asociaciones

No tiene asociaciones adicionales.

### Restricciones

[1] No es una rutina diferida y modifica el estado del sujeto.

```
not self.isDeferred and not self.isQuery
```

[2] El conjunto de argumentos es no vacío y por lo menos debe haber uno de entrada.

```
self.arguments →notEmpty( ) and
    self.arguments ->select (arg | arg.direction= #in) → size( ) >=1
```

[3] Su accesibilidad debe ser disponible o selectivamente disponible, en este último caso, una instancia de clase Observer deberá existir entre sus clientes.

```
self.availability = #available or
    (self.availability = #selectively_available implies
            (self.clients->exists(c | c.oclIsKindOf (Observer)  ) ) )
```

## Subject

### Generalizaciones

- EiffelClass (de ISM-Eiffel)

### Descripción

Esta metaclase especifica las características que debe tener una clase sujeto en el modelo de un patrón *Observer* a nivel de código Eiffel.

### Asociaciones

| | |
|---|---|
| • attach: Attach [1..*] | Toda instancia de Subject debe tener por lo menos una rutina instancia de Attach. Especifica una rutina propia. Subconjunto de EiffelClass::ownedRoutine. |
| • detach: Detach [1..*] | Toda instancia de Subject debe tener por lo menos una rutina instancia de Detach. Especifica una rutina propia Subconjunto de EiffelClass::ownedRoutine. |
| • notify: Notify[1..*] | Toda instancia de Subject debe tener por lo menos una rutina instancia de Notify. Especifica una rutina propia Subconjunto de EiffelClass::ownedRoutine. |
| • observerReference: ObserverReference [0..1] | Denota el atributo que permite al sujeto mantener una referencia a sus observadores. Subconjunto de EiffelClass::attribute. |
| • subjectObserverReference: ObserverReference [0..1] | Denota el atributo, el cual es una referencia a una clase que maneja la relación sujeto-observadores. Subconjunto de EiffelClass::attribute. |

### Restricciones

No hay restricciones adicionales.

## SubjectObserverAssociation

- EiffelClass (de ISM-Eiffel)

### Descripción

Esta metaclase especifica las características de la clase encargada de administrar la relación entre un sujeto y sus observadores.

### Asociaciones

| | |
|---|---|
| • mapping:SubjectObserverMapping [1] | Especifica un atributo propio. Subconjunto de EiffelClass::attribute. |
| • notify: NotifyObservers [1..*] | Especifica una rutina propia. Subconjunto de EiffelClass::ownedRoutine. |
| • register: AddLink [1..*] | Especifica una rutina propia. Subconjunto de EiffelClass::ownedRoutine. |
| • unregister: RemoveLink [1..*] | Especifica una rutina propia. Subconjunto de EiffelClass::ownedRoutine. |

### Restricciones

No hay restricciones adicionales.

## SubjectObserverMapping

- Attribute (de ISM-Eiffel)

### Descripción

Esta metaclase especifica el atributo de la clase SubjectObserverAssociation, encargado de mantener el mapping entre un sujeto y sus observadores.

### Asociaciones

- subjectObserverAssociation: SubjectObserverAssociation [1] — Designa la clase que declara este atributo. Redefine Attribute::class.

### Restricciones

[1] Su accesibilidad es privada.

```
self.availability = #secret
```

## SubjectObserverReference

- Attribute (de ISM-Eiffel)

### Descripción

Este atributo es una referencia a una clase SubjectObserverAssociation, la cual es la encargada de mantener la relación entre un sujeto y sus observadores.

### Asociaciones

- subject: Subject [1] — Designa la clase que declara este atributo. Redefine Attribute:: class.
- type: SubjectObserverAssociation [1] — Referencia al tipo de este atributo. Redefine Attribute::type.

### Restricciones

[1] La accesibilidad de este atributo es privada.

```
self.availability = #secret
```

## SubjectReference

### Generalizaciones

- Attribute (de ISM-Eiffel)

### Descripción

Este atributo representa una referencia al sujeto o a los sujetos de un observador dado.

### Asociaciones

- effectiveObserver: EffectiveObserver [1] Designa la clase que declara este atributo. Redefine Attribute:: class.

### Restricciones

[1] La accesibilidad de este atributo es secreta.

```
self.availability = #secret
```

[2] El tipo de este atributo podrá corresponder a un atributo de tipo Subject o a alguna de las colecciones de la librería Eiffel, y el tipo de elementos que almacene deberá ser del tipo Subject (según el observador observe un sujeto o varios sujetos respectivamente).

```
self.type.oclIsKindOf(Subject) or (
    self.type.oclIsKindOf(EiffelCollection) and self.type.parameter->size() =1 and
        self.type.parameters.ownedParameteredElement.oclIsTypeOf(Subject) )
```

## Update

### Generalizaciones

- Routine (de ISM-Eiffel)

### Descripción

Define la rutina declarada por Observer que especifica un servicio que puede ser requerido por otro objeto.

### Asociaciones

- Observer: Observer [1] Designa la clase que declara esta rutina. Redefine Routine::class.

### Restricciones

[1] Es una rutina que no cambia el estado del observador.

```
self.isQuery
```

[2] Su accesibilidad debe ser disponible o selectivamente disponible, en este último caso, una instancia de clase Subject o SubjectObserverAssociation deberá existir entre sus clientes.

```
self.availability = #available or
    (self.availability = #selectively_available implies
        (self.clients->exists(c | c.oclIsKindOf (Subject) or
                                  c.oclIsKindOf (SubjectObserverAssociation) ) ) )
```

#### 4.2.2.4.2. Metamodelo del patrón *Observer* a nivel ISM Java

Las Figuras desde 4.9.a hasta 4.9.e muestran el metamodelo del patrón *Observer* a nivel ISM para Java. Fue construido en base al metamodelo ISM-Java (Anexo B). Las metaclases en color gris claro corresponden a las metaclases de este último mientras que las de color gris oscuro corresponden a las metaclases del metamodelo UML.

La principal diferencia con el metamodelo del patrón *Observer* a nivel PSM específico a la plataforma Java es que a nivel de código no existen los extremos de asociación y aparecen detalles de implementación.

El metamodelo establece que un sujeto puede ser una clase o una interfaz Java. Si es una clase, además de tener al menos tres métodos instancias de Attach, Detach y Notify, contendrá una referencia a sus observadores a través de un atributo, el cual puede ser una instancia de ObserverReference o una instancia de SubjectObserverReference. En el primer caso, el atributo referencia a la colección de observadores, en el segundo caso referencia a la clase intermedia encargada de mantener la relación entre sujetos y observadores en el caso que dicha relación sea compleja.

Una clase que cumpla el rol de observador podrá ser una clase o una interfaz Java y tendrá al menos un método instancia de Update.

La metaclase ConcreteSubject especifica una clase Java que tiene un estado conformado por un “field” o un conjunto de “fields”, los cuales serán objeto de observación, y como mínimo métodos que permitan obtener y modificar sus valores. Tanto los “fields” como los métodos pueden ser propios o heredados.

Si una instancia de ConcreteSubject hereda de una instancia de ClassSubject, heredará un campo instancia de ObserverReference o instancia de SubjectObserverReference, por lo tanto no necesita declarar ninguna referencia a sus observadores. Por el contrario si implementa una interfaz instancia de InterfaceSubject, deberá declarar un campo instancia de ObserverReference o SubjectObserverReference para mantener información sobre sus observadores.

ConcreteObserver especifica una clase Java, que puede heredar de una clase instancia de ClassObserver o implementar una interfaz instancia de InterfaceObserver y mantendrá una referencia al sujeto o sujetos que observa a través de un atributo instancia de SubjectReference.

SubjectObserverAssociation especifica la clase Java encargada de mantener la relación entre un sujeto y sus observadores. Esta clase tendrá un atributo instancia de SubjectObserverMapping que almacenará los links entre el sujeto y sus observadores y tendrá métodos para adherir y remover dichos links y notificar a los observadores de los cambios producidos en el sujeto.

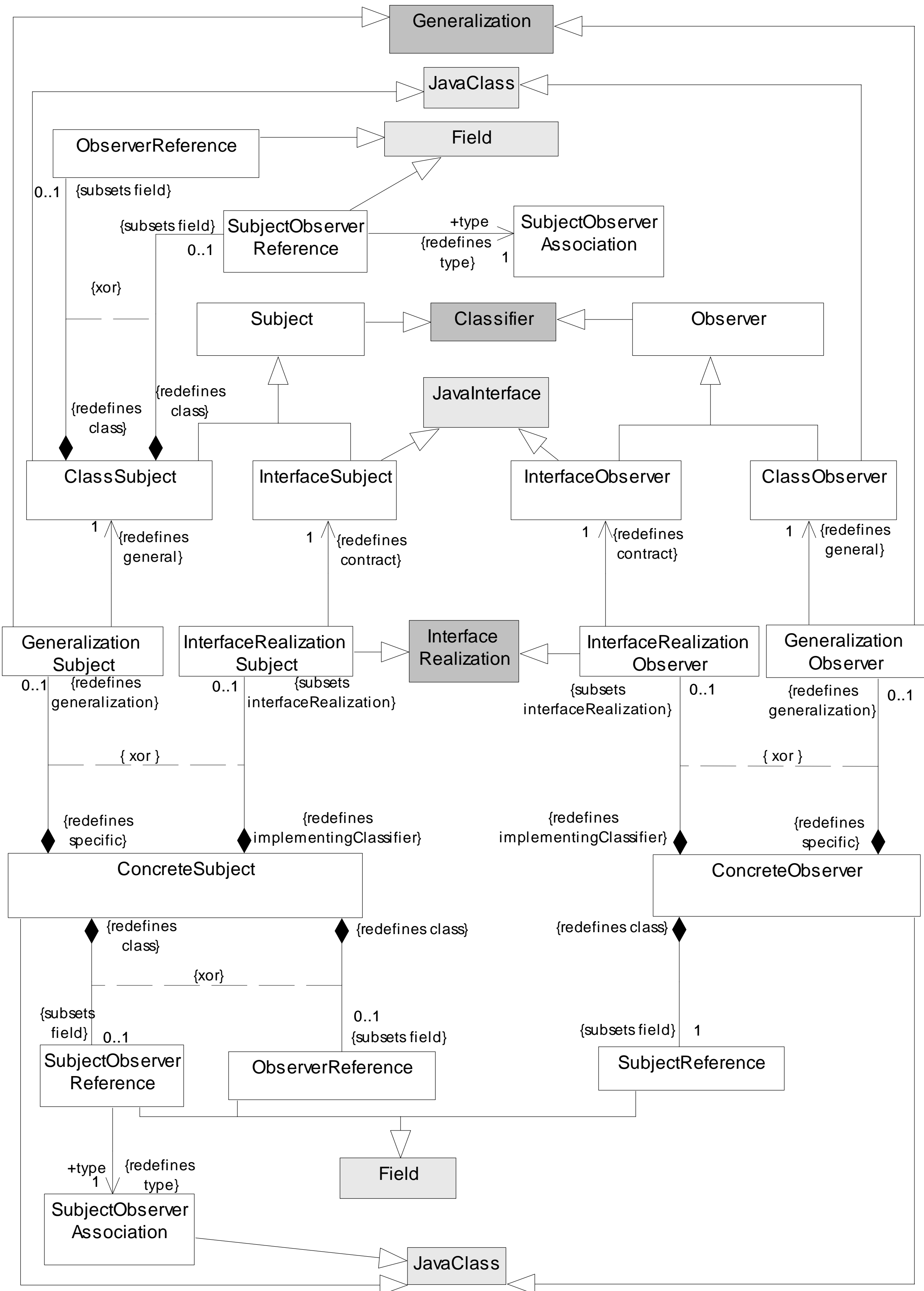
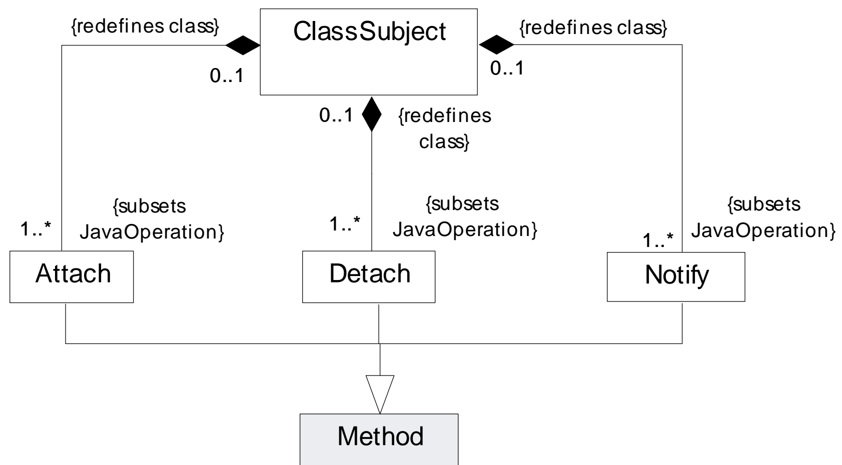



**Figura 4.9.a.** Metamodelo ISM-Java del patrón *Observer*- Diagrama de Clases



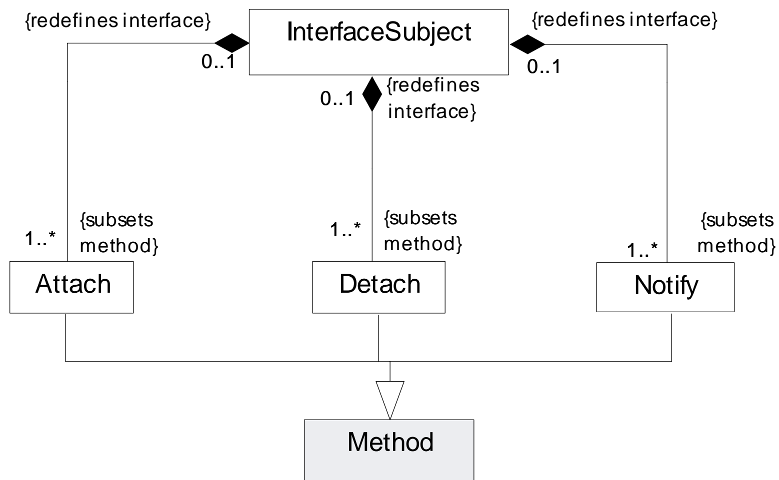




**Figura 4.9.b.** Sujeto abstracto: Operaciones

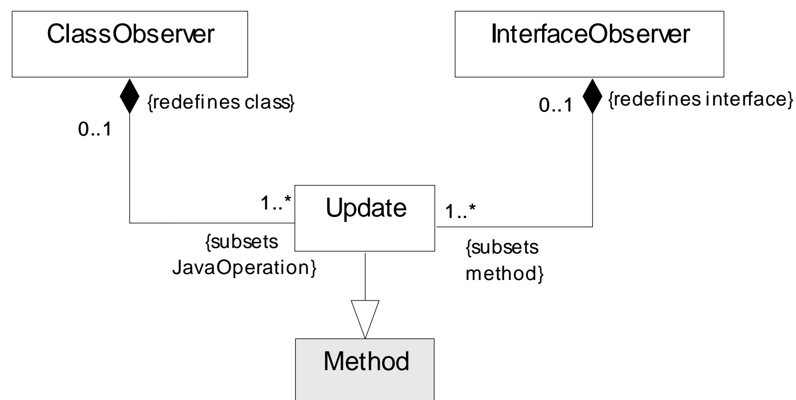



**Figura 4.9.c.** Observador abstracto: Operaciones

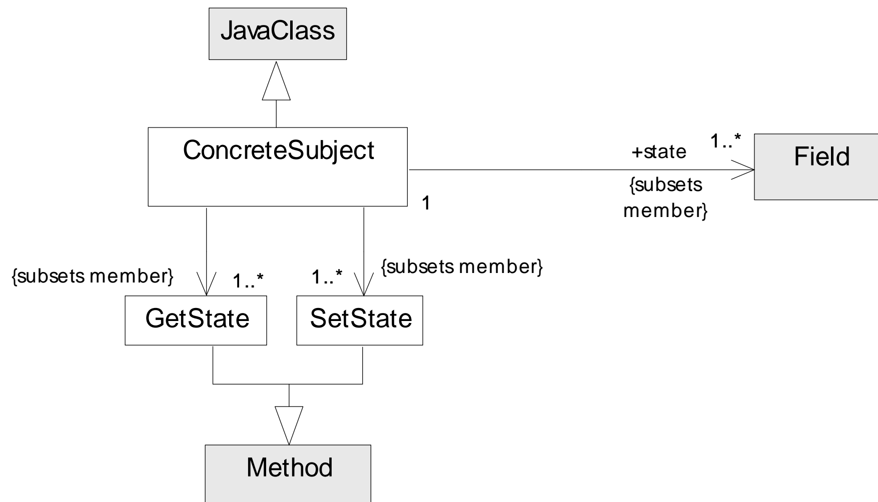



**Figura 4.9.d.** Sujeto concreto: Operaciones y atributos

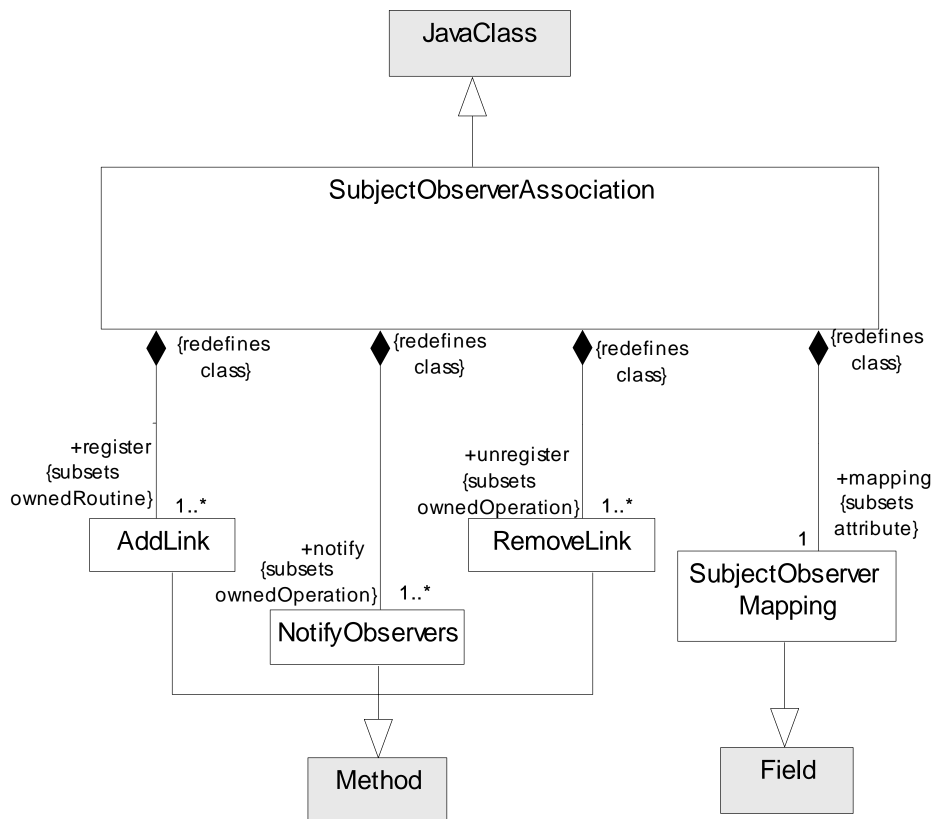



**Figura 4.9.e.** SubjectObserverAssociation: Operaciones

**Descripción de las metaclases**

## AddLink

### Generalizaciones

- Method (de ISM-Java)

### Descripción

Define un método declarado por una instancia de la clase SubjectObserverAssociation.

### Asociaciones

- subjectObserverAssociation: SubjectObserverAssociation [1] — Designa la clase que declara este método. Redefine JavaOperation::class.

### Restricciones

[1] Este método cambia el estado de la instancia que lo define.

```
not self.isQuery
```

[2] Tiene un conjunto no vacío de parámetros y uno de ellos debe ser de entrada y del tipo Observer.

```
self.parameter->notEmpty( ) and
self. parameter ->select ( param |    param.direction= #in and
      param.type= oclIsKindOf(Observer)) -> size( ) = 1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## Attach

### Generalizaciones

- Method (de ISM-Java)

### Descripción

Define un método declarado por un Sujeto.

### Asociaciones

- classSubject: ClassSubject [0..1] — Designa la clase que declara este método. Redefine JavaOperation::class.
- interfaceSubject: InterfaceSubject [0..1] — Designa la interfaz que declara este método. Redefine Method::interface.

### Restricciones

[1] Este método cambia el estado del sujeto.

```
not self.isQuery
```

[2] Tiene un conjunto no vacío de parámetros y uno de ellos debe ser de entrada y del tipo Observer.

```
self.parameter->notEmpty( ) and
self. parameter ->select ( param |    param.direction= #in and
      param.type= oclIsKindOf(Observer)) -> size( ) = 1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

[4] Si el sujeto que declara esta rutina tiene una referencia a una clase SubjectObserverAssociation, esta rutina delegará su tarea a esta clase, invocando a una rutina instancia de AddLink.

```
not self.subject.subjectObserverReference -> isEmpty implies
    self.invokedRoutine->exists (r | r.oclIsTypeOf(AddLink))
```

## ClassObserver

### Generalizaciones

- Observer, JavaClass (de ISM-Java)

### Descripción

Una metaclase ClassObserver especifica las características que debe tener una clase Java que cumpla el rol de observador en el modelo de un patrón *Observer*.

### Asociaciones

- update: Update [1..*] — Toda instancia de ClassObserver debe tener al menos un método instancia de Update. Subconjunto de JavaClass::javaOperation.

### Restricciones

No hay restricciones adicionales.

## ClassSubject

### Generalizaciones

- Subject, JavaClass (de ISM-Java)

### Descripción

Esta metaclase especifica las características que debe tener una clase Java que cumpla el rol de sujeto en el modelo de un patrón *Observer*.

### Asociaciones

- attach: Attach [1..*] — Toda instancia de ClassSubject tiene por lo menos un método instancia de Attach. Subconjunto de JavaClass:: javaOperation.
- detach: Detach [1..*] — Toda instancia de ClassSubject tiene por lo menos un método instancia de Detach. Subconjunto de JavaClass:: javaOperation.

- notify: Notify[1..*] — Toda instancia de ClassSubject tiene por lo menos un método instancia de Notify. Subconjunto de JavaClass:: javaOperation.
- observerReference: ObserverReference[0..1] — Denota el atributo que permite al sujeto mantener una referencia a sus observadores. Subconjunto de JavaClass::field.
- subjectObserverReference: SubjectObserverReference [0..1] — Denota el atributo, el cual es una referencia a una clase que mantiene la relación sujeto-observadores. Subconjunto de JavaClass::field.

### Restricciones

No hay restricciones adicionales.

## ConcreteObserver

### Generalizaciones

- JavaClass (de ISM-Java)

### Descripción

Esta metaclase especifica las características que debe tener una clase Java con el comportamiento de un observador concreto en el modelo de un patrón *Observer*.

### Asociaciones

- generalizationObserver: GeneralizationObserver [0..1] — Designa una relación de generalización donde ConcreteObserver cumple el rol de hijo (specific). Redefine Classifier::generalization.
- interfaceRealizationObserver: InterfaceRealizationObserver [0..1] — Designa una relación de realización de interfaz donde ConcreteObserver cumple cumple el rol del clasificador que implementa el contrato (implementingClassifier). Subconjunto de BehavioredClassifier::interfaceRealization.
- subjectReference: SubjectReference [0..1] — Denota una referencia al sujeto o sujetos que el observador está observando. Subconjunto de JavaClass::field.

### Restricciones

[1] Una instancia de un observador concreto no puede ser una clase abstracta.

```
not self.isAbstract
```

[2] Si una instancia de un observador concreto participa en una realización de interfaz, entonces debe ser un BehavioredClassifier.

```
self.interfaceRealizationObserver -> notEmpty ()
    implies self.superClass -> exists (c | c.oclIsTypeOf (BehavioredClassifier) )
```

## ConcreteSubject

### Generalizaciones

- JavaClass (de ISM-Java)

### Descripción

Esta metaclase especifica las características que debe tener una clase que cumpla el rol de sujeto concreto en el modelo de un patrón *Observer*.

### Asociaciones

| | |
|---|---|
| • generalizationSubject: GeneralizationSubject [0..1] | Designa una relación de generalización donde ConcreteSubject cumple el rol de hijo (specific). Redefine Classifier::generalization. |
| • getState: GetState [1..*] | Toda instancia de ConcreteSubject debe tener uno o más métodos instancias de GetState. Pueden ser propios o heredados. Subconjunto de NameSpace::member. |
| • interfaceRealizationSubject: InterfaceRealization [0..1] | Designa una relación de realización de interfaz donde ConcreteSubject cumple el rol del clasificador que implementa el contrato (implementingClassifier). Subconjunto de BehavioredClassifier::interfaceRealization. |
| • observerReference: ObserverReference[0..1] | Denota el atributo que permite al sujeto mantener una referencia a sus observadores. Subconjunto de JavaClass::field. |
| • setState: GetState [1..*] | Toda instancia de ConcreteSubject debe tener uno o más métodos instancias de SetState. Pueden ser propios o heredados. Subconjunto de NameSpace::member. |
| • state: Property [1..*] | Especifica un conjunto no vacío de todos los atributos de ConcreteSubject. Pueden ser propios o heredados. Subconjunto de NameSpace::member. |
| • subjectObserverReference: SubjectObserverReference [0..1] | Denota el atributo, el cual es una referencia a una clase que mantiene la relación sujeto-observadores. Subconjunto de JavaClass::field. |

### Restricciones

[1] Una instancia de un sujeto concreto no puede ser una clase abstracta.

```
not self.isAbstract
```

[2] Si una instancia de un sujeto concreto participa en una realización de interfaz, entonces debe ser un BehavioredClassifier.

```
self.interfaceRealizationSubject -> notEmpty ()
    implies self.superClass -> exists (c | c.oclIsTypeOf (BehavioredClassifier) )
```

[3] Si una instancia de un sujeto concreto hereda de ClassSubject, heredará el campo ObserverReference, o el campo subjectObserverReference, por lo tanto no necesita declarar ninguna referencia a sus observadores. Por el contrario si implementa la interfaz InterfaceSubject, deberá declarar un campo del tipo ObserverReference o SubjectObserverReference para mantener información sobre sus observadores.

```
not self.generalizationSubject -> isEmpty () implies
    self.observerReference -> isEmpty() and self.subjectObserverReference-> isEmpty()
and
    not self.interfaceRealizationSubject-> isEmpty() implies
       not self.observerReference->isEmpty() xor not self.observerReference-> isEmpty()
```

## Detach

### Generalizaciones

- Method (de ISM-Java)

### Descripción

Define un método declarado por un sujeto.

### Asociaciones

- classSubject: ClassSubject [0..1] — Designa la clase que declara este método. Redefine JavaOperation::class.
- interfaceSubject: InterfaceSubject [0..1] — Designa la interfaz que declara este método. Redefine Method::interface.

### Restricciones

[1] Este método cambia el estado del sujeto.

```
not self.isQuery
```

[2] Tiene un conjunto no vacío de parámetros y uno de ellos debe ser de entrada y del tipo Observer.

```
self.parameter ->notEmpty( )  and
self.parameter ->select ( param |    param.direction= #in and
                                param.type= oclIsKindOf(Observer)) -> size( )  = 1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

[4] Si el sujeto que declara esta rutina tiene una referencia a una clase SubjectObserverAssociation, esta rutina delegará su tarea a esta clase, invocando a una rutina instancia de RemoveLink.

```
not self.subject.subjectObserverReference -> isEmpty implies
    self.invokedRoutine->exists (r | r.oclIsTypeOf(RemoveLink))
```

## GeneralizationObserver

### Generalizaciones

- Generalization (de Kernel)

### Descripción

Esta metaclase especifica una relación de generalización entre un observador abstracto (ClassObserver) y un observador concreto (ConcreteObserver) en el modelo de un patrón *Observer*.

**Asociaciones**

- classObserver: ClassObserver [1] — Designa el elemento general de esta relación. Redefine Generalization::general.
- concreteObserver: ConcreteObserver [1] — Designa el elemento específico de esta relación. Redefine Generalization::specific.

**Restricciones**

No hay restricciones adicionales.

## GeneralizationSubject

**Generalizaciones**

- Generalization (de Kernel)

**Descripción**

Esta metaclase especifica una relación de generalización entre un sujeto abstracto (ClassSubject) y un sujeto concreto (ConcreteSubject) en el modelo de un patrón *Observer*.

**Asociaciones**

- classSubject: ClassSubject [1] — Designa el elemento general de esta relación. Redefine Generalization::general.
- concreteSubject: ConcreteSubject [1] — Designa el elemento específico de esta relación. Redefine Generalization::specific.

**Restricciones**

No hay restricciones adicionales.

## GetState

**Generalizaciones**

- Method (de ISM-Java)

**Descripción**

Define un método miembro de ConcretetSubject. Especifica un servicio que puede ser requerido desde otro objeto.

**Asociaciones**

No tiene asociaciones adicionales.

**Restricciones**

[1] Es un método observador y no abstracto.

```
self.isQuery and not self.isAbstract
```

[2] Como debe retornar el estado del sujeto, el conjunto de parámetros no debe ser vacío, debe haber por lo menos uno cuya dirección sea *out* o *return*.

```
self.parameter -> notEmpty( ) and
   self.parameter->select (par | par.direction = #return or par.direction = #out)->size () >=1
```

[3] Su visibilidad debe ser pública.

self.visibility = #public

## InterfaceObserver

### Generalizaciones

- Observer, JavaInterface (de ISM-Java)

### Descripción

Un InterfaceObserver especifica las características que debe tener una interfaz Java que cumpla el rol de observador abstracto en el modelo de un patrón *Observer*.

### Asociaciones

- update: Update [1..*] Toda instancia de InterfaceObserver tiene al menos una operación instancia de Update. Subconjunto de JavaInterface::method.

### Restricciones

No hay restricciones adicionales.

## InterfaceSubject

### Generalizaciones

- Subject, JavaInterface (de ISM-Java)

### Descripción

Esta metaclase especifica las características que debe tener una interfaz Java que cumpla el rol de sujeto abstracto en el modelo de un patrón *Observer*.

### Asociaciones

- attach: Attach [1..*] Toda instancia de InterfaceSubject tiene por lo menos un método instancia de Attach. Subconjunto de JavaInterface::method.
- detach: Detach [1..*] Toda instancia de InterfaceSubject tiene por lo menos un método instancia de Detach. Subconjunto de JavaInterface::method.
- notify: Notify[1..*] Toda instancia de InterfaceSubject tiene por lo menos un método instancia de Notify. Subconjunto de JavaInterface::method.

### Restricciones

No hay restricciones adicionales.

## InterfaceRealizationObserver

### Generalizaciones

- InterfaceRealization (de Kernel)

### Descripción

Esta metaclase especifica una relación de realización de interfaz entre un observador abstracto (InterfaceObserver) y un observador concreto (ConcreteObserver) en el modelo de un patrón *Observer*.

### Asociaciones

- concreteObserver: ConcreteObserver [1] — Designa el elemento que implementa el contrato en esta relación. Redefine InterfaceRealization::implementingClassifier.
- interfaceObserver: InterfaceObserver [1] — Designa el elemento que define el contrato en esta relación. Redefine InterfaceRealization::contract.

### Restricciones

No hay restricciones adicionales.

## InterfaceRealizationSubject

### Generalizaciones

- InterfaceRealization (de Kernel)

### Descripción

Esta metaclase especifica una relación de realización de interfaz entre un sujeto abstracto (InterfaceSubject) y un sujeto concreto (ConcreteSubject) en el modelo de un patrón *Observer*.

### Asociaciones

- concreteSubject: ConcreteSubject [1] — Designa el elemento que implementa el contrato en esta relación. Redefine InterfaceRealization::implementingClassifier.
- interfaceSubject: InterfaceSubject [1] — Designa el elemento que define el contrato en esta relación. Redefine InterfaceRealization::contract.

### Restricciones

No hay restricciones adicionales.

## Notify

### Generalizaciones

- Method (de ISM-Java)

#### Descripción

Define un método declarado por un sujeto.

#### Asociaciones

- classSubject: ClassSubject [0..1] — Designa la clase que declara este método. Redefine JavaOperation::class.
- interfaceSubject: InterfaceSubject [0..1] — Designa la interfaz que declara este método. Redefine Method::interface.

#### Restricciones

[1] Es un método que no cambia el estado del sujeto.

```
self.isQuery
```

[2] Su visibilidad debe ser pública.

```
self.visibility = #public
```

[3] Si el sujeto que declara esta rutina tiene una referencia a una clase SubjectObserverAssociation, esta rutina delegará su tarea a esta clase, invocando a una rutina instancia de NotifyObserver.

```
not self.subject.subjectObserverReference -> isEmpty implies
    self.invokedRoutine->exists (r | r.oclIsTypeOf(NotifyObserver))
```

### NotifyObservers

#### Generalizaciones

- Method (de ISM-Java)

#### Descripción

Define un método declarado por una instancia de SubjectObserverAssociation.

#### Asociaciones

- subjectObserverAssociation: SubjectObserverAssociation [1] — Designa la clase que declara este método. Redefine JavaOperation::class.

#### Restricciones

[1] Es un método que no cambia el estado de la instancia que lo define.

```
self.isQuery
```

[2] Su visibilidad debe ser pública.

```
self.visibility = #public
```

### Observer

#### Generalizaciones

- Classifier (de Kernel)

### Descripción

Un Observer es un clasificador especializado que especifica al clasificador que cumple el rol de observador en el modelo de un patrón *Observer*. Es una metaclase abstracta.

### Asociaciones

No hay asociaciones adicionales.

### Restricciones

No hay restricciones adicionales

## ObserverReference

### Generalizaciones

- Field (de ISM-Java)

### Descripción

Este campo representa una referencia a los observadores de un sujeto dado.

### Asociaciones

- sujeto: Subject [1]   Designa la clase que declara este campo. Redefine Field::class.

### Restricciones

[1] El tipo de este campo deberá corresponder a alguna de las colecciones de la librería Java y el tipo de elementos que almacene deberá ser del tipo Observer.

```
self.type.oclIsKindOf(JavaCollection) and
    self.type.parameter->size() =1 and
    self.type.parameter.ownedParameteredElement.oclIsTypeOf(Observer)
```

[2] Su visibilidad debe ser privada o protegida.

```
self.visibility = #private or  self.visibility = #protected
```

## RemoveLink

### Generalizaciones

- Method  (de ISM-Java)

### Descripción

Define un método declarado por una instancia de SubjectObserverAssociation.

### Asociaciones

- subjectObserverAssociation: SubjectObserverAssociation [1]   Designa la clase que declara este método. Redefine JavaOperation::class.

### Restricciones

[1] Este método cambia el estado de la instancia que lo define.

```
not self.isQuery
```

[2] Tiene un conjunto no vacío de parámetros y uno de ellos debe ser de entrada y del tipo Observer.

```
self.parameter ->notEmpty( )  and
self.parameter ->select ( param |    param.direction= #in and
                                     param.type= oclIsKindOf(Observer)) -> size( )  = 1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## SetState

### Generalizaciones

- Method (de ISM-Java)

### Descripción

Define una operación miembro de ConcreteSubject. Especifica un servicio que puede ser requerido desde otro objeto.

### Asociaciones

No tiene asociaciones adicionales.

### Restricciones

[1] Es un método que no es abstracto y que modifica el estado del sujeto.

```
not self.isAbstract and not self.isQuery
```

[2] El conjunto de parámetros es no vacío y por lo menos debe haber por lo menos uno de entrada.

```
self.parameter->notEmpty( ) and
    self.parameter ->select (param | param.direction= #in) ->size( ) >=1
```

[3] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## Subject

### Generalizaciones

- Classifier (de Kernel)

### Descripción

Esta metaclase especifica un clasificador especializado que cumple el rol de sujeto en el modelo de un patrón *Observer*. Es una metaclase abstracta.

### Asociaciones

No hay asociaciones adicionales.

### Restricciones

No hay restricciones adicionales.

## SubjectObserverAssociation

### Generalizaciones

- JavaClass (de ISM-Java)

### Descripción

Esta metaclase especifica las características de la clase encargada de mantener la relación entre un sujeto y sus observadores.

### Asociaciones

| | |
|---|---|
| • mapping: SubjectObserverMapping [1] | Especifica un atributo propio. Subconjunto de JavaClass::field. |
| • notify: NotifyObservers [1..*] | Especifica una operación propia. Subconjunto de JavaClass::method. |
| • register: AddLink [1..*] | Especifica una operación propia. Subconjunto de JavaClass::method. |
| • unregister: RemoveLink [1..*] | Especifica una operación propia. Subconjunto de JavaClass::method. |

### Restricciones

No hay restricciones adicionales.

## SubjectObserverMapping

- Field (de ISM-Java)

### Descripción

Esta metaclase especifica el atributo de la clase SubjectObserverAssociation, encargado de mantener el mapping entre un sujeto y sus observadores.

### Asociaciones

| | |
|---|---|
| • subjectObserverAssociation: SubjectObserverAssociation [1] | Designa la clase que declara este atributo. Redefine Field::Class. |

### Restricciones

[1] Su visibilidad debe ser privada o protegida.

```
self.visibility = #private or self.visibility = #protected
```

## SubjectObserverReference

- Field (de ISM-Java)

### Descripción

Este atributo es una referencia a una clase SubjectObserverAssociation, la cual es la encargada de mantener la relación entre un sujeto y sus observadores.

### Asociaciones

- subject: Subject [1] — Designa la clase que declara este atributo. Redefine Attribute:: class.
- type: SubjectObserverAssociation [1] — Referencia al tipo de este atributo. Redefine Attribute::type.

### Restricciones

No hay restricciones adicionales.

## Update

### Generalizaciones

- Method (de ISM-Java)

### Descripción

Define un método declarado por un Observer que especifica un servicio que puede ser requerido por otro objeto.

### Asociaciones

- classObserver: ClassObserver [0..1] — Designa la clase que declara esta operación. Redefine JavaOperation::class.
- interfaceObserver: InterfaceObserver [0..1] — Designa la interfaz que declara esta operación. Redefine Method::interface.

### Restricciones

[1] Es un método que no cambia el estado del observador.

```
self.isQuery
```

[2] Su visibilidad debe ser pública.

```
self.visibility = #public
```

## 4.3. Especificando transformaciones de modelos basadas en metamodelos

Una definición de una transformación de modelos es una especificación de un mecanismo para convertir los elementos de un modelo, que es una instancia de un metamodelo particular, en elementos de otro modelo, que es instancia de otro metamodelo o posiblemente el mismo. Representan familias de transformaciones de modelos.

En esta tesis las transformaciones de modelos basadas en metamodelos son especificadas como contratos OCL. Éstos son descriptos por medio del nombre de la transformación, parámetros (modelo origen y modelo destino), precondiciones, postcondiciones y operaciones locales. La precondición es una invariante que establece las condiciones que deben ser cumplidas por el modelo origen para que la regla de transformación pueda ser aplicada. La postcondición es una invariante que el modelo destino cumple cuando la transformación es aplicada.

La definición de una transformación es declarativa y puede ser usada en las etapas de especificación del desarrollo basado en MDA para chequear la validez de una transformación.

La decisión de especificar transformaciones como contratos se basó en los beneficios conocidos del "diseño por contrato" (Meyer, 1997) tales como la construcción de un software de calidad, facilitando el reuso, la legibilidad, la documentación, el testeo y la depuración.

Se seleccionó OCL para la especificación de dichos contratos por ser un lenguaje estándar de OMG concebido para la especificación de restricciones sobre modelos y metamodelos en el contexto de MDA.

Si bien las transformaciones se definieron como contratos OCL en una notación muy simple, existe tan sólo una brecha de estructuración sintáctica con QVT. Podría decirse que la notación propuesta está alineada con QVT dado que este estándar está basado en el paquete CORE que depende de EMOF (Essential MOF) (MOF, 2006) y EssentialOCL (OCL, 2006).

### 4.3.1. Sintaxis para la definición de transformaciones

Cada definición de transformación comienza con la palabra clave **Transformation** seguida por el nombre de la regla y su cuerpo delimitado por llaves. Éste consta de parámetros, precondiciones, postcondiciones y operaciones locales.

Los parámetros se denotan mediante la palabra clave **parameters** seguida por una lista de parámetros. Esta lista está formada por dos elementos: el modelo origen y el modelo destino.

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

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

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

```
Transformation <nombre_transformación> {
    parameters <lista_parámetros>
    preconditions <lista_expresiones_OCL>
    postconditions <lista_expresiones_OCL>
    local operations <lista_expresiones_OCL>
    }
```

Pueden intercalarse líneas de comentarios precedidas por --.

A continuación se ejemplifican definiciones de transformaciones desde un PIM a un PSM-Eiffel y de un PSM-Eiffel a ISM-Eiffel.

### 4.3.2 Transformación PIM-UML a PSM-EIFFEL

La definición de transformación de modelos integra los distintos niveles de abstracción estableciendo la relación que existe entre los elementos de cada modelo.

La definición de transformación de PIM a PSM que se muestra parcialmente a continuación permite transformar un modelo del patrón de diseño *Observer* a nivel PIM, en un modelo del patrón O*bserver* en la plataforma Eiffel. La definición usa dichos modelos como parámetros origen y destino respectivamente.

Las postcondiciones establecen correspondencias entre clases e interfaces del modelo origen con las clases en el modelo destino garantizando, por ejemplo, que para cada clase del modelo origen, existirá una clase en el modelo destino, tal que:

- ambas clases tienen el mismo nombre, la misma visibilidad, el mismo grado de abstracción, las mismas restricciones y los mismos parámetros (si los tuviesen),
- los atributos de la clase destino se corresponden con los atributos de la clase origen, es decir, deben tener el mismo nombre, el tipo del atributo de la clase destino debe conformar al tipo del atributo de la clase origen, etc.
- las operaciones entre ambas clases deben corresponderse, es decir, deben tener el mismo nombre, la misma visibilidad, las mismas restricciones, los mismos parámetros, etc.

De la misma manera, para cada interfaz del modelo fuente, existirá una clase abstracta en el modelo destino, tal que ambas se corresponden, es decir tengan el mismo nombre, la misma visibilidad, los mismos elementos miembros, la clase abstracta deberá tener todos sus miembros públicos, etc.

```
Transformation PIM-UML to PSM-EIFFEL {

parameters
    sourceModel: Observer-PIM-Metamodel:: Package
    targetModel: Observer-PSM-EIFFEL-Metamodel:: Package

preconditions
    -- Para el caso general la precondición es true.


postconditions
post:
    -- sourceModel y targetModel tienen el mismo número de clasificadores.
    targetModel.ownedMember-> select(oclIsTypeOf(EiffelClass))-> size()   =
            sourceModel.ownedMember-> select(oclIsTypeOf(Class))-> size() +
            sourceModel.ownedMember-> select(oclIsTypeOf(Interface))-> size()

post:
    -- Para cada interface 'sourceInterface' en sourceModel existe una clase 'targetClass'
    -- en targetModel tal que:
    sourceModel.ownedMember -> select(oclIsTypeOf(Interface))-> forAll ( sourceInterface |
            targetModel.ownedMember-> select(oclIsTypeOf(EiffelClass))-> exists ( targetClass |
```

```
        -- 'targetClass' se corresponde con ' sourceInterface'.
          targetClass.oclAsType(EiffelClass).classInterfaceMatch
                                    (sourceInterface.oclAsType(Interface))  ) )
```

**post:**

```
  -- Para cada clase 'sourceClass' en sourceModel existe una clase 'targetClass' en targetModel
  -- tal que:
  sourceModel.ownedMember -> select(oclIsTypeOf(Class))-> forAll ( sourceClass |
    targetModel.ownedMember -> select(oclIsTypeOf(EiffelClass))-> exists ( targetClass |

          -- 'targetClass' se corresponde con 'sourceClass'.
          targetClass.oclAsType(EiffelClass).classClassMatch
                                    (sourceClass.oclAsType(Class))  ) )
```

**local operations**

**Observer-PSM-EIFFEL-Metamodel::EiffelClass::**
**classClassMatch (aClass: Observer-PIM-Metamodel::Class) : Boolean**

```
classClassMatch (aClass)  =

-- la clase a la cual se aplica esta operación (self) se corresponde con la clase 'aClass'
-- pasada como parámetro si:

-- tienen el mismo nombre,
self.name = aClass.name and

-- tienen el mismo grado de abstracción,
self.isDeferred = aClass.isAbstract and

-- tienen la misma visibilidad,
self.visibility  = aClass.visibility  and

-- tienen las mismas restricciones,
self.invariant  = aClass.ownedRule  and

-- tienen los mismos parámetros,
self.parameters = aClass.ownedTemplateSignature.parameter  and

-- el número de clases padres de self es igual al número de clases padres de aClass  más
-- el número de interfaces que ésta implementa,
self.parents->size()  =
      aClass.superClass->size() + aClass.interfaceRealization.contract->size()   and

-- para cada clase 'sourceParent' padre de aClass, existe una clase 'targetParent' en
-- targetModel  que es padre de self, tal 'targetParent' se corresponde con 'sourceParent',
aClass.superClass-> forAll ( sourceParent |
    self.parents -> exists ( targetParent |
          targetParent.classClassMatch(sourceParent)  ))    and

-- para cada interface 'sourceContract' que aClass  implementa, existe una clase
-- 'targetParent' en targetModel que es padre de self, tal que 'targetParent' se
-- corresponde con 'sourceContract',
aClass.interfaceRealization.contract-> forAll ( sourceContract |
    self.parents-> exists ( targetParent |
          targetParent.classInterfaceMatch(sourceContract)   ))    and

-- el número de rutinas de self es el mismo que el número de operaciones de aClass,
self.ownedRoutine ->size()  = aClass.ownedOperation ->size()   and

-- para cada operación 'sourceOperation' de aClass que no retorna un resultado existe
-- un procedimiento 'targetProcedure' en self que se corresponde con la primera,
aClass.ownedOperation -> select (op |
      not op.ownedParameter->exists (direction = #return) )-> forAll(sourceOperation|
      self.ownedRoutine -> exists ( targetProcedure |
```

```
                    targetProcedure.oclIsTypeOf(Procedure) and
                    targetProcedure.rutineOperationMatch(sourceOperation)  ) )     and
-- para cada operación 'sourceOperation' de aClass que retorna un resultado existe una
-- función 'targetFunction' en self que se corresponde con la primera,
aClass.ownedOperation -> select (op |
        op.ownedParameter->exists (direction = #return) )-> forAll(sourceOperation|
        self.ownedRoutine -> exists ( targetFunction |
                    targetFunction.oclIsTypeOf(Function) and
                    targetFunction.rutineOperationMatch (sourceOperation)  ) )    and
-- el número de atributos más el número de extremos de asociación de self es igual al
-- número de propiedades de aClass,
self.associationEnd->size() + self.attribute-> size() = aClass.ownedAttribute -> size()   and
-- para cada propiedad 'sourceEnd' de aClass, que es un extremo de asociación, existe en
-- self un extremo de asociación 'targetEnd' que se corresponde con el  primero  y
aClass.ownedAttribute -> select (end | end.association -> size () = 1 ) ->
        forAll(sourceEnd | self.associationEnd -> exists ( targetEnd |
                    targetEnd.propertyMatch(sourceEnd)  ) )     and
-- para cada propiedad 'sourceAtt' de aClass, que es un atributo, existe en self un atributo
-- 'targetAtt' que se corresponde con el primero.
aClass.ownedAttribute -> select (att | att.association -> size () = 0 ) ->
        forAll(sourceAtt| self.attribute -> exists ( targetAtt |
                    targetAtt.propertyMatch(sourceAtt)   ) )
```

**Observer-PSM-EIFFEL-Metamodel:: EiffelClass ::**
**classInterfaceMatch (anInterface: Observer-PIM-Metamodel::Interface ) : Boolean**

```
classInterfaceMatch (anInterface)  =
-- la clase a la cual se aplica esta operación (self) se corresponde con la interface
-- 'anInterface' pasada como parámetro si:
-- tienen el mismo nombre,
self.name = anInterface.name   and
-- self es diferida,
self.isDeferred   and
-- tienen la misma visibilidad,
self.visibility = anInterface.visibility   and
-- tienen las mismas restricciones,
self.invariant = anInterface.ownedRule  and
-- tienen los mismos parámetros,
self.parameters = anInterface.ownedTemplateSignature.parameter and
-- para cada interface padre de anInterface existirá una clase padre de self tal que ambas
-- se corresponden,
anInterface.superClass-> forAll ( sourceParent |
       self.parents -> exists ( targetParent |
                    targetParent.classInterfaceMatch (sourceParent)  ))    and
-- el número de rutinas de self es igual que el número de operaciones de anInterface
self.ownedRoutine ->size()  = anInterface.ownedOperation ->size()   and
-- para cada operación 'sourceOperation' de anInterface que no retorna un resultado,
-- existe un procedimiento 'targetProcedure' en self se corresponde con la primera,
        anInterface.ownedOperation -> select (op |
        not op.ownedParameter->exists (direction = #return) )-> forAll(sourceOperation|
        self.ownedRoutine -> exists ( targetProcedure |
```

```
                    targetProcedure.oclIsTypeOf(Procedure) and
                    targetProcedure.routineOperationMatch(sourceOperation)  ) )  and
   -- para cada operación 'sourceOperation' de anInterface que retorna un resultado existe una
   -- función 'targetFunction' en self que se corresponde con la primera,
   anInterface.ownedOperation -> select (op |
         op.ownedParameter->exists (direction = #return) )-> forAll(sourceOperation|
         self.ownedRoutine -> exists ( targetFunction |
                    targetFunction.oclIsTypeOf(Function) and
                    targetFunction.routineOperationMatch(sourceOperation)   ) )  and
   -- el número de atributos más el número de extremos de asociación de self es igual al
   -- número de propiedades de anInterface,
   self.associationEnd->size() + self.attribute->size() = anInterface.ownedAttribute->size()  and
   -- para cada propiedad 'sourceEnd' de anInterface, que es un extremo de asociación,
   -- existe en self un extremo de asociación  'targetEnd' tal que ambas propiedades se
   -- corresponden y
   anInterface.attribute -> select (end | end.association -> size () = 1 ) ->
         forAll(sourceEnd | self.associationEnd -> exists ( targetEnd |
                    targetEnd.propertyMatch(sourceEnd)  ) )    and
   -- para cada propiedad 'sourceAtt' de anInterface, que es un atributo, existe en self un
   -- atributo 'targetAtt' tal que ambas propiedades se corresponden.
   anInterface.attribute -> select (att | att.association -> size () = 0 ) ->
         forAll(sourceAtt| self.attribute -> exists ( targetAtt |
                    targetAtt.propertyMatch(sourceAtt)  ) )
```

**Observer-PSM-EIFFEL-Metamodel:: Routine ::**
**routineOperationMatch(anOperation: Observer-PIM-Metamodel::Operation): Boolean**

```
   routineOperationMatch (anOperation)  =
   -- la rutina a la cual se aplica esta operación (self) se corresponde con la operación
   -- 'anOperation' pasada como parámetro si:
   -- tienen el mismo nombre,
   self.name = anOperation.name and
   -- tienen la misma visibilidad,
   self.visibility = anOperation.visibility   and
   -- tienen el mismo valor para isFrozen e isLeaf,
   self.isFrozen  = anOperation.isLeaf and
   -- tienen las mismas restricciones,
   self.precondition = anOperation.precondition        and
   self.postcondition = anOperation.postcondition     and
   self.bodycondition = anOperation.bodycondition   and
   -- tienen parámetros con los mismos valores de atributos  y tipos que se corresponden y
   anOperation.ownedParameter->forAll ( sourceParam |
         self.ownedParameter->exists( targetParam |
         targetParam.name = sourceParam.name and
         targetParam.direction = sourceParam.direction and
         targetParam.defaultValue = sourceParam.defaultValue and
         targetParam.isOrdered = sourceParam.isOrdered and
         targetParam.upperValue = sourceParam.upperValue and
         targetParam.lowerValue = sourceParam.lowerValue and
         (targetParam.type =sourceParam.type  or
                    targetParam.type.conformsTo(sourceParam.type))     )) and
   -- si anOperation pertenece a una interface implica que self es diferida.
   anOperation.interface -> size() =1 implies  self.isDeferred
```

```
Observer-PSM-EIFFEL-Metamodel:: Property ::
   propertyMatch(aProperty: Observer-PIM-Metamodel::Property) : Boolean

   attributeMatch (aProperty)  =

   -- La propiedad a la cual se le aplica esta operación (self) se corresponde con aProperty
   -- pasada como parámetro si tienen atributos con los mismos  valores y
   self.name = aProperty.name  and
   self.isDerived = aProperty.isDerived and
   self.isReadOnly = aProperty.isReadOnly and
   self.isDerivedOnly = aProperty.isDerivedOnly and
   self.aggregation = aProperty.aggregation and
   self.default = aProperty.default and
   self.isComposite = aProperty.isComposite and
   self.isStatic = aProperty.isStatic and
   self.isOrdered = aProperty.isOrdered and
   self.isUnique = aProperty.isUnique and
   self.upper = aProperty.upper and
   self.lower = aProperty.lower and
   self.ownedRule = aProperty.ownedRule and
   self.isFrozen = aProperty.isLeaf and
   self.visibility = aProperty.visibility and

   -- el tipo de self conforma al tipo de aProperty.
   self.type = aProperty.type    or    self.type.conformsTo(aProperty.type)
```

```
Observer-PSM-EIFFEL-Metamodel:: Type ::
         conformsTo (aType: Observer-PIM-Metamodel:: Type) : Boolean

   conformsTo (aType)  =

   -- Esta operación determina si el tipo al cual se le aplica esta operación (self) y el tipo
   -- pasado como parámetro se corresponden.

   -- Si aType es un tipo de OCL, self podrá corresponder a alguno de los tipos definidos en la
   -- librería de clases Eiffel. Hay dos casos:
   -- Si aType es un tipo primitivo OCL, self podrá corresponder a alguno de los tipos
   -- primitivos definidos en la librería Kernel de Eiffel
   if  aType.oclIsKindOf(Primitive) then (
       aType.oclIsTypeOf(Integer) implies self.oclIsTypeOf(INTEGER) and
       aType.oclIsTypeOf(Real) implies self.oclIsTypeOf(REAL) and
       aType.oclIsTypeOf(String) implies self.oclIsTypeOf(STRING) and
       aType.oclIsTypeOf(Boolean) implies self.oclIsTypeOf(BOOLEAN)
   ) else



       -- Si aType es un tipo colección de OCL, self podrá corresponeder  a alguno de los
       -- tipos colección definidos en la librería de estructuras de datos de Eiffel
       if aType.oclIsKindOf(Collection) then (
           aType.oclIsTypeOf(SetType) implies self.oclIsKindOf( SET ) and
           aType.oclIsTypeOf(OrderedSetType) implies
                           self.oclIsKindOf(TWO_WAY_SORTED_SET) and
           aType.oclIsTypeOf(SequenceType) implies
                           self.oclIsKinkOf( SEQUENCE ) and
           aType.oclIsTypeOf(BagType) implies self.oclIsKindOf( BAG )  )
       endif
   endif
```

### 4.3.3 Transformación PSM-EIFFEL A ISM-EIFFEL

La definición de transformación de PSM a ISM establece que para cada clase del modelo origen (en la plataforma Eiffel) existirá una clase Eiffel en el modelo destino (código Eiffel) tal que ésta es resultado de la traducción de la primera.

Es importante destacar el caso particular de los extremos de asociación propios de una clase que pueden ser traducidos a código de formas diferentes dependiendo de la decisión que tome el usuario respecto de la traducción de las asociaciones. Éstas pueden ser implementadas con un atributo en cada clase asociada que contenga una referencia al objeto relacionado, apropiado en el caso de una asociación binaria con multiplicidad uno a uno, o pueden ser implementadas como una clase diferente, en la cual cada instancia representa un vínculo y sus atributos, la mejor propuesta si la asociación tiene una multiplicidad muchos a muchos. Para el primer caso, cada extremo de asociación perteneciente a una clase será traducido como un atributo privado de dicha clase, con las operaciones *get* y *set* correspondientes (observadora y modificadora) y cuyo tipo conforma al tipo del extremo de asociación. Para el segundo caso, será traducido como un atributo privado y su tipo corresponderá a la clase que surge como resultado de la traducción a Eiffel de la asociación de la cual es parte.

Para llevar a cabo esta propuesta, se propone la definición de componentes reusables específicos para asociaciones a fin de ser utilizados en el proceso de generación de código (Favre, 2005; Favre y otros, 2003). Luego la implementación de una asociación surge de la instanciación del componente correspondiente.

Otro caso que merece particular atención es la traducción de las restricciones. En el caso particular del lenguaje Eiffel, las restricciones OCL pueden ser traducidas como tales (la mayoría de los lenguajes no permite una traducción de las mismas). La definición de transformación establece que para cada restricción establecida para una clase en el modelo origen, existirá una invariante de clase en el modelo destino que resulta de la traducción de la primera. En el contexto de las operaciones, las precondiciones y las postcondiciones pueden ser traducidas como precondiciones y postcondiciones de la operación resultante de la traducción, luego, la definición de transformación establece que para cada precondición establecida para una rutina en el modelo origen, existirá una precondición en la cláusula *require* de una rutina en el modelo destino que resulta de la traducción de la primera. De la misma manera, para cada postcondición establecida para una rutina en el modelo origen, existirá una poscondición en la cláusula *ensure* de la rutina en el modelo destino, de tal forma que ésta es el resultado de la traducción de la primera. Esta traducción es llevada a cabo por medio de la aplicación de esquemas y de heurísticas.

A continuación se muestra parcialmente la definición de transformación de PSM a ISM. Ésta permite transformar un modelo del patrón de diseño *Observer* a nivel PSM, en un modelo del patrón *Observer* en código Eiffel. Dicha definición usa dichos modelos como parámetros origen y destino respectivamente.

```
Transformation PSM-EIFFEL to ISM-EIFFEL {

  parameters
    sourceModel: Observer-PSM-EIFFEL-Metamodel:: Package
    targetModel: Observer-ISM-EIFFEL-Metamodel:: Cluster

  preconditions
    -- Para el caso general la precondición es true.

  postconditions

  post:
    -- targetModel tiene al menos el mismo número de clases que sourceModel
    targetModel.ownedClass -> size()  >=
              sourceModel.ownedMember -> select(oclIsTypeOf(EiffelClass))->  size()

  post:
    -- Para cada clase 'sourceClass' en sourceModel existe una clase 'targetClass'  en targetModel
    -- tal que:
    sourceModel.ownedMember-> select(oclIsTypeOf(EiffelClass))-> forAll ( sourceClass |
      targetModel.ownedClass-> exists ( targetClass |

        -- 'targetClass' se corresponde con 'sourceClass'.
        targetClass.classClassMatch(sourceClass.oclAsType(EiffelClass))
    ) )


  local operations

    Observer-ISM-EIFFEL-Metamodel::EiffelClass ::
    classClassMatch ( aClass: Observer-PSM-EIFFEL-Metamodel::EiffelClass): Boolean

      classClassMatch (aClass)  =

      -- La clase a la cual se aplica esta operación (self) se corresponde con la clase 'aClass'
      -- pasada como parámetro si:

      -- tienen el mismo nombre,
      self.name = aClass.name   and

      -- tienen el mismo grado de abstracción,
      self.isDeferred = aClass.isDeferred   and

      -- si la visibilidad de aClass es package significa que self no tiene clientes fuera del cluster
      -- que lo contiene,
      aClass.visibility= #package implies
          (self.cluster.universe.cluster->excluding(
                 self.cluster)).ownedClass ->forAll ( class |
                        self.supplierDependency.client->excludes(class) )  and

      -- para cada invariante 'sourceInv' de aClass existirá una invariante 'targetInv' en self tal que
      -- 'targetInv' es el resultado  de la traducción de 'sourceInv' a Eiffel,
      aClass.invariant-> forAll(sourceInv | self.invariant-> exists( targetInv |
          targetInv= sourceInv.Translate_Eiffel ()  ) )  and

      -- tienen la misma cantidad de parámetros y para cada parámetro de aClass, existe uno en
      -- self tal que tienen el mismo nombre y su tipos se corresponden,
      self.parameters-> size() = aClass.parameters-> size()   and
      aClass.parameters-> forAll( sourcePar | self.parameters-> exists (
          targetPar | targetPar.name = sourcePar.name and
          targetPar.type.conformsTo (sourcePar.type)    ))   and
```

```
-- para cada clase padre de aClass, existe una clase en targetModel que es padre de self tal
-- que ambas clases se corresponden,
aClass.superClass-> forAll ( sourceParent |
      targetClass.parents -> exists ( targetParent |
            targetParent.classClassMatch(sourceParent)  ))    and

-- para cada rutina de aClass existe una rutina en self tal que ambas se corresponden,
aClass.ownedRoutine-> forAll(sourceRoutine|
      self.ownedRoutine -> exists ( targetRoutine |
            targetRoutine.routineMatch (sourceRoutine)          and

      -- si targetRoutine es de tipo Attach, Detach o Notify, y la clase subject que la declara
      -- tiene una referencia a una clase SubjectObserverAssociation, la rutina tendrá una
      -- invocación a la operación AddLink, RemoveLink o NotifyObservers de dicha clase
      -- según corresponda,
      if targetRoutine.oclIsTypeOf(Attach) and
         targetRoutine.class.subjectObserverReference-> notEmpty() then
            targetRoutine.invokedRoutine -> exists (r | r.oclIsTypeOf(AddLink))
      else if sourceClass.oclIsTypeOf(Detach) and
            targetRoutine.class.subjectObserverReference-> notEmpty() then
            targetRoutine.invokedRoutine->exists (r | r.oclIsTypeOf(RemoveLink))
         else if sourceClass.oclIsTypeOf(Notify) and
               targetRoutine.class.subjectObserverReference-> notEmpty() then
               targetRoutine.invokedRoutine->exists (r |r.oclIsTypeOf (NotifyObservers))
            endif
         endif
      endif
) ) and

-- para cada atributo de aClass, existirá en self un atributo tal que ambos se corresponden,
-- y en caso que el atributo de aClass sea público, también existirá en self una función get
-- y un procedimiento set para manipular dicho atributo,
aClass.ownedAttribute -> forAll(sourceAtt|
      self.attribute -> exists ( targetAtt | targetAtt.attributeMatch(sourceAtt)
         and
         if sourceAtt.visibility = #public then
            self.ownedRoutine -> exists ( aFunction|
               aFunction.oclIsTypeOf(Function) and
               aFunction.name =‘get_’ + sourceAtt.name and
               aFunction.availability = #available  and
               aFunction.argument -> isEmpty()  and
               targetAtt.type.conformTo(aFunction.type)  )
            and
            self.ownedRoutine -> exists (aProcedure|
               aProcedure.oclIsTypeOf(Procedure)   and
               aProcedure.name =‘set_’ + sourceAtt.name and
               aProcedure.availability = #available  and
               aProcedure.argument->exists(arg |arg.name= targetAtt.name and
                     targetAtt.type.conformTo(arg.type)     ) )
         endif
      )   )   and

-- cada extremo de asociación de aClass será traducido a código Eiffel de formas diferentes
-- dependiendo de la decisión que tome el usuario respecto a la traducción de la asociación
-- a la cual éste pertenece, existen dos posibilidades de implementación:
aClass.associationEnd -> forAll(assocEnd |

   -- 1) self tiene un atributo del tipo del extremo de Asociación con sus operaciones
   --    get y set correspondientes
      ( self.attribute-> exists (att |

      -- Si la multiplicidad del extremo de asociación assocEnd es 1, att es un atributo
      -- simple cuyo tipo conforma al tipo del extremo,
```

```
 (assocEnd.multiplicity.upper = 1) implies
                              att.type.conformsTo (assocEnd.type) and

-- si la multiplicidad del extremo de asociación assocEnd es mayor que 1,
-- att es una colección, cuyos elementos conforman al tipo del extremo,
 (assocEnd.multiplicity.upper > 1) implies att.type.oclIsKindOf(COLLECTION) and
                              att.type.elementType.conformsTo (assocEnd.type ) and

-- el nombre de att es igual al nombre de assocEnd,
att.name= assocEnd.name and

-- su disponibilidad es privada,
att.availability = #secret and

-- si assocEnd tiene restricciones, att tendrá aserciones que son el resultado de la
-- traducción de dichas restricciones a Eiffel,
att.assertion = assocEnd.constraint.Translate_Eiffel () and

-- rutinas get y set,
self.ownedRoutine-> exists (aFunction |
            aFunction.oclIsTypeOf(Function) and
            aFunction.name ='get_' + att.name and
            aFunction.availability = #available  and
            aFunction.argument -> isEmpty()  and
            aFunction.type.conformsTo (att.type)  )
        and
self.ownedRoutine -> exists (aProcedure|
            aProcedure.oclIsTypeOf(Procedure) and
            aProcedure.name ='set_' + att.name and
            aProcedure.availability = #available  and
            aProcedure.argument->exists(arg |arg.name = att.name and
                                        arg.type.conformsTo (att.type))  and
) or


-- 2)  self tendrá un atributo privado tal que
self.attribute-> exists (att |

          -- el nombre de att es igual al nombre de assocEnd,
          att.name= assocEnd.name and

          -- su disponibilidad es privada,
          att.availability = #secret and

          -- si assocEnd tiene restricciones, att tendrá aserciones que son el resultado
          -- de la traducción de dichas restricciones a Eiffel y
          att.assertion = assocEnd.constraint.Translate_Eiffel () and

          --  el tipo de att  corresponderá a la clase que surge como resultado de la
          -- traducción de la asociación de la cual es parte.
          att.type = assocEnd.Association.Translate_Eiffel(typeConstructorName,
                                                        <parameter-list>)
          -- Translate_Eiffel traduce la asociación a una clase Eiffel a través de la
          --  instanciación de componentes Eiffel para la asociaciones.
    )
)
```

**Observer-ISM-EIFFEL-Metamodel:: Routine ::**
**routineMatch ( aRoutine: Observer-PSM-EIFFEL-Metamodel:: Routine) : Boolean**

routineMatch (aRoutine) =

*-- la rutina (self) sobre la cual se aplica esta operación se corresponde a la rutina aRoutine pasada como parámetro si:*

*-- ambas rutinas tienen el mismo nombre,*
self.name = aRoutine.name and

*-- tienen visibilidad equivalente,*
self.availability.**isEquivalentTo** (aRoutine.visibility) and

*-- si aRoutine tiene precondiciones, self tiene precondiciones que son el resultado de*
*-- traducir aquellas a Eiffel.*
aRoutine.precondition-> forAll(sourcePre | self.require -> exists (targetPre |
targetPre = sourcePre.$Translate_{Eiffel}$ () ) ) and

*-- las postcondiciones que aparecen en la cláusula ensure de self son el resultado de la*
*-- traducción de las poscondiciones y bodycondition de aRoutine*
aRoutine.postcondition->union (self.bodycondition) -> forAll (
sourcePost | self.ensure -> exists ( targetPost |
targetPost = sourcePost.$Translate_{Eiffel}$ () ) ) and

*--aRoutine y self tienen el mismo número de parámetros y para cada parámetro de aRoutine*
*-- existe un argumento en self tal que el tipo de éste conforma al tipo del parámetro*
*-- correspondiente y*
self.argument ->size()= aRoutine.ownedParameter->size()and
aRoutine.ownedParameter->forAll(par| self.argument-> exists (arg |
arg.name= par.name and arg.type.**conformsTo**(par.type) )) and

*-- si aRoutine es una función, el tipo de self debe conformar al tipo de sourceRoutine.*
aRoutine.oclIsTypeOf(Function) **implies**
self.type **conformsTo**(aRoutine.returnType)

**Observer-ISM-EIFFEL-Metamodel:: Attribute ::**
**attributeMatch ( anAttribute: Observer-PSM-EIFFE-Metamodel :: Attribute): Boolean**

attributeMatch (anAttribute) =

*-- el atributo (self) sobre el cual se aplica esta operación se corresponde al atributo*
*-- 'anAttribute' pasado como parámetro si:*

*-- tienen el mismo nombre,*
self.name = anAttribute.name and

*-- self es privado,*
self.availability = #secret and

*-- para cada restricción de anAttribute existirá una invariante correspondiente a self que*
*-- es el resultado de la traducción a Eiffel de la primera*
anAttribute.ownedRule->forAll(rule | self.assertion->exists(ass | ass= rule.$Translate_{Eiffel}$() ) )
and

*-- sourceAtt tiene el mismo valor isFrozen que self y*
self.isFrozen = anAttribute.isFrozen and

*-- el tipo de self conforma al tipo de sourceAtt.*
self.type.**conformsTo**(anAttribute.type)

**Observer-ISM-EIFFEL-Metamodel:: Type ::**
**conformsTo ( aType: Observer-PSM-EIFFE-Metamodel:: EiffelClass ) : Boolean**

```
conformsTo (aType) =

-- verifica si el tipo (self) sobre el cual se aplica esta operación se corresponde al tipo
-- aType pasado como parámetro.

-- si aType es un tipo de OCL, self deberá corresponder con alguno de los tipos definidos
-- en la librería de clases Eiffel. Hay dos casos:
-- Si aType es un tipo primitivo OCL, self corresponderá a alguno de los tipos primitivos
-- definidos en la librería Kernel de Eiffel

if aType.oclIsKindOf(Primitive) then (
    aType.oclIsTypeOf(Integer) implies self.oclIsTypeOf(INTEGER) and
    aType.oclIsTypeOf(Real) implies self.oclIsTypeOf(REAL) and
    aType.oclIsTypeOf(String) implies self.oclIsTypeOf(STRING) and
    aType.oclIsTypeOf(Boolean) implies self.oclIsTypeOf(BOOLEAN)
    ) else

    -- Si aType es un tipo colección de OCL, self corresponderá a alguno de los tipos
    -- colección definidos en la librería de estructuras de datos de Eiffel
    if aType.oclIsKindOf(Collection) then (
        aType.oclIsTypeOf(SetType) implies self.oclIsKindOf( SET ) and
        aType.oclIsTypeOf(OrderedSetType) implies
                        self.oclIsKindOf(TWO_WAY_SORTED_SET) and
        aType.oclIsTypeOf(SequenceType) implies
                        self.oclIsKinkOf( SEQUENCE ) and
        aType.oclIsTypeOf(BagType) implies self.oclIsKindOf( BAG )
        ) else

        -- si aType es un tipo declarado por el usuario o es un tipo perteneciente a la
        -- librería Eiffel, self y aType deben ser iguales
        self = aType
    endif
endif
```

**Observer-ISM-EIFFEL-Metamodel:: FeatureAvailability ::**
**isEquivalentTo ( aVisibility: Observer-PSM-EIFFE-Metamodel:: Visibility): Boolean**

```
isEquivalent (aVisibility) =

-- si aVisibility es privada o protegida, self será privada y
(aVisibility= #private or aVisibility= #protected) implies self = #secret    and

-- si aVisibility es public, self será pública.
aVisibility= #public implies self = #available
```

# 5. Formalizando Componentes MDA

## 5.1. Introducción

El desarrollo de componentes reusables requiere poner énfasis sobre la calidad del software, por lo cual las técnicas tradicionales para la verificación y validación son esenciales para lograr atributos de calidad en el software, tales como consistencia, correctitud, robustez, reusabilidad, eficiencia, confiabilidad, etc.

En general, una propuesta de reuso no puede ser exitosa sin un grado de formalismo. Un componente no puede ser entregado sin por lo menos la forma básica de especificación: el contrato. Un contrato indica lo que el componente espera de sus clientes (la precondición), las propiedades que el resultado asegura (la poscondición), y las condiciones que éste mantiene (invariantes). La analogía con el campo del hardware es bastante clara en este punto: ¿Cómo haría un ingeniero de hardware para reusar un componente electrónico sin una especificación impecable de sus entradas, salidas y sus condiciones para el correcto funcionamiento? (Meyer, 1997b).

En el contexto de los procesos MDD, los formalismos pueden ser usados para detectar inconsistencias tanto en los modelos internos o entre el modelo origen y el modelo destino que ha sido obtenido a través de transformaciones de modelos. Una especificación formal clarifica el significado deseado de los metamodelos y de las transformaciones de modelos basadas en metamodelos ayudando a validarlos y proveyendo una referencia para la implementación.

UML y OCL son imprecisos y ambiguos cuando se trata de la simulación, verificación, validación y predicción de las propiedades del sistema y más aún si se trata de generar modelos o implementaciones a través de transformaciones. La falta de formalización de UML es referida en la Infraestructura del lenguaje donde se cita: "It is important to note that the specification of UML as a metamodel does not preclude it from being specified via a mathematically formal language (e.g., Object-Z or VDM) at a later time" (UML-Infrastructure, 2007, p.11).

Desde la aparición de UML han surgido distintas opiniones acerca de la falta de precisión en la definición semántica que dieron origen a distintas propuestas de formalización de UML, o partes del mismo a través de la utilización de distintos formalismos como Z (Shroff y France, 1997), Object-Z (Kim y Carrington, 1999), B (Snook y Butler (2000)), CASL (Hussmann y otros, 1999; Padawitz, 2000) También han habido propuestas de extensiones y/o modificaciones a UML para lograr la formalización de mismo (Opdahl y otros, 2001; Bruel y otros, 2001; Stevens, 2001). Junto con UML surge pUML (The Precise UML Group) cuyo objetivo fue dar precisión a UML, y el cual se encuentra actualmente trabajando en esta tarea.

Con la aparición de OCL también surgieron trabajos que discuten la falta de precisión semántica del mismo (Mandel y Cengarle, 1999) como así también surgieron numerosas propuestas de formalización (Bidoit y otros, 1999; Hamie y otros, 1998; Richters y Gogolla, 1998; Clark 1999; Cengarle y Knapp, 2002 ).

Más tarde con la aparición de MDA, surgieron también trabajos sobre la formalización del metamodelo de UML en distintos lenguajes (Emerson y Sztipanovits, 2004; Poernomo, 2006).

La integración de especificaciones semiformales UML/OCL y los lenguajes formales ofrece lo mejor de ambos mundos (Favre y otros, 2001; Favre y otros, 2002; Favre y otros, 2003; Favre y otros, 2004; Martinez y Favre, 2004; Favre y otros, 2005). En esta dirección, se propone la utilización de la notación de metamodelado NEREUS y la utilización de un sistema de reglas de transformación para transformar metamodelos MOF en NEREUS que son el resultado de investigaciones previas (Favre, 2005, Favre, 2007).

Se optó por la utilización del lenguaje de metamodelado NEREUS por varios motivos:

- Está alineado con MOF: NEREUS y MOF tienen mecanismos de estructuración similares, la mayoría de los conceptos de los metamodelos basados en MOF pueden ser traducidos a NEREUS en una forma directa.
- Puede ser visto como una notación intermedia abierta a muchos otros lenguajes formales, lo que facilita la interoperabilidad con lenguajes formales clásicos. En particular se definió una forma de traducir automáticamente cada construcción NEREUS a CASL (Favre, 2006) lo que permitiría el uso de las herramientas que dan soporte al mismo.
- La integración de OCL con un lenguaje algebraico como NEREUS permite una sólida definición del sistema de tipos, relaciones de herencia, subtipo, refinamientos entre especificaciones.

En este capítulo se muestra como formalizar los componentes definidos para patrones de diseño a través de la formalización de los metamodelos MOF y de las transformaciones de modelos basadas en metamodelos usando la notación de metamodelado NEREUS (Favre y Martinez, 2006). La propuesta es ilustrada usando el patrón de diseño Observer.

En el anexo C se muestra parcialmente la especificación del metamodelo UML en NEREUS.

## 5.2. Formalizando metamodelos MOF

Los metamodelos MOF y NEREUS tienen construcciones y mecanismos de estructuración similares. Cada paquete en el metamodelo es traducido a un paquete NEREUS. Cada clase o asociación en el metamodelo es traducido en una clase o una asociación en NEREUS. Todo el proceso de traducción que se detalla a continuación se basa en el resultado de trabajos previos. Un análisis detallado puede ser encontrado en (Favre, 2005; Favre. 2007).

El proceso de traducción de especificaciones OCL a NEREUS es soportado por un sistema de reglas de transformación (Favre, 2005). En los metamodelos, las especificaciones OCL pueden aparecer como precondiciones, poscondiciones o invariantes de clases, restricciones de atributos y restricciones de asociaciones. Analizando especificaciones OCL se pueden derivar axiomas que serán incluidos en las especificaciones NEREUS. Las precondiciones escritas en OCL se usan para generar precondiciones en NEREUS. Las poscondiciones e invariantes permiten generar axiomas en NEREUS.

La Figura 5.1 grafica las etapas de la traducción de UML/OCL a NEREUS. El texto de la especificación NEREUS se va completando gradualmente hasta obtener una especificación que refleja toda la información del diagrama UML. Primero, se obtiene la signatura de la clase y axiomas vinculados a los atributos de la clase UML a partir de la instanciación de un esquema reutilizable denominado *BOX_*. Luego, se construyen las especificaciones de asociaciones instanciando esquemas reutilizables y subcomponentes del componente

*Association*. Finalmente se traducen las especificaciones OCL aplicando reglas de un sistema de transformación de OCL a NEREUS, las cuales consisten en pares de esquemas, uno en OCL y el otro en NEREUS.

Se describe a continuación la transformación de un *package* UML básico (que no depende de otros) y que incluye sólo clases y sus relaciones.

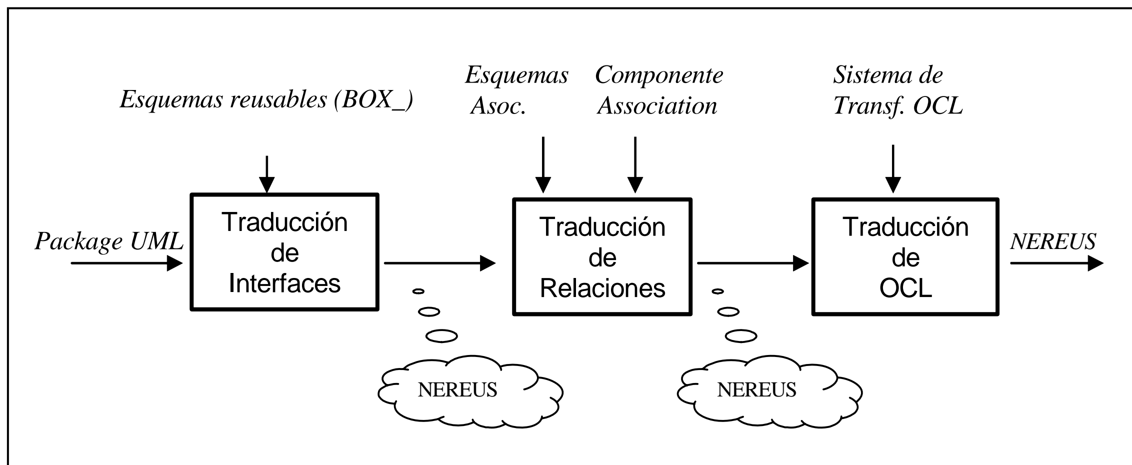



**Figura 5.1.** Transformaciones de UML/OCL a NEREUS

### 5.2.1. Traducción de clases UML a NEREUS

La Figura 5.2 muestra el esquema BOX_. La cláusula IMPORTS expresa relaciones de dependencia, las cláusulas IS-SUBTYPE-OF e INHERITS expresan relaciones de herencia y la cláusula ASSOCIATES asociaciones.

Las instanciaciones de BOX_ definen una operación constructora *create*, cuya funcionalidad se obtiene traduciendo cada tipo de atributo. Los atributos de una clase son traducidos a dos operaciones, una operación de acceso y una operación modificadora. En NEREUS no existen convenciones estándar, pero en general se usan nombres tales como *get_* y *set_* para las mismas.

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 $TP_1$, $TP_2$,..., $TP_n$ con los tipos de los argumentos de las operaciones. Esta información es extraída de la especificación de la clase en el diagrama.

### 5.2.2. Traducción de asociaciones UML a NEREUS

La traducción de las asociaciones se realiza a partir del diagrama de clases y el componente Association. La información registrada en el diagrama de clases permite seleccionar de estos componentes una especificación particular que será instanciada con la información específica utilizando el esquema de la Figura 5.3.

**CLASS** *BOX_*<br>
**IMPORTS** $TP_1,\ldots, TP_m, T\text{-}attr_1, T\text{-}attr_2,\ldots, T\text{-}attr_n$<br>
**IS-SUBTYPE-OF** *B1, B2,…, Bm*<br>
**INHERITS** *C1, C2,…, Cr*<br>
**ASSOCIATES**<br>
<<*Aggregation-*$E_1$>>,…, <<*Aggregation-*$E_p$>>,<br>
<<*Composition-*$C_1$>>,…, <<*Composition-*$C_k$>>,<br>
<<*Association-*$D_1$>>,…, <<*Association-*$D_s$>><br>
**EFFECTIVE**<br>
**TYPE** Name<br>
**FUNCTIONS**<br>
createName: $T\text{-}attr_1 \times T\text{-}attr_2, \times \ldots \times T\text{-}attr_n \rightarrow Name$<br>
$set_i : Name \times T\text{-}attr_i \rightarrow Name$<br>
$get_i : Name \rightarrow T\text{-}attr_i$ -- $1 \le i \le n$<br>
**DEFERRED**<br>
**FUNCTIONS**<br>
$meth_1: Name \times Tpi_1 \times Tpi_2 \times \ldots \times Tpi_n \rightarrow Tpi_j$<br>
…<br>
$meth_r: Name \times Tpr_1 \times Tpr_2 \times \ldots \times Tpr_n \rightarrow Tpr_j$<br>
**AXIOMS** $t_1, t_1' : T\text{-}attr_1; t_2, t_2' : T\text{-}attr_2; \ldots; t_n, t_n' : T\text{-}attr_n$<br>
$get_i(create(t_1,t_2,\ldots,t_n)) = t_i$ -- $1 \le i \le n$<br>
$set_i(create(t_1,t_2,\ldots,t_i,\ldots,t_n), t_{i'}) = create(t_1,t_2,\ldots, t_{i'}, \ldots,t_n)$<br>
**END-CLASS**

**Figura 5. 2.** Esquema reusable BOX_

**ASSOCIATION** __<br>
**IS** __ [_:Class1;_:Class2;_:Role1;_:Role2;_:Mult1;_:Mult2;_:Visibility1;_: Visibility2]<br>
**CONSTRAINED BY** __<br>
**END**

**Figura 5. 3.** Traducción de asociaciones UML a NEREUS

La cláusula IS expresa una relación de instanciación de la relación seleccionada a partir de una lista de renombres. La cláusula CONSTRAINED-BY posibilita especificar *constraints* estáticos en lógica de primer orden que pueden ser aplicados a la asociación. Una asociación puede tener sus propias operaciones y propiedades, es decir operaciones que no pertenecen a las clases asociadas, sino a la asociación misma. Este tipo de asociación se la denomina ASSOCIATION CLASS.

### 5.2.3. Traducción de restricciones OCL a NEREUS

OCL no puede interpretarse independientemente de UML. En el contexto de los modelos estáticos UML las restricciones OCL aparecen asociadas a operaciones en precondiciones y postcondiciones, en invariantes de clases, en atributos y asociaciones.

Como ya se mencionó las especificaciones OCL se traducen a NEREUS aplicando reglas de un sistema de transformación las cuales consisten en pares de esquemas, uno en OCL y el otro en NEREUS. La Figura 5.4 resume la traducción de algunas expresiones OCL simples a

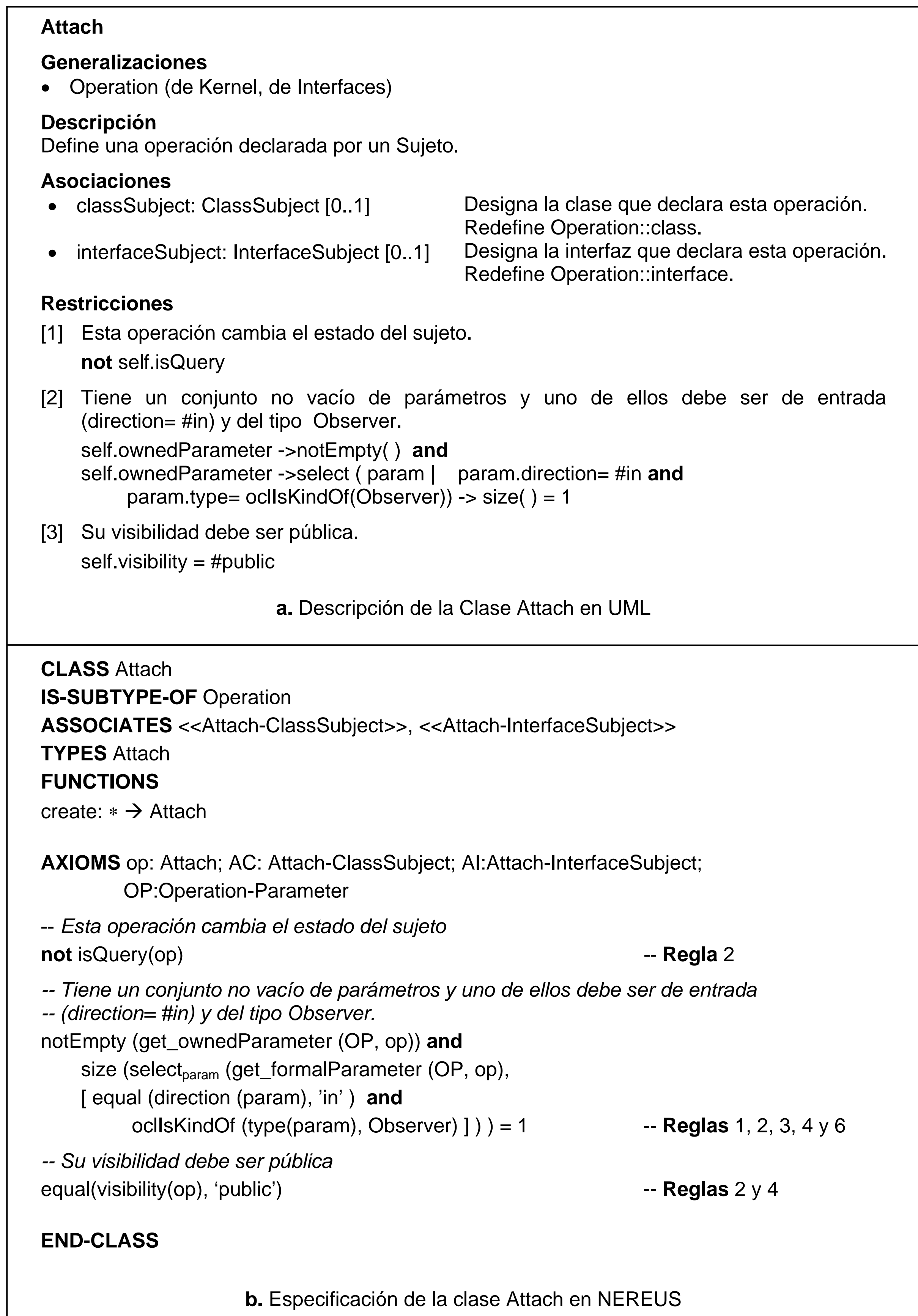
NEREUS, donde la primer columna contiene el número de la regla, la segunda una expresión OCL y la tercera la expresión resultante en NEREUS. Las funciones $\text{Translate}_{\text{NEREUS}}$ son las encargadas de traducir expresiones lógicas de OCL en fórmulas en primer orden en NEREUS.

### 5.2.4. Traducción de la clase *Attach* a NEREUS

Para ejemplificar la traducción de una clase UML a NEREUS, en esta sección se detalla como traducir a NEREUS la clase Attach del metamodelo del patrón *Observer* a nivel PIM especificada en el capítulo 4 (ver Figura 4.5.b). La Figura 5.5.a transcribe la descripción de esta clase que aparece en el Capítulo 4 (página 34) y la Figura 5.5.b muestra la clase Attach resultante de la traducción a NEREUS.

| Regla | OCL | NEREUS |
|---|---|---|
| 1 | v. operation (v') (*v es una variable*)<br>v->operation (v') | operation (v, v') |
| 2 | v. attribute | attribute (v) |
| 3 | **context** A (*A es una asociación*)<br>object.rolename | get_ rolename (a, object)<br>*Let a:A* |
| 4 | OCLexp1 opBin OCLexp2<br>opbin::= and \| or \| xor \| = \| <> \| < \|<br><= \| > \| >= \| + \| - \| / \| * | $\text{Translate}_{\text{NEREUS}}$ (OCLexp1) opBin<br>$\text{Translate}_{\text{NEREUS}}$ (OCLexp2)<br>o<br>$\text{Translate}_{\text{NEREUS}}$ (opBin) (<br>$\text{Translate}_{\text{NEREUS}}$ (OCLexp1) ,<br>$\text{Translate}_{\text{NEREUS}}$ (OCLexp2) ) |
| 5 | e.op (*e es una expresión OCL*) | op (Translate $_{\text{NEREUS}}$ (e)) |
| 6 | collection->op(v:Elem\|bool-expr-with-v)<br><br>op ::=select \| forAll \| reject \| exists | **LET**<br>**FUNCTIONS**<br>f: Elem -> Boolean<br>**AXIOMS** v : Elem<br>f (v)=Translate $_{\text{NEREUS}}$ (bool-expr-with-v )<br>**IN** op (collection, f)<br>**END-LET**<br>---<br>$op_v$ (collection,<br>[$\text{Translate}_{\text{NEREUS}}$ (bool-expr-with-v ) ]<br>*Notación concisa equivalente* |

**Figura 5.4.** Reglas para expresiones OCL simples



**Attach**

**Generalizaciones**

- Operation (de Kernel, de Interfaces)

**Descripción**

Define una operación declarada por un Sujeto.

**Asociaciones**

- classSubject: ClassSubject [0..1] — Designa la clase que declara esta operación. Redefine Operation::class.
- interfaceSubject: InterfaceSubject [0..1] — Designa la interfaz que declara esta operación. Redefine Operation::interface.

**Restricciones**

[1] Esta operación cambia el estado del sujeto.

**not** self.isQuery

[2] Tiene un conjunto no vacío de parámetros y uno de ellos debe ser de entrada (direction= #in) y del tipo Observer.

self.ownedParameter ->notEmpty( ) **and**
self.ownedParameter ->select ( param | param.direction= #in **and**
param.type= oclIsKindOf(Observer)) -> size( ) = 1

[3] Su visibilidad debe ser pública.

self.visibility = #public

**a.** Descripción de la Clase Attach en UML

**CLASS** Attach
**IS-SUBTYPE-OF** Operation
**ASSOCIATES** <<Attach-ClassSubject>>, <<Attach-InterfaceSubject>>
**TYPES** Attach
**FUNCTIONS**
create: ∗ → Attach

**AXIOMS** op: Attach; AC: Attach-ClassSubject; AI:Attach-InterfaceSubject;
OP:Operation-Parameter

-- *Esta operación cambia el estado del sujeto*
**not** isQuery(op) -- **Regla** 2

-- *Tiene un conjunto no vacío de parámetros y uno de ellos debe ser de entrada*
-- *(direction= #in) y del tipo Observer.*
notEmpty (get_ownedParameter (OP, op)) **and**
size (select$_{param}$ (get_formalParameter (OP, op),
[ equal (direction (param), 'in' ) **and**
oclIsKindOf (type(param), Observer) ] ) ) = 1 -- **Reglas** 1, 2, 3, 4 y 6

-- *Su visibilidad debe ser pública*
equal(visibility(op), 'public') -- **Reglas** 2 y 4

**END-CLASS**

**b.** Especificación de la clase Attach en NEREUS

**Figura 5.5.** Construyendo la clase Attach en NEREUS

El primer paso es la instanciación del esquema BOX_:

- La cláusula IS-SUBTYPE-OF expresa relaciones de herencia, la clase Attach hereda de la clase Operation por lo tanto será instanciada con ella.
- Attach está asociada con las clases ClassSubject e InterfaceSubject, por lo tanto la cláusula ASSOCIATES será instanciada con ambas asociaciones, Attach-ClassSubject y Attach-InterfaceSubject.
- La cláusula TYPE es instanciada con el nombre de la clase: Attach.
- La interfaz de la clase se obtiene especificando una operación constructora *create* junto con las operaciones definidas por la clase.
- La clase que hereda completa la constructora de sus padres, por lo tanto la funcionalidad de *create* se obtiene de las propiedades heredadas más la traducción de cada tipo de sus atributos propios. La clase Attach no tiene atributos propios, por lo tanto la funcionalidad de create puede ser denotada por un asterisco para indicar solamente las propiedades heredadas.

El siguiente paso es la traducción de las restricciones OCL de la clase Attach a axiomas en la especificación.

Para ejemplificar, se detallará como traducir la restricción número 2 aplicando las reglas de transformación mostradas en la Figura 5.4. La restricción establece que una operación el tipo Attach tiene un conjunto no vacío de parámetros y uno de ellos debe ser de entrada (direction= #in) y del tipo Observer:

```
self.ownedParameter->notEmpty( ) and
self. ownedParameter ->select ( param | param.direction= #in and
            param.type.oclIsKindOf(Observer) ) -> size( ) = 1
```

Está formada por la conjunción de dos expresiones booleanas (regla 4):

1. La primera expresión OCL:

   ```
   self.ownedParameter->notEmpty( )
   ```

   se corresponde con el esquema OCL de la regla número 1:

   ```
   v->operation(),
   ```

   donde v es la expresión self.ownedParameter y la operación es notEmpty. La expresión self.ownedParameter se corresponde a su vez con el esquema de la expresión OCL de la regla número 3:

   ```
   context A  -- A es una asociación
   object.rolename
   ```

   donde object es una instancia de la clase Attach (self), y rolename es ownedParameter, es decir la colección formada por los parámetros de dicha instancia. Esta colección se obtiene a través de la operación get_ownedParameter de la asociación entre la clase Operation y la clase Parameter.
   Sea op una instancia de la clase attach y OP una instancia de la asociación Operation-Parameter y aplicando la regla 3 se obtiene:

   ```
   get_ownedParameter(OP,op)
   ```

y luego aplicando la regla 1 se obtiene:

notEmpty (get_ownedParameter (OP, op))

2. La segunda expresión OCL:

   self.ownedParameter ->select ( param | param.direction= #in **and**

   param.type.oclIsKindOf(Observer) ) -> size( ) = 1

   se corresponde con el esquema OCL de la regla 4, OCLexp1 opBin OCLexp2, donde:

   - OCLexp1 es la expresión self.ownedParameter ->select ( param | param.direction= #in and param.type.oclIsKindOf(Observer) )
   - opBin es el operador binario '='
   - OCLexp2 es el número 1

   OCLexp1 se corresponde con el esquema OCL de la regla número 6:

   collection->select(v:Elem | bool-expr-with-v)

   En esta última expresión:

   - la colección es self.ownedParameter que se corresponde con el esquema OCL de la regla 3 que como se mencionó se traduce a get_ownedParameter(OP,op) y
   - bool-expr-with-v es param.direction= #in and param.type.oclIsKindOf(Observer). Ésta es la conjunción de dos expresiones OCL (regla 4), la primera se corresponde al esquema OCL de las reglas números 2 y 4 y la segunda se corresponde con la regla 1 v.operation(v'), donde v es param.type (esquema OCL de la regla 2) y v' es Observer.

Aplicando las reglas mencionadas a la restricción OCL, se obtiene la siguiente expresión en NEREUS:

notEmpty (get_ownedParameter (OP, op)) and

size (selectparam (get_formalParameter (OP, op), [ equal (direction (param), 'in' )

and oclIsKindOf (type(param), Observer) ] ) ) = 1

### 5.2.5. Traducción de la asociación *AssocEndSubject-SubjectObserver* a NEREUS

La traducción de las asociaciones se realiza a partir del diagrama de clases y el componente Association. Para ejemplificar, se muestra como traducir la asociación *AssocEndSubject-SubjectObserver* que vincula las metaclases AssocEndSubject y SubjectObserver del metamodelo del patrón *Observer* a nivel PIM (Figura 4.5.a). La Figura 5.6 muestra la parte del diagrama de clases UML correspondiente a dicha asociación.

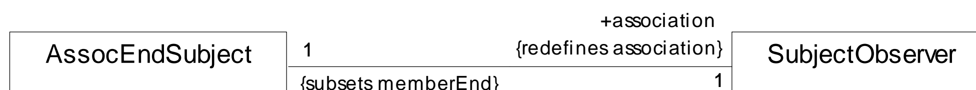



**Figura 5.6.** Asociación *AssocEndSubject-SubjectObserver*

Del diagrama se desprende que se trata de una asociación binaria, bidireccional. Se conocen los nombres, la multiplicidad, la visibilidad y las restricciones de sus roles. Esta información permite seleccionar el componente correspondiente, en este caso el componente Bidirectional-1 (capítulo 3, Figura 3.5), el cual será instanciado con la información específica utilizando el esquema mostrado en la Figura 5.3. La instanciación es la siguiente:

**ASSOCIATION** *AssocEndSubject-SubjectObserver*

**IS** *Bidirectional-1* [ *AssocEndSubject*: Class1; *SubjectObserver*: Class2;
*assocEndSubject*: Role1; *association*: Role2;
*1*: Mult1; *1*: Mult2;
+: Visibility1; +: Visibility2]

**CONSTRAINED-BY**

association: redefines association

assocEndSubject: subsets memberEnd

**END**

### 5.2.6. Formalizando el Metamodelo del Patrón de Diseño Observer

En esta sección se muestra la especificación en NEREUS del metamodelo del patrón *Observer* del nivel PIM descripto en el capítulo previo (Figura 4.5). Los nombres de las asociaciones hacen referencia a las clases asociadas, por ejemplo *AssocEndConcreteObserver-ConcreteObserver* denota una asociación entre las clases *AssocEndConcreteObserver* y *ConcreteObserver*. Las clases y las asociaciones dentro del paquete son presentadas por orden alfabético.

**PACKAGE** ObserverMetamodel
**IMPORTS** Kernel, Interfaces, Dependencies

***-- Especificación de las metaclases***

**CLASS** AssocEndConcreteObserver
**IS-SUBTYPE-OF** Property
**ASSOCIATES**
<<AssocEndConcreteObserver-ConcreteObserver>>,
<<AssocEndConcreteObserver-ObserverSubject>>
**GENERATED_BY** create
**TYPES** AssocEndConcreteObserver
**FUNCTIONS**
create: $*$ → AssocEndConcreteObserver
**AXIOMS** assEnd: AssocEndConcreteObserver
get_lower(assEnd) >= 0 and get_upper(assEnd)>= 1
**END-CLASS**

**CLASS** AssocEndConcreteSubject
**IS-SUBTYPE-OF** Property
**ASSOCIATES**
<<AssocEndConcreteSubject-ConcreteSubject>>, <<AssocEndConcreteSubject-ObserverSubject>>
**GENERATED_BY** create
**TYPES** AssocEndConcreteSubject
**FUNCTIONS**
create: $*$ → AssocEndConcreteSubject
**AXIOMS** assEnd: AssocEndConcreteSubject
get_lower(assEnd) >=0 and get_upper(assEnd)>= 1
isNavigable(assEnd) = True
**END-CLASS**

**CLASS** AssocEndObserver
**IS-SUBTYPE-OF** Property
**ASSOCIATES**
<<AssocEndObserver-SubjectObserver>>, <<AssocEndObserver-Observer>>
**GENERATED_BY** create
**TYPES** AssocEndObserver
**FUNCTIONS**
create: ∗ → AssocEndObserver
**AXIOMS** assEnd: AssocEndObserver
get_lower(assEnd) >=0 and get_upper(assEnd)>= 1
get_isNavigable(assEnd) = True
**END-CLASS**

**CLASS** AssocEndSubject
**IS-SUBTYPE-OF** Property
**ASSOCIATES**
<<AssocEndSubject-SubjectObserver>>, <<AssocEndSubject-Subject>>
**GENERATED_BY** create
**TYPES** AssocEndSubject
**FUNCTIONS**
create: ∗ → AssocEndSubject
**AXIOMS** assEnd: AssocEndSubject
get_lower(assEnd) >=0 and ( get_upper(assEnd)>= 1
**END-CLASS**

**CLASS** Attach
**IS-SUBTYPE-OF** Operation
**ASSOCIATES** <<Attach-ClassSubject>>, <<Attach-InterfaceSubject>>
**TYPES** Attach
**FUNCTIONS**
create: ∗ → Attach
**AXIOMS**
op:Attach; OP:Operation-Parameter
not isQuery(op)
notEmpty (get_formalParameter (OP, op))
    and size ($select_{param}$ (get_formalParameter (OP, op),
    [equal(direction (param),'in') and oclIsKindOf(type(param), Observer)] )) = 1
equal(visibility(op), 'public')
**END-CLASS**

**CLASS** ClassObserver
**IS-SUBTYPE-OF** Class, Observer
**ASSOCIATES**
<<ClassObserver-Update>>
**GENERATED_BY** create
**TYPES** ClassObserver
**FUNCTIONS**
create: ∗ → ClassObserver
**END-CLASS**

**CLASS** ClassSubject
**IS-SUBTYPE-OF** Class, Subject
**ASSOCIATES**
<< Attach-ClassSubject>>, <<ClassSubject-Detach>>, <<ClassSubject-Notify>>
**GENERATED_BY** create
**TYPES** ClassSubject
**FUNCTIONS**
create: ∗ → ClassSubject
**END-CLASS**

**CLASS** ConcreteObserver
**IS-SUBTYPE-OF** Class
**ASSOCIATES**
<<AssocEndConcreteObserver-ConcreteObserver>>,<<ConcreteObserver-GeneralizationObserver>>,
<<ConcreteObserver-InterfaceRealizationObserver>>
**TYPES** ConcreteObserver
**FUNCTIONS**
create: ∗ → ConcreteObserver
**AXIOMS** obs: ConcreteObserver; CI: ConcreteObserver-InterfaceRealizationObserver
not isAbstract(obs)
notEmpty(get_interfaceRealizationObserver(CI,obs)) => oclIsKindOf(obs,BehavioredClassifier)
**END-CLASS**

**CLASS** ConcreteSubject
**IS-SUBTYPE-OF** Class
**ASSOCIATES**
<<AssocEndConcreteSubject-ConcreteSubject>>, <<ConcreteSubject-SetState>>,
<<ConcreteSubject-GeneralizationSubject>>, <<ConcreteSubject-GetState>>,
<<ConcreteSubject-InterfaceRealizationSubject>>, <<ConcreteSubject-Property>>
**TYPES** ConcreteSubject
**FUNCTIONS**
create: ∗ → ConcreteSubject
**AXIOMS** sub:ConcreteSubject; CP: ConcreteSubject-Property; AP: Association-Property;
CI: ConcreteSubject-InterfaceRealizationSubject
not isAbstract(sub)
notEmpty(get_interfaceRealizationObserver(CI,sub)) => oclIsKindOf(sub,BehavioredClassifier)
forAll$_p$ (get_state (CP,sub), [ isEmpty(get_association(AP,p) ] )
**END-CLASS**

**CLASS** Detach
**IS-SUBTYPE-OF** Operation
**ASSOCIATES** <<ClassSubject-Detach>>, <<Detach-InterfaceSubject>>
**TYPES** Detach
**FUNCTIONS**
create: ∗ → Detach
**AXIOMS**
op: Detach; OP:Operation-Parameter
not isQuery(op)
notEmpty (get_formalParameter (OP, op))
and size (select$_{param}$ (get_formalParameter (OP, op),
[equal(direction (param),'in') and (oclIsKindOf(type(param), Observer) ] ) ) = 1

equal(visibility(op), ‘public’)
**END-CLASS**

**CLASS** GeneralizationObserver
**IS-SUBTYPE-OF** Generalization
**ASSOCIATES**
<<ClassObserver-GeneralizationObserver>>, << ConcreteObserver-GeneralizationObserver>>
**GENERATED_BY** create
**TYPES** GeneralizationObserver
**FUNCTIONS**
create: ∗ → GeneralizationObserver
**END-CLASS**

**CLASS** GeneralizationSubject
**IS-SUBTYPE-OF** Generalization
**ASSOCIATES**
<<ClassSubject-GeneralizationSubject>>, <<ConcreteSubject-GeneralizationSubject>>
**GENERATED_BY** create
**TYPES** GeneralizationSubject
**FUNCTIONS**
create: ∗ → GeneralizationSubject
**END-CLASS**

**CLASS** GetState
**IS-SUBTYPE-OF** Operation
**GENERATED_BY** create
**TYPES** GetState
**FUNCTIONS**
create: ∗ → GetState
**AXIOMS** op:GetState; CO: Class-Operation; CI: Class-Interface; OP:Operation-Parameter
isQuery(op)
notEmpty(get_ownedParameter(OP, op) and
size( $select_{par}$(get_ownedParameter(OP, op) ,
[equal(direction(par), ‘return’) or equal(direction(par) , ‘out’)] ) )  >= 1
equal(visibility(op), ‘public’)
**END-CLASS**

**CLASS** InterfaceObserver
**IS-SUBTYPE-OF** Observer, Interface
**ASSOCIATES**
<<InterfaceObserver-Update>>
**GENERATED_BY** create
**TYPES** InterfaceObserver
**FUNCTIONS**
create: ∗ → InterfaceObserver
**END-CLASS**

**CLASS** InterfaceSubject
**IS-SUBTYPE-OF** Subject, Interface
**ASSOCIATES**
<<Attach-InterfaceSubject >>, <<Detach-InterfaceSubject>>, << InterfaceSubject-Notify>>

**GENERATED_BY** create
**TYPES** InterfaceSubject
**FUNCTIONS**
create: ∗ → InterfaceSubject
**END-CLASS**

**CLASS** InterfaceRealizationObserver
**IS-SUBTYPE-OF** InterfaceRealization
**ASSOCIATES**
<<InterfaceObserver-InterfaceRealizationObserver>>,
<<ConcreteObserver-InterfaceRealizationObserver>>
**GENERATED_BY** create
**TYPES** InterfaceRealizationObserver
**FUNCTIONS**
create: → InterfaceRealizationObserver
**END-CLASS**

**CLASS** InterfaceRealizationSubject
**IS-SUBTYPE-OF** InterfaceRealization
**ASSOCIATES**
<<InterfaceRealizationSubject-InterfaceSubject>>, <<ConcreteSubject-InterfaceRealizationSubject>>
**GENERATED_BY** create
**TYPES** InterfaceRealizationSubject
**FUNCTIONS**
create: → InterfaceRealizationSubject
**END-CLASS**

**CLASS** Notify
**IS-SUBTYPE-OF** Operation
**ASSOCIATES**
<<ClassSubject-Notify>>, <<InterfaceSubject-Notify>>
**GENERATED_BY** create
**TYPES** Notify
**FUNCTIONS**
create: ∗ → Notify
**AXIOMS** op: Notify
isQuery(op)
equal (visibility(op), ‘public’)
**END-CLASS**

**CLASS** Observer
**IS-SUBTYPE-OF** Classifier
**ASSOCIATES** <<AssocEndObserver-Observer>>
**GENERATED_BY** create
**DEFFERRED**
**TYPES** Observer
**FUNCTIONS**
create: ∗ → Observer
**END-CLASS**

**CLASS** ObserverSubject
**IS-SUBTYPE-OF** Association
**ASSOCIATES**
<<AssocEndConcreteSubject-ObserverSubject>>, <<AssocEndConcreteObserver-ObserverSubject>>
**GENERATED_BY** create
**TYPES** ObserverSubject
**FUNCTIONS**
create: ∗ → ObserverSubject
**AXIOMS** a: ObserverSubject; AP: Association-Property
size(get_memeberEnd(AP,a)) = 2
**END-CLASS**

**CLASS** SetState
**IS-SUBTYPE-OF** Operation
**GENERATED_BY** create
**TYPES** SetState
**FUNCTIONS**
create: ∗ → SetState
**AXIOMS** op:SetState; CO: Class-Operation; CI: Class-Interface; OP: Operation-Parameter
not isQuery(op)
notEmpty(get_ownedParameter(OP, op) and
size( select$_{par}$(get_ownedParameter(OP, op) , [equal(direction(par), ‘in’)] ) >= 1 )
equal(visibility(op), ‘public’)
**END-CLASS**

**CLASS** Subject
**IS-SUBTYPE-OF** Classifier
**ASSOCIATES**
<<AssocEndSubject-Subject>>
**GENERATED_BY** create
**DEFFERRED**
**TYPES** Subject
**FUNCTIONS**
create: ∗ → Subject
**END-CLASS**

**CLASS** SubjectObserver
**IS-SUBTYPE-OF** Association
**ASSOCIATES**
<<AssocEndSubject-SubjectObserver>>, <<AssocEndObserver-SubjectObserver>>
**GENERATED_BY** create
**TYPES** SubjectObserver
**FUNCTIONS**
create: ∗ → SubjectObserver
**AXIOMS** a: SubjectObserver ; AP: Association-Property
size(get_memberEnd(AP,a)) = 2
**END-CLASS**

**CLASS** Update
**IS-SUBTYPE-OF** Operation
**ASSOCIATES**

<<ClassObserver-Update>>, <<InterfaceObserver-Update>>
**GENERATED_BY** create
**TYPES** Update
**FUNCTIONS**
create: * → Update
**AXIOMS** op: Update
isQuery(op)
equal (visibility(op), 'public')
**END-CLASS**

***-- Especificación de las asociaciones***

**ASSOCIATION** AssocEndConcreteObserver-ConcreteObserver
**IS** Bidirectional-1 [AssocEndConcreteObserver: Class1; ConcreteObserver: Class2; assocEndConcreteObserver: role1; participant: role2; 1:mult1; 1:mult2; +:visibility1; +:visibility2]
**END**

**ASSOCIATION** AssocEndConcreteObserver-ObserverSubject
**IS** Bidirectional-1 [AssocEndConcreteObserver: Class1; ObserverSubject: Class2; assocEndConcreteObserver: role1; association: role2; 1:mult1; 1:mult2; +:visibility1; +:visibility2]
**CONSTRAINED_BY**
assocEndConcreteObserver: subsets memberEnd
association: redefines association
**END**

**ASSOCIATION** AssocEndConcreteSubject-ConcreteSubject
**IS** Bidirectional-1 [AssocEndConcreteSubject: Class1; ConcreteSubject: Class2; assocEndConcreteSubject:role1; participant:role2; 1:mult1; 1:mult2; +:visibility1; +:visibility2]
**END**

**ASSOCIATION** AssocEndConcreteSubject-ObserverSubject
**IS** Bidirectional-1 [AssocEndConcreteSubject: Class1; ObserverSubject: Class2; assocEndConcreteSubject:role1; association:role2; 1:mult1; 1:mult2; +:visibility1; +:visibility2]
**CONSTRAINED_BY**
assocEndConcreteSubject: subsets memberEnd
association: redefines association
**END**

**ASSOCIATION** AssocEndObserver-Observer
**IS** Bidirectional-1 [AssocEndObserver: Class1; Observer: Class2; assocEndObserver:role1; participant:role2; 0..1:mult1; 1:mult2; +:visibility1; +:visibility2]
**END**

**ASSOCIATION** AssocEndObserver-SubjectObserver
**IS** Bidirectional-1 [AssocEndObserver: Class1; SubjectObserver: Class2; assocEndObserver:role1; association:role2; 1:mult1; 1:mult2; +:visibility1; +:visibility2]
**CONSTRAINED_BY**
assocEndObserver: subsets memberEnd
association: redefines association
**END**

**ASSOCIATION** AssocEndSubject-Subject
**IS** Bidirectional-1 [AssocEndSubject: Class1; Subject: Class2; assocEndSubject:role1; participant:role2; 0..1:mult1; 1:mult2; +:visibility1; +:visibility2]
**END**

**ASSOCIATION** AssocEndSubject-SubjectObserver
**IS** Bidirectional-1 [AssocEndSubject: Class1; SubjectObserver: Class2; assocEndSubject:role1; association:role2; 1:mult1; 1:mult2; +:visibility1; +:visibility2]
**CONSTRAINED_BY**
assocEndSubject: subsets memberEnd
association: redefines association
**END**

**ASSOCIATION** Attach-ClassSubject
**IS** Composition-2 [ClassSubject: Whole; Attach: Part; classSubject: role1; attach: role2; 0..1:mult1; 1..*:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
classSubject: redefines Class
attach: subsets ownedOperation
**END**

**ASSOCIATION** Attach-InterfaceSubject
**IS** Composition-2 [InterfaceSubject: Whole; Attach: Part; interfaceSubject: role1; attach: role2; 0..1:mult1; 1..*:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
interfaceSubject: redefines Interface
attach: subsets ownedOperation
**END**

**ASSOCIATION** ClassObserver-GeneralizationObserver
**IS** Unidirectional-1 [GeneralizationObserver:Class1; ClassObserver:Class2; generalizationObserver:role1; classObserver:role2; 1:mult1; 1:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
classObserver: redefines general
**END**

**ASSOCIATION** ClassObserver-Update
**IS** Composition-2 [ClassObserver: Whole; Update: Part; classObserver: role1; update: role2; 0..1: mult1; 1..*:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
classObserver: redefines Class
update: subsets ownedOperation
**END**

**ASSOCIATION** ClassSubject-Detach
**IS** Composition-2 [ClassSubject: Whole; Detach: Part; classSubject: role1; detach: role2; 0..1:mult1; 1..*:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
classSubject: redefines Class
detach: subsets ownedOperation
**END**

**ASSOCIATION** ClassSubject-GeneralizationSubject
**IS** Unidirectional-1 [GeneralizationSubject:Class1; ClassSubject:Class2; generalizationSubject:role1; classSubject:role2; 1:mult1; 1:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
classSubject: redefines general
**END**

**ASSOCIATION** ClassSubject-Notify
**IS** Composition-2 [ClassSubject: Whole; Notify: Part; classSubject: role1; notify: role2; 0..1: mult1; 1..*:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
classSubject: redefines Class
notify: subsets ownedOperation
**END**

**ASSOCIATION** ConcreteObserver-GeneralizationObserver
**IS** Composition-1 [ConcreteObserver: Whole; GeneralizationObserver: Part; concreteObserver: role1; generalizationObserver:role2; 1:mult1; 0..1:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
generalizationObserver: subsets generalization
concreteObserver: redefines specific
**END**

**ASSOCIATION** ConcreteObserver -InterfaceRealizationObserver
**IS** Composition-1 [ConcreteObserver: Whole; InterfaceRealizationObserver: Part; concreteObserver: role1; interfaceRealizationObserver:role2; 1:mult1; 0..1:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
interfaceRealizationObserver: subsets interfaceRealization
concreteObserver: redefines implementingClassifier
**END**

**ASSOCIATION** ConcreteSubject-GeneralizationSubject
**IS** Composition-1 [ConcreteSubject: Whole; GeneralizationSubject: Part; concreteSubject: role1; generalizationSubject: role2; 1:mult1; 0..1:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
generalizationSubject: subsets generalization
concreteSubject: redefines specific
**END**

**ASSOCIATION** ConcreteSubject-GetState
**IS** Unidirectional-2 [ConcreteSubject:Class1; GetState:Class2; concreteSubject:role1; getState:role2; 1:mult1; 1..*:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
getState: subsets member
**END**

**ASSOCIATION** ConcreteSubject-InterfaceRealizationSubject
**IS** Composition-1 [ConcreteSubject: Whole; InterfaceRealizationSubject: Part; concreteSubject: role1; interfaceRealizationSubject:role2; 1:mult1; 0..1:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
interfaceRealizationSubject: subsets interfaceRealization

concreteSubject: redefines implementingClassifier
**END**

**ASSOCIATION** ConcreteSubject-Property
**IS** Unidirectional-2 [ConcreteSubject:Class1; Property:Class2; concreteSubject:role1; state:role2; 1:mult1; 1..*:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY** state: subsets member
**END**

**ASSOCIATION** ConcreteSubject-SetState
**IS** Unidirectional-2 [ConcreteSubject:Class1; SetState:Class2; concreteSubject:role1; setState:role2; 1:mult1; 1..*:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY** setState: subsets member
**END**

**ASSOCIATION** Detach-InterfaceSubject
**IS** Composition-2 [InterfaceSubject: Whole; Detach: Part; interfaceSubject: role1; detach: role2; 0..1:mult1; 1..*:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
interfaceSubject: redefines interface
detach: subsets ownedOperation
**END**

**ASSOCIATION** InterfaceObserver-Update
**IS** Composition-2 [InterfaceObserver: Whole; Update: Part; interfaceObserver: role1; update: role2; 0..1: mult1; 1..*:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
interfaceObserver: redefines interface
update: subsets ownedOperation
**END**

**ASSOCIATION** InterfaceObserver-InterfaceRealizationObserver
**IS** Unidirectional-1 [InterfaceRealizationObserver:Class1; InterfaceObserver:Class2; interfaceRealizationObserver:role1; interfaceObserver:role2; 1:mult1; 1:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY** interfaceObserver: redefines contract
**END**

**ASSOCIATION** InterfaceRealizationSubject-InterfaceSubject
**IS** Unidirectional-1 [InterfaceRealizationSubject:Class1; InterfaceSubject:Class2; interfaceRealizationSubject:role1; interfaceSubject:role2; 1:mult1; 1:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
interfaceSubject: redefines contract
**END**

**ASSOCIATION** InterfaceSubject-Notify
**IS** Composition-2 [InterfaceSubject: Whole; Notify: Part; interfaceSubject: role1; notify: role2; 0..1:mult1; 1..*:mult2; +:visibility1; +:visibility2]
**CONSTRAINED-BY**
interfaceSubject: redefines interface
notify: subsets ownedOperation
**END**

**END-PACKAGE**

## 5.3. Formalizando refinamientos

En el capítulo 4 se especificaron los refinamientos a través de la definición de transformación de modelos, la cual es una especificación de un mecanismo para convertir los elementos de un modelo, que es una instancia de un metamodelo particular, en elementos de otro modelo, que es instancia de otro metamodelo o posiblemente el mismo.

La especificación de los refinamientos de un componente puede ser vista como la especificación de un grafo dirigido acíclico **G = ( V , A )** (ver Figura 5.3), donde:

- **V** es el conjunto de vértices formado por los metamodelos que participan en los refinamientos. Están divididos en clases de equivalencias $V_{PIM}$, $V_{PSM}$ y $V_{ISM}$ tal que $V_{PIM} \cup V_{PSM} \cup V_{ISM} = V$
  - $V_{PIM}$: es el conjunto de los metamodelos a nivel PIM
  - $V_{PSM}$: es el conjunto de los metamodelos a nivel PSM
  - $V_{ISM}$: es el conjunto de los metamodelos a nivel de código
- **A** es un conjunto de arcos que representan refinamientos entre metamodelos. Cada arco es una relación binaria entre los elementos de V de manera tal que:
  - $\forall$ arco (m1, m2) $\in$ A: $(m1 \in V_{PIM} \wedge m2 \in V_{PSM}) \vee (m1 \in V_{PSM} \wedge m2 \in V_{ISM})$

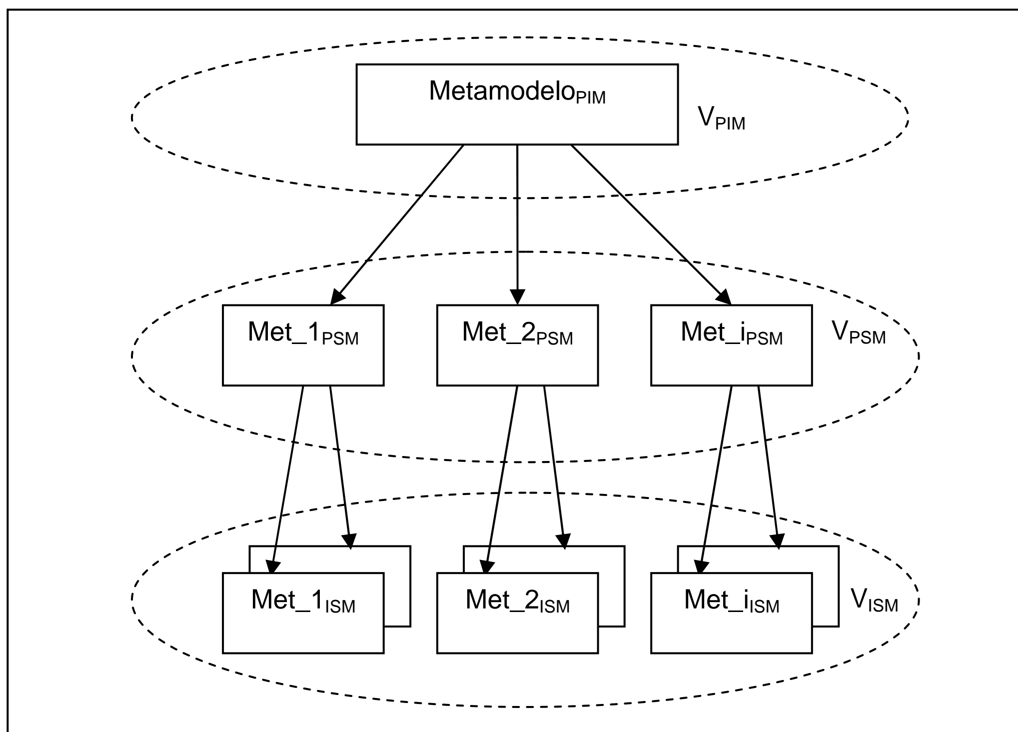



**Figura 5.3.** Formalizando de Refinamientos

Las instancias de dichos refinamientos se pueden traducir a especificaciones NEREUS por medio de la instanciación de esquemas reusables.

La Figura 5.4 muestra el esquema de un componente en NEREUS. La Figura 5.4.a muestra la signatura del esquema mientras de la Figura 5.4.b. los axiomas. Este esquema se instanciará con el nombre del componente. C-Metamodel es un metamodelo de componente, es decir, que permite la definición de modelos a nivel PIM, PSM o código.

**CLASS** *Component-name*

**IMPORTS** Set [C-Metamodel], Transformation

**GENERATED-BY** create, addLink, addMetamodel

**EFFECTIVE**

**TYPE** *Component-name*

**FUNCTIONS**

create: → *Component-name*

addLink: *Component-name* (c) *x* Transformation (t) → *Component-name*

**pre:** ( (get-source(t) ∈ $get_V_{PIM}$ (c) and get-target (t) ∈ $get_V_{PSM}$(c) ) or
(get-source(t) ∈ $get_V_{PSM}$ (c) and get-target (t) ∈ $get_V_{ISM}$(c) ) ) and
not existsLink (c, t)

addMetamodel: *Component-name* (c) x C-Metamodel (m) → *Component-name*

**pre:** not existsMetamodel (c, m)

removeLink: *Component-name* (c) *x* Transformation (t)→ *Component-name*

**pre:** existsLink (c, t)

removeMetamodel: *Component-name* (c) *x* C-Metamodel (m)→ *Component-name*

**pre:** existsMetamodel (c, m)

existsLink*: Component-name x* Transformation →Boolean

existsMetamodel*: Component-name x* C-Metamodel →Boolean

$get_V_{PIM}$: *Component-name* → Set [C-Metamodel]

$get_V_{PSM}$: *Component-name* → Set [C-Metamodel]

$get_V_{ISM}$: *Component-name* → Set [C-Metamodel]

**Figura 5.4.a.** Un esquema de Componente- Signatura

```
AXIOMS  c: Component-name; t, t1, t2: Transformation; m,m1,m2: C-Metamodel;
        s1,s2,s3: Set [C-Metamodel]

removeLink (addLink (c,t1),t2)= IF equal (t1,t2) THEN c
                                ELSE addLink(removeLink(c,t2),t1) ENDIF
removeLink (addMetamodel (c,m),t) = addMetamodel(removeLink (c,t), m)

removeMetamodel(addLink(c,t), m)= IF (get_source(t)= m  or get_target(t)= m
                                THEN removeMetamodel(c,m)
                                ELSE addLink(removeMetamodel(c,m), t) ENDIF
removeMetamodel(addMetamodel(c,m1),m2)= IF m1=m2 THEN c
                                ELSE addMetamodel(removeMetamodel(c,m2),m1) ENDIF

existsLink (create(), t) = false
existsLink (addLink (c, t1), t2) = IF equal(t1,t2) THEN true ELSE existsLink (c, t2) ENDIF
existsLink (addMetamodel (c,m),t) = existsLink (c, t)

existsMetamodel(create(), m)=  false
existsMetamodel(addLink(c, l),m)= existsMetamodel(c,m)
existsMetamodel(addMetamodel(c, m1),m2)= IF m1=m2 THEN true
                                ELSE existsMetamodel(c,m2) ENDIF

get_V_PIM (create()) = createSet()
get_V_PIM(addLink(c,t))= get_V_PIM(c)
get_V_PIM(addMetamodel(c,m))= IF equal( type(m),'PIM') THEN add (get_V_PIM(c),m)
                                THEN get_V_PIM(c) ENDIF

get_V_PSM (create()) = createSet()
get_V_PSM(addLink(c,t))= get_V_PSM(c)
get_V_PSM(addMetamodel(c,m))= IF equal(type(m), 'PSM') THEN add (get_V_PSM(c),m)
                                THEN get_V_PSM(c) ENDIF

get_V_ISM (create()) = createSet()
get_V_ISM(addLink(c,t))= get_V_ISM(c)
get_V_ISM(addMetamodel(c,m))= IF equal(type(m), 'ISM') THEN add(get_V_ISM(c),m)
                                THEN get_V_ISM(c) ENDIF
END-CLASS
```

**Figura 5.4.b.** Un esquema de Componente- Axiomas

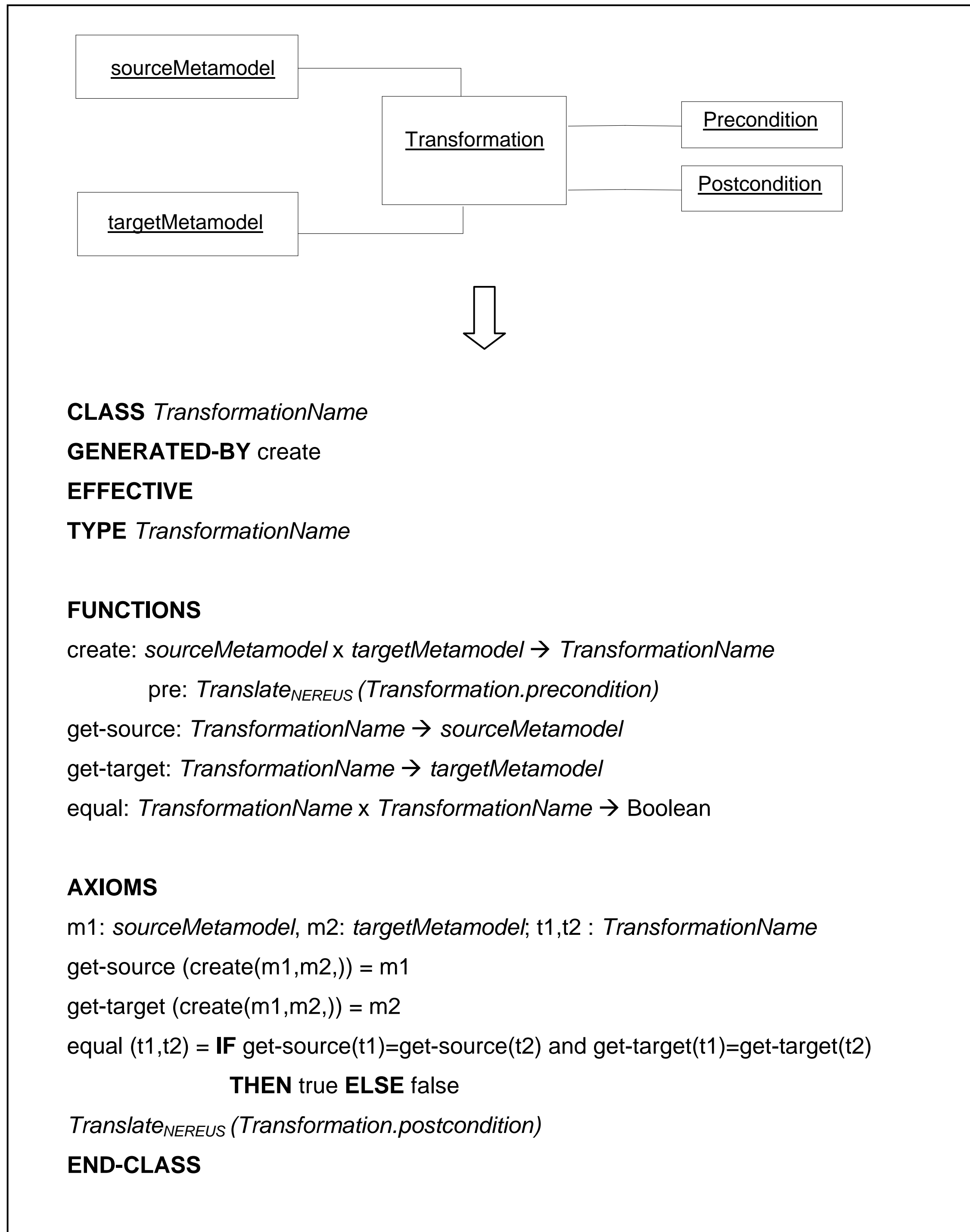



**Figura 5.5.** Formalizando Refinamientos. Un esquema de transformación

La Figura 5.5 muestra el esquema para la clase Transformation en NEREUS usada por el esquema de la Figura 5.4. Cada transformación es instanciada con un nombre, los metamodelos origen y destino junto con las precondiciones y postcondiciones.

La función $Translate_{NEREUS}$ *(transformation.precondition)* que aparece en el esquema de la transformación después la palabra “pre:” de la operación *create*, traduce a NEREUS la precondición de la transformación. La función $Translate_{NEREUS}$ *(transformation.postcondition)* que aparece en los axiomas traduce a NEREUS las postcondiciones de la transformación.

Una instanciación del esquema de transformación para la transformación PIMtoPSMEiffel entre un metamodelo del patrón Observer a nivel PIM y un metamodelo del mismo patrón a nivel PSM-EIFFEL descripta en el capítulo 4 es la siguiente:

[*TransformationName*: PIM-UML to PSM-EIFFEL;
*sourceMetamodel*: Observer-PIM-Metamodel;
*targetMetamodel*: Observer-PSM-EIFFEL-Metamodel;
*precondition*: OCLexp1;
*postcondition*: OCLexp2 ]

donde:

- *TransformationName* es instanciado con el nombre de la transformación PIM-UML to PSM-EIFFEL,
- *sourceMetamodel* es instanciado con el nombre del metamodelo origen Observer-PIM-Metamodel,
- *targetMetamodel* es instanciado con el nombre del metamodelo destino Observer-PSM-EIFFEL-Metamodel y
- *precondition* y *postcondition* son las expresiones OCL que siguen a las palabras 'pre:' y 'post:' de dicha transformación.
- Si la definición de transformación tiene operaciones adicionales, la signatura de éstas pasa a formar parte de la signatura de la transformación en NEREUS como operaciones privadas de la clase y el cuerpo de estas operaciones se traduce a axiomas de la clase.

La Figura 5.6 muestra parcialmente el resultado de la formalización de dicha transformación.

```
CLASS PIM-UML_to_PSM-EIFFEL
GENERATED-BY create
EFFECTIVE
TYPE PIM-UML_to_PSM-EIFFEL
FUNCTIONS
create:Observer-PIM-Metamodel x Observer-PSM-EIFFEL-Metamodel→PIM-UML_to_PSM-EIFFEL
get-source: PIM-UML_to_PSM-EIFFEL → Observer-PIM-Metamodel
get-target: PIM-UML_to_PSM-EIFFEL → Observer-PSM-EIFFEL-Metamodel
equal: PIM-UML_to_PSM-EIFFEL x PIM-UML_to_PSM-EIFFEL → Boolean
-- operaciones locales (privadas)
- classClassMatch: Observer-PSM-EIFFEL-Metamodel::EiffelClass x
                                        Observer-PIM-Metamodel::Class → Boolean
- classInterfaceMatch: Observer-PSM-EIFFEL-Metamodel::EiffelClass x
                                        Observer-PIM-Metamodel::Interface → Boolean
…
AXIOMS
  m1: Observer-PIM-Metamodel,   m2: Observer-PSM-EIFFEL-Metamodel;
  t1,t2 : PIM-UML_to_PSM-EIFFEL; PP : Package-PackageableElement;
  e: Observer-PSM-EIFFEL-Metamodel::EiffelClass; c: Observer-PIM-Metamodel::Class; …

get-source (create(m1,m2)) = m1
get-target (create(m1,m2)) = m2
equal (t1,t2) = IF get-source(t1)=get-source(t2) and get-target(t1)=get-target(t2)
          THEN true ELSE false
-- Translate_NEREUS (Transformation.postcondition)
-- sourceModel y targetModel tienen el mismo número de clasificadores.
size (select_elem( get_ownedMember(PP,m2), [oclIsTypeOf(elem,EiffelClass)] ))  =
    size ( select_elem( get_ownedMember(PP,m1 ), [oclIsTypeOf(elem, Class)] )) +
    size ( select_elem( get_ownedMember(PP,m1 ), [oclIsTypeOf(elem, Interface)] ) ) and

-- Para cada interface 'sourceInterface' en sourceModel existe una clase 'targetClass'
-- en targetModel tal que:
forAll_sourceInterface( select_elem( get_ownedMember(PP,m1) , [oclIsTypeOf(elem, Interface)] ),
  [exists_targetClass (select_elem( get_ownedMember(PP,m2) , [oclIsTypeOf(elem, EiffelClass)] ),
    -- sourceInterface y targetClass se corresponden
    [ interfaceClassMatch(oclAsType(targetClass, EiffelClass),
                          oclAsType(sourceInterface, Interface))  ]  ) ]  ) and

-- Para cada clase 'sourceClass' en sourceModel existe una clase 'targetClass' en targetModel
-- tal que:
forAll_sourceClass( select_elem( get_ownedMember(PP,m1) , [oclIsTypeOf(elem, Class)] ),
  [exists_targetClass (select_elem( get_ownedMember(PP,m2) , [oclIsTypeOf(elem, EiffelClass)] ),
    -- sourceClass y targetClass se corresponden
    [classClassMatch (oclAsType(targetClass,EiffelClass),oclAsType(sourceClass, Class))] )])

-- operaciones locales
classClassMatch(e,c) = equal(name(e), name(c)) and equal (isDeferred(e),isAbstract(c)) and …
END-CLASS
```

**Figura 5.6.** Una instanciación del esquema de transformación

# 6. Conclusiones

MDA es un framework de desarrollo de software que sistematiza la construcción de software a partir de modelos del sistema en distintos niveles de abstracción para guiar todo el proceso de desarrollo. El éxito de esta propuesta depende no sólo de la definición de transformaciones entre modelos, sino además de librerías de componentes que tengan un impacto significativo sobre las herramientas que provean un soporte para la misma.

En esta tesis se presentó un "megamodelo" para definir familias de componentes para patrones de diseño a través de una técnica de metamodelado que permite alcanzar un alto nivel de reusabilidad y adaptabilidad en una perspectiva de MDA.

La definición de los componentes reusables se hizo a través de la especificación de metamodelos para patrones de diseño en tres niveles de abstracción (PIM, PSM e ISM) y de la especificación de transformaciones de modelo a modelo (refinamientos) basadas en dichos metamodelos.

Por otra parte, el desarrollo de componentes reusables requiere poner énfasis sobre la calidad del software, por lo cual las técnicas tradicionales para la verificación y validación son esenciales para lograr atributos de calidad en el software. Una especificación formal clarifica el significado deseado de los metamodelos y de las transformaciones de modelos basadas en metamodelos ayudando a validarlos y proveyendo una referencia para la implementación. En esta dirección, se propuso formalizar los componentes usando la notación de metamodelado NEREUS a través de la formalización de los metamodelos MOF y de las transformaciones de modelos basadas en metamodelos.

La propuesta fue ilustrada usando el patrón de diseño *Observer*.

## 6.1. Contribuciones

A continuación se mencionan las principales contribuciones de este trabajo.

- *Un megamodelo especificado como una unidad de encapsulamiento*. Define familias de componentes que son instancias del megamodelo. Un megamodelo encapsula metamodelos PIM, PSM e ISM y refinamientos basados en metamodelos.
- *Metamodelos de patrones de diseño especificados en tres niveles de abstracción*. Los metamodelos de los patrones fueron especificados:
    - independientes de una plataforma en particular,
    - dependientes de una plataforma y
    - dependientes de un lenguaje de programación en particular.
- *Especialización del metamodelo UML*. Para la construcción de los metamodelos de los patrones de diseño en los distintos niveles fue necesaria la especialización del metamodelo UML con la creación de nuevas metaclases, asociaciones y restricciones OCL.

- *Una técnica de metamodelado para la descripción de componentes MDA*. La técnica propuesta integra especificaciones basadas en metamodelos MOF y especificaciones formales. Si bien se especificaron componentes para patrones de diseño estándar, la propuesta no está limitada a ellos.
- *Una formalización de Componentes en NEREUS*. Se propuso la utilización de la notación de metamodelado NEREUS junto con un sistema de reglas de transformación que permiten traducir metamodelos MOF a NEREUS. De esta manera se independiza a los desarrolladores de las especificaciones formales, sólo necesitan manipular con los modelos que ellos han creado.
- *Chequeo de consistencia riguroso*. Por medio de los metamodelos, que son una especificación de los patrones de diseño, puede hacerse un chequeo para determinar si un modelo de un patrón dado concuerda con su especificación.
- *Una propuesta de formalización que permite la Interoperabilidad de lenguajes formales.* El lenguaje NEREUS utilizado para la formalización de los componentes, es un lenguaje que puede ser visto como una notación intermedia abierta a muchos otros lenguajes formales.

## 6.2. Futuros trabajos

Los patrones de diseño pueden ser modelados utilizando distintas vistas, estructura estática, interacciones y conducta basada en estados. Los metamodelos que forman parte de los componentes fueron construidos para especificar la vista estructural de los patrones de diseño, una de las metas es completar el catálogo de componentes para patrones de diseño de manera tal que reflejen el comportamiento, es decir especificar metamodelos para los diagramas de interacción.

Un problema crucial es cómo hacer para detectar qué parte de un diagrama matchea con un patrón. El metamodelado puede ayudar en la identificación de patrones de diseño por medio de un matching de signaturas y semántico. Completar dicho matching es otra de las metas propuestas.

Por último, una de las principales metas es la integración de la propuesta con herramientas CASE UML de manera tal de completar a través de la implementación todo el trabajo de investigación.

# Bibliografía


Albin-Amiot H. y Guéhéneuc Y. (2001a). Design Pattern Application: Pure-Generative Approach vs. Conservative-Generative Approach. OOPSLA Workshop on Generative Programming. USA.

Albin-Amiot H. y Guéhéneuc Y. (2001b). Meta-modeling Design Patterns: application to pattern detection and code synthesis. ECOOP Workshop on Automating Object-Oriented Software Development Methods, Budapest, Hungary.

Albin-Amiot, H., Coint, P. , Guéhéneuc, Y. y Jussien, N. (2001). Instantianting and Detecting Design Patterns : Putting Bits and Pieces Together. IEEE 2001. Páginas 166-173.

Alpert, S., Brown, K. y Woolf, B. (1998). The design Patterns Smalltalk Companion. Addison Wesley.

Arnout, K. (2004). From Patterns to Components. Tesis Doctoral, Swiss Institute of Technology (ETH Zurich).

Beck D., Coplien J., Crocker, R., Dominick, L., Meszaros, G. y Paulisch, F. (1996). Industrial experience with design patterns. ICSE-18 (International Conference on Software Engineering), Technical University of Berlin, Germany. Páginas 103-113.

Bettin, J. (2003). Practicalities of Implementing Component-Based Development and Model-Driven Architecture. Proceedings de Workshop Process Engineering for Object-Oriented and Component-Based Development, OOSPLA 2003, USA.

Bézivin, J, Gérard, S, Muller P. y Rioux L. (2003). "MDA Components: Challenges and Opportunities". Proceeding de Metamodelling for MDA. York, Inglaterra.

Bézivin, J. , Jouault, F. y Valduriez, P. (2004). On the Need for Megamodels. Proceedings de Best Practices for Model-Driven Software Development (MDSD 2004). OOSPLA 2004 Workshop.

Bidoit, M., Hennicker, R., Tort, F. y Wirsing M. (1999). Correct Realizations of Interface Constraints with OCL. 2nd International Conference UML'99, The Unified Modeling Language-Beyond the Standard. LNCS 1723. Springer-Verlag. Páginas 399-415.

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.

Bollati, V. , Vara, J. , Vela, B. y Marcos, E. ( 2007). Una revisión de herramientas MDA. Actas del IV Taller sobre Desarrollo de Software Dirigido por Modelos MDA y Aplicaciones (DSDM'07). España. Páginas 91-100.

Bruel, J., Henderson-Sellers, B., Barbier, F., Le Parc, A. y France, R.. (2001). Improving the UML Metamodel to Rigorously Specify Aggregation and Composition. 7th International Conference on Object-Oriented Information Systems (OOIS'01). Calgary, CA, Springer-Verlag. Páginas. 5-14.

Budinsky, F., Finni, M., Vlissides, J. y Yu, P. (1996). Automatic code generation from design patterns. IBM System Journal, Vol. 35, N° 2.

Bulka, A. (2002). Design Pattern Automation. Third Asia-Pacific Conference on Pattern Languages of Programs (KoalaPLoP 2002), Melbourne, Australia.

Buschmann, F, Meunier, R., Rohnert, H., Sommerland, P. y Stal , M. (1996). Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley.

Cariou, E. , Marvie, R., Seinturier, L. y Duchien L. (2004). OCL for the Specification of Model Transformation Contracts. Proceedings de Workshop OCL and Model Driven Engineering, Lisboa, Portugual.

CASE TOOLS (2007). Disponible en: www.objectsbydesign.com/tools/umltools_byCompany.html

Cengarle, M. y Knapp A. (2001). A Formal Semantics for OCL 1.4. <<UML>> 2001, Modeling Languages, Concepts and Tools. (M. Gogolla; C. Kobryn eds.) LNCS 2185. Springer-Verlag.

Coplien, J. (1992). Advanced C++ Programming Styles and Idioms. Reading: Addison-Wesley.

Czarnecki, K., Helsen, S. (2003). Classification of Model Transformation Approaches. En: Bettin J.y otros (eds). Proceedings de OOSPLA’03 Workshop on Generative Techniques in the Context of Model-Driven Architecture. USA.

CWM (2003). Common Warehouse Metamodel. Versión 1.1.Documento: formal/2003-03-02.

Debnath, N., Garis, A, Riesco, D. y Montejano, G. (2006). Defining Patterns Using UML Profiles. IEEE International Conference on Computer System and Application. ISBN: 1-4244-0211-5. Páginas 1147-1150.

D´Souza, D. y Cameron Wills, A. (1998) Objects, Components and Framework with UML. The catalysis Approuch. Addison-Wesley.

Eden, A., Yehudai, A. y Gil, J. (1997). Precise specification and automatic application of design patterns. 1997 International Conference on Automated Software Engineering (ASE’ 97), Lake Tahoe, Canada. Páginas 143-152.

Elaasar, M., Briand L. y Labiche, Y. (2006). A Metamodeling Approach to Pattern Specification and Detection. Proceedings de ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006). LNCS 4199. Páginas 484-498.

Emerson, M, y Sztipanovits, J. (2004). Implementing a MOF-Based Metamodeling Environment Using Graph Transformations. 4th OOPSLA Workshop on Domain-Specific Modeling. Vancouver, Canadá. 24 de Octubre. Páginas 83-92.

Favre, L. (2001). A Formal Mapping between UML Static Models and Algebraic Specifications. Lecture Notes in Informatics (p. 7) SEW Practical UML-Based Rigorous Development Methods- Countering or Integrating the eXtremists (Eds. Evans, R. France, A. Moreira, B. Rumpe) GI Edition, Konner Kollen-Verlag. Páginas 113-127.

Favre, L. (2007). El Lenguaje NEREUS. Reporte interno. Grupo de Tecnología de Software, INTIA. Universidad Nacional del Centro de la Provincia de Buenos Aires, Argentina.

Favre, L. (2005). Foundations for MDA-based Forward Engineering. Journal of Object Technology (JOT). Vol. 4, Nº 1. 129-153.

Favre, L. (2006). "A Rigorous Framework for Model Driven Development". Keng Siau (ed.). Advanced Topics in Database Research, Vol. 5. Chapter I, IGP, USA, 1-27, 2006

Favre, L. y Martinez, L. (2006). Formalizing MDA Components. Internacional Conference on Software Reuse (ICRS 2006). Torino. Italia. LNCS 4039, Springer Verlag Berlin Heidelberg, ISNN 0302-9743. Páginas 326-339.

Favre, L., Matinez, L. y Pereira, C. (2000). Transforming UML Static Models to Object Oriented Code. Proceedings de 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. y Martinez, L., Pereira C. (2001). Una integración de modelos estáticos UML y Eiffel. Publicado en: Proceedings del VII Congreso Argentino de Ciencias de la Computación (CACIC 2001). Calafate, Santa Cruz, Argentina. Páginas 521-530.

Favre, L., Martinez, L. y Pereira, C. (2002). Forward Engineering and UML: From UML Static Models to Eiffel Code. Publicado en: Proceedings de 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. ISBN 1-931777-44-6. USA. Páginas 199-217.

Favre, L., Martinez, L. y Pereira, C. (2004) Integrating Design Patterns into Forward Engineering Processes. Proceedings de 2004 Information Resources Management Association International Conference (IRMA 2004). New Orleans, USA. ISBN 1-59140-261-1. Páginas 502-505.

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-V Mehdi Khosrow-Pour (Editor). IGP (Idea Group Publishing) .USA. ISBN 1-59140-553-X. Páginas 1212-1217.

Favre, L., Martinez, L. y Pereira, C. (2008). Foundations for MDA Case Tools. Artículo aceptado para la Encyclopedia of Information Science and Technology, Second Edition, IGI Global, USA. A ser publicado en 2008.

France, R., Kim, D., Ghosh, S. y Song, Eunjee. (2004). A UML-Based Pattern Specification Technique. IEEE Transactions on Software Engineering. Vol. 30, Nº3, Marzo, 2004. IEEE Computer Society. Pag. 193-206.

Florijn, G., Meijers, M. y van Winsen, P. (1997) Tool support for object-oriented patterns. ECOOP (European Conference on Object Oriented Programming) '97, Jyväskylä, Finland. Páginas 472-795.

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

García Molina, J., Rodríguez, J., Menárguez, M., Ortín, M. y Sánchez, J. (2004). Un estudio comparativo de dos herramientas MDA: OptimalJ y ArcStyler. I Taller sobre Desarrollo de Software Dirigido por Modelos MDA y Aplicaciones (DSDM'04). España. Páginas 88-99.

Giandini, R. y Pons, C. (2006). Un lenguaje para Transformación de Modelos basado en MOF y OCL. Proceeding de 32th Latin-American Conference on Informatics (CLEI, 2006). Santiago, Chile.

Giandini, R., Pérez, G. y Pons, C. (2007) A Minimal OCL-based Profile for Model Transformation. Publicado en las actas de las VI Jornadas Iberoamericanas de Ingenieria de software e Ingeniería del Conocimiento (JIISIC´07) ISBN: 978-9972-2885-2-4. Lima, Perú.

Grand, M. (1998). Patterns in Java. A Catalog of Reusable Desing Patterns Illustrated with UML. Wiley Computer Publishing.

Hamie, A., Civello, F., Howse, J., Kent, S. y Mitchell R. (1998). Reflections on the Object Constraint Language. Proceeding de UML'98 International Workshop, Mulhouse.

Hussmann, M., Cerioli, Reggio, G. y Tort, F. (1999) Abstract Data Types and UML Models. Report DISI-TR-99-15, University of Genova, France, ESSAIM. Páginas 137-145.

Judson, S., Carver D. y France, R. (2003). A metamodeling approach to model transformation. OOPSLA Companion 2003. California, USA. Páginas 326-327.

Kim, D, France, R., Ghosh, S. y Song, E. (2003a). A UML-Based Metamodeling Language to Specifying Desing Patterns. Wisme@UML'2003-UMLWorkshop W2 (Workshop in Software Model Engineering). San Francisco, USA.

Kim, D., France, R., Ghosh, S. y Song, E. (2003b). A Role-Based Metamodeling Approach to Specifying Desing Patterns. Proceedings de the 27th Annual International Computer Software and Applications Conference (COMPSAC'03). IEEE Computer Society, 2003.

Kim, S. y Carrington D. (1999). Formalizing the UML Class Diagram using OBJECT-Z. Proceedings de UML 99. LNCS 1723, Springer-Verlag. Páginas 83-98.

Kleppe, A., Warmer, J. y Bast W. (2003). MDA Explained: The Model Driven Architecture™: Practice and Promise. Addison Wesley.

Kuster, J., Sendall S. y Wahler M. (2004). Comparing Two Model Transformation Approaches. En: Bezivin, J. et. al (eds.). Proceedings de OCL and Model Driven Engineering Workshop. Lisboa, Portugal.

Mandel, L. y Cengarle, M. V. (1999). On the Expressive Power of the Object Constraint Language OCL. FM'99- Formal Methods. Volumen. I, LNCS 1708. Springer-Verlag Berlin Heidelberg. Páginas 854-874.

Martinez, L. y Favre, L. (2004). Una Integración de Patrones de Diseño en Procesos de Ingeniería Forward de Modelos Estáticos UML. Publicado en: VI Workshop de Investigadores en Ciencias de la Computación (WICC 2004). Neuquén, Argentina. Páginas 249-253.

Martinez, L. y Favre, L. (2006). MDA-Based Design Pattern Components. Publicado en: Proceedings de 2006 Information Resources Management Association International Conferences (IRMA 2006). Washington, D.C. USA. ISBN 1-59904-019-0. Páginas 259-263.

MDA (2003). MDA Guide. Version 1.0.1. Documento: omg/2003-06-01. Disponible en www.omg.org/mda

MDA (2007). Model-Driven Arquitecture. Disponible en www.omg.org/mda

Meyer, B. (1997a). Object-Oriented Software Construction. Prentice Hall PTR.

Meyer, B. (1997b). The Next Software Breakthrough. Computer, Vol. 30, Nº 7. Pág. 113-114.

Meyer, B. (2003). The Grand Challenge of Trusted Components. 25th International Conference on Software Engineering (ICSE), Portland, Oregon. IEEE Computer Press. Páginas 660-667.

ModelMaker (2007). Disponible en www.modelmakertools.com

MOF (2006). Meta Object Facility. Documento: formal/2006-01-01. Disponible en www.omg.org/mof

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.

Mottu, J-M., Baudry, B. y Le Traon, Y. (2006) “Reusable MDA Components: A Testing-for-Trust Approach”. Proceedings de ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2006). Génova, Italia. LNCS 4199. Páginas 589-603.

OCL (2006). Object Constraint Language. Versión 2.0. Documento : formal/06-05-01. Disponible en www.omg.org

Objecteering/UML (2007). Disponible en: www.objecteering.com/products.php

Opdahl, A., Henderson-Sellers, B. y Barbier F. (2001) Ontological Analysis of Whole-Part Relationship in OO-Models. Information and Software Technology, Volumen 43. Elsevier Science. Páginas. 387-399.

QVT (2007). MOF Query/View/Transformation Specification. Versión 1.0 Beta 2. ptc/07-07-07. Disponible en: www.omg.org

Padawitz, P. (2000). Swinging UML: How to Make Class Diagrams and State Machines Amenable to Constraint Solving and Proving. Proceedings de <<UML>> 2000 The Unified Modeling Language. Advancing the Standard, Third International Conference (A.Evans, S. Kent eds.). LNCS 1939. Springer-Verlag. Páginas 162-177.

Poernomo, I.. (2006) "The meta-Object facility Typed". Proceedings del 2006 ACM Symposium on Applied. Computing (SAC), Dijon, France. Páginas 1845-1849.

Risi, W. y Rossi, G. (2004). An architectural pattern catalog for mobile web information system. International Journal of Mobile Communication (IJMC). Vol. 2. Páginas 235-247.

Richters, M. y Gogolla, M. (1998). On Formalizing the UML Object Constraint Language OCL. Proceedings de International Conference on Conceptual Modeling (ER'98), Singapore, Noviembre 16-19, LNCS 1507. Springer-Verlag. Páginas 449-464.

Shaw, M. y Garlan, D. (1996). Software Architecture - Perspectives on an Emerging discipline. Uper Saddle River: Prentice-Hall.

Shroff, M y France, R. (1997). Towards a Formalization of UML Class Structures. Twenty-First Annual International Computer Software and Applications Conference (COMPSAC 97). IEEE Computer Society. Páginas 646-651.

Snook, C. y Butler, M. (2000). Tool-Supported Use of UML for Constructing B Specifications. Technical Report, Department of Electronics and Computer Science, University of Southampton, United Kingdom.

Stevens, P. (2001) On Associations in the Unified Modeling Language. Proceedings <<UML>> 2001-Modeling Languages, Concepts and Tools. LNCS 2185 (M. Gogolla y C. Kobryn eds.). Springer-Verlag. Página 361.

Szyperski, C., Gruntz, D. y Murer, S. (2002). Component Software. Beyond Object-Oriented Programming, Second Edition. Addison-Wesley and ACM Press.

UML (2007). Unified Modeling Language Specification. Versión 2.1.1. Disponible en www.omg.org

UML-Infrastructure (2007). Unified Modeling Language: Infrastructure. Versión 2.1.1. Documento: formal/07-02-06. Disponible en www.omg.org

UML-Superstructure(2007). Unified Modeling Language: Superstructure Versión 2.1.1. Documento: formal/07-02-05. Disponible en www.omg.org

UMLStudio (2007). Disponible en www.pragsoft.com

# Anexo A: Metamodelo UML

## A.1. Diagramas principales del paquete *Kernel* del metamodelo UML

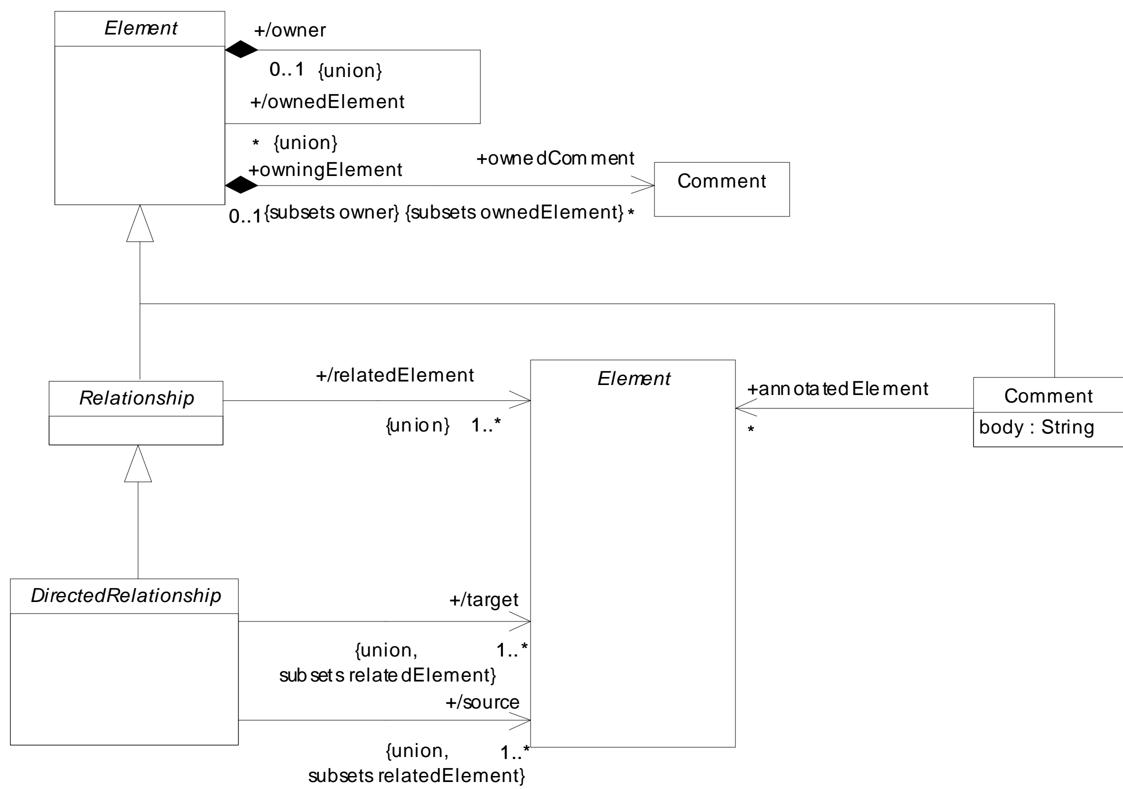
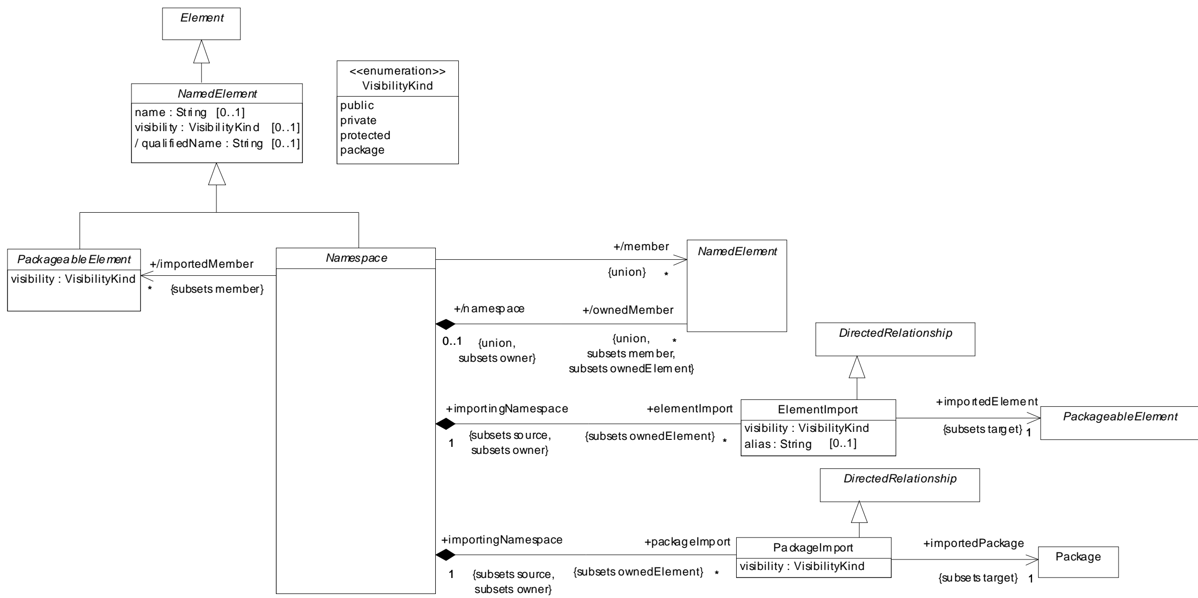
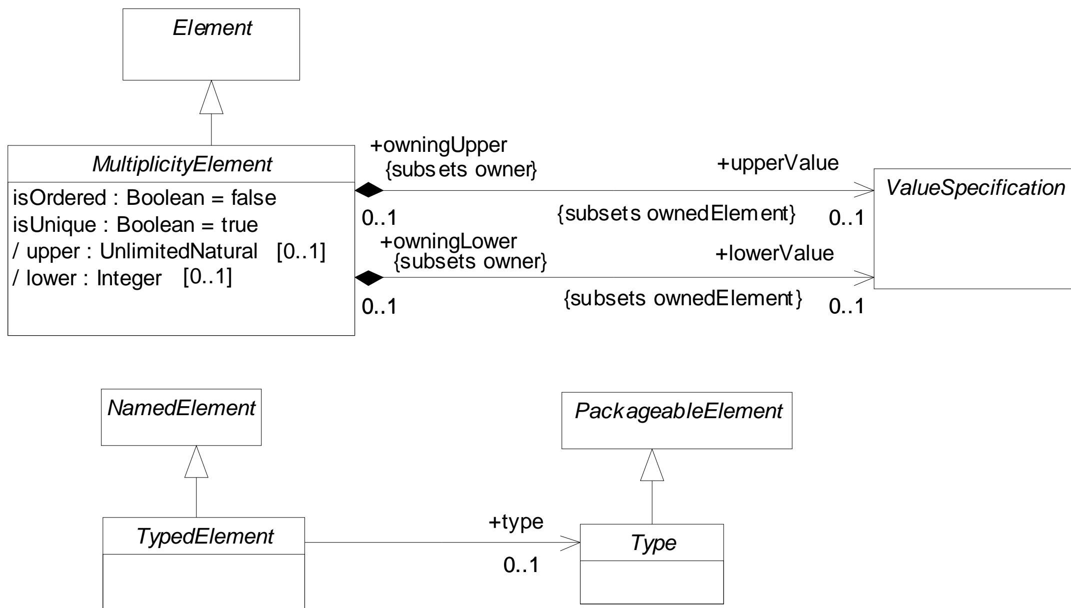


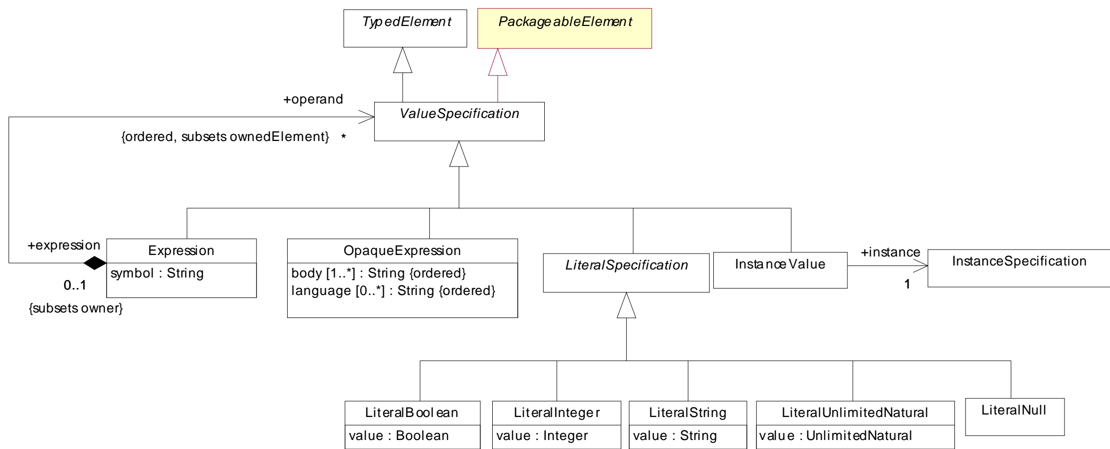

**Diagrama de clases *Root***



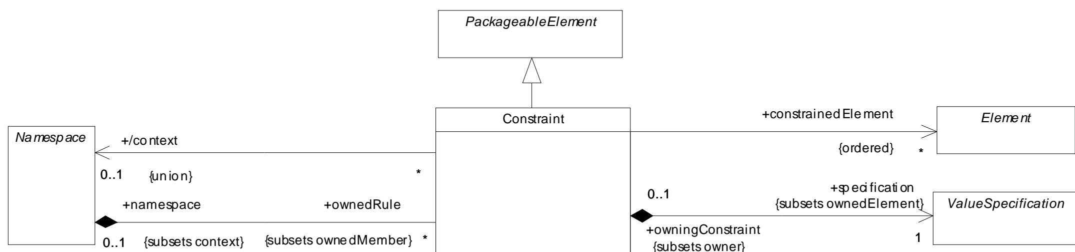

## Diagrama de clases de *Namespaces*



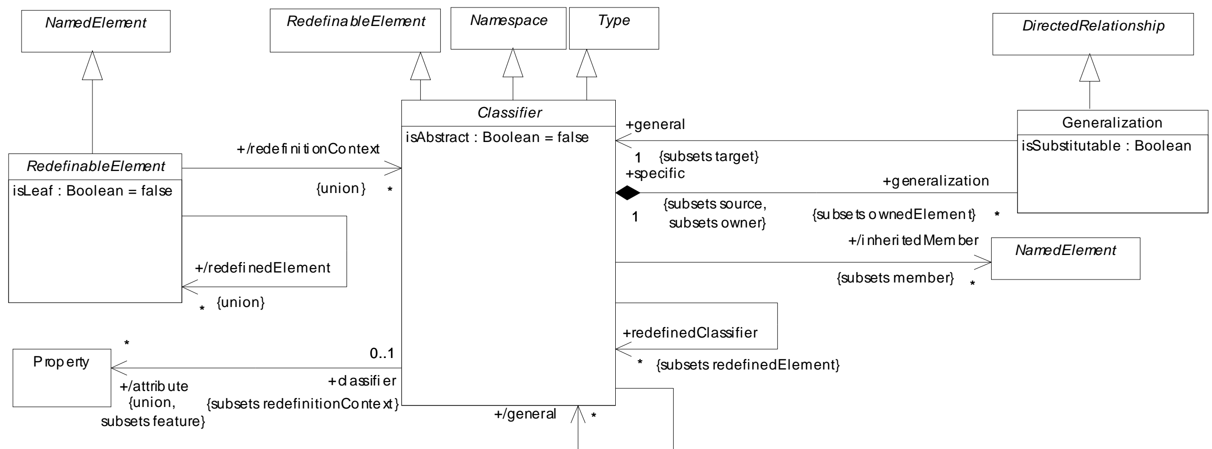

## Diagrama de clases de *Multiplicities*



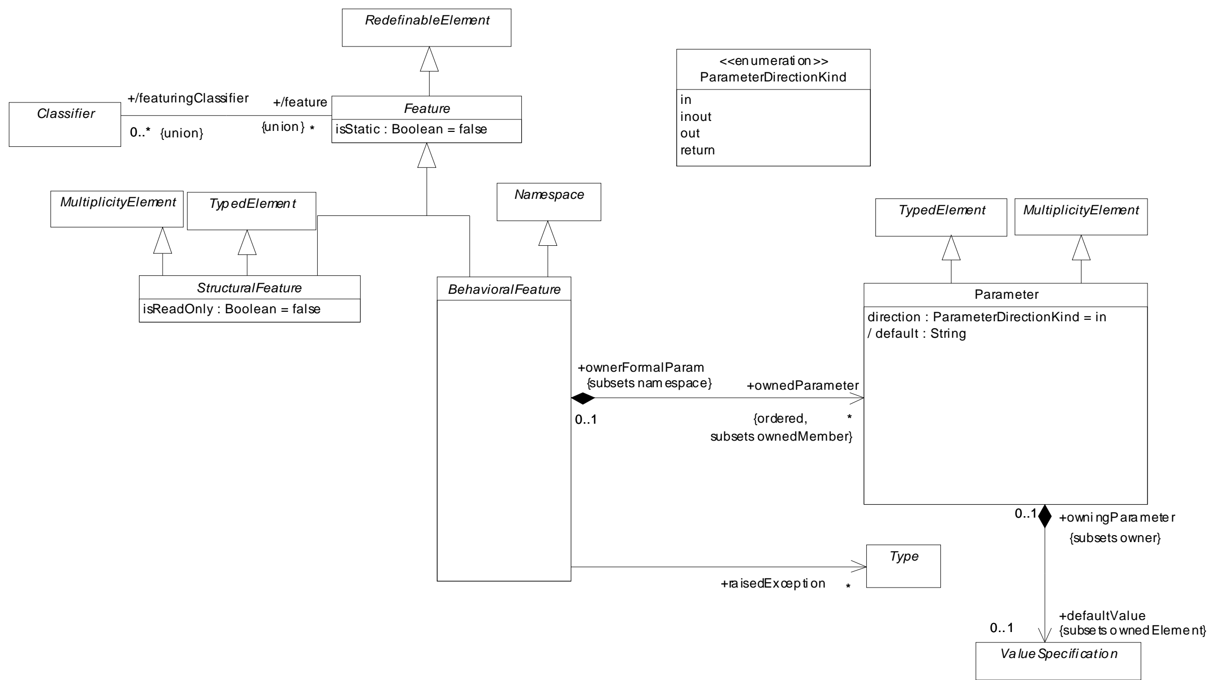

## Diagrama de clases de *Expressions*



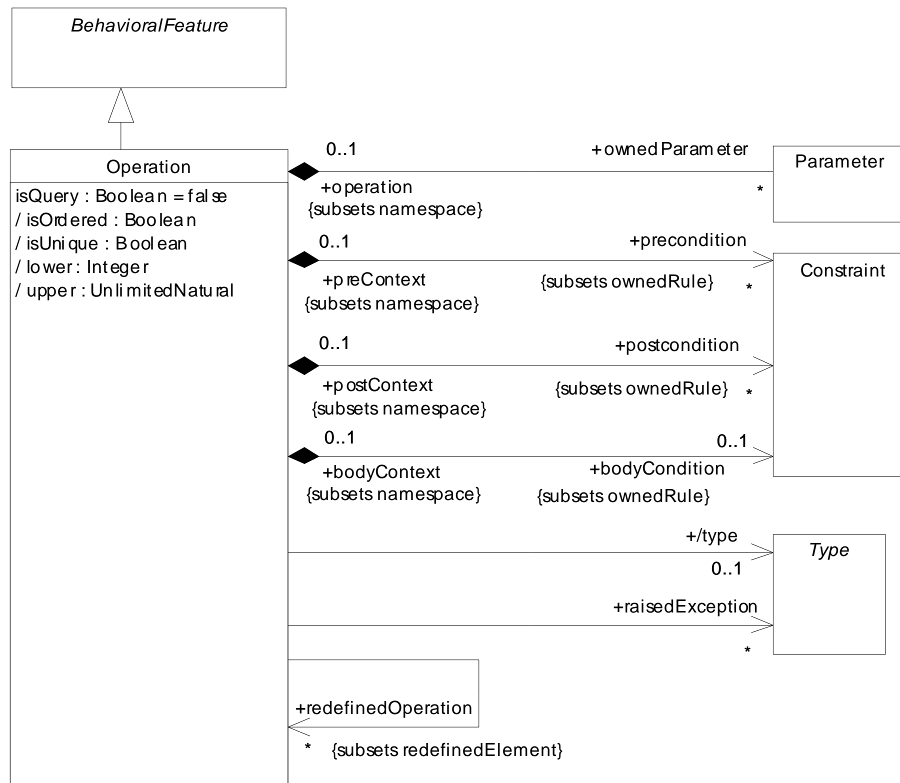

## Diagrama de clases de *Constraints*



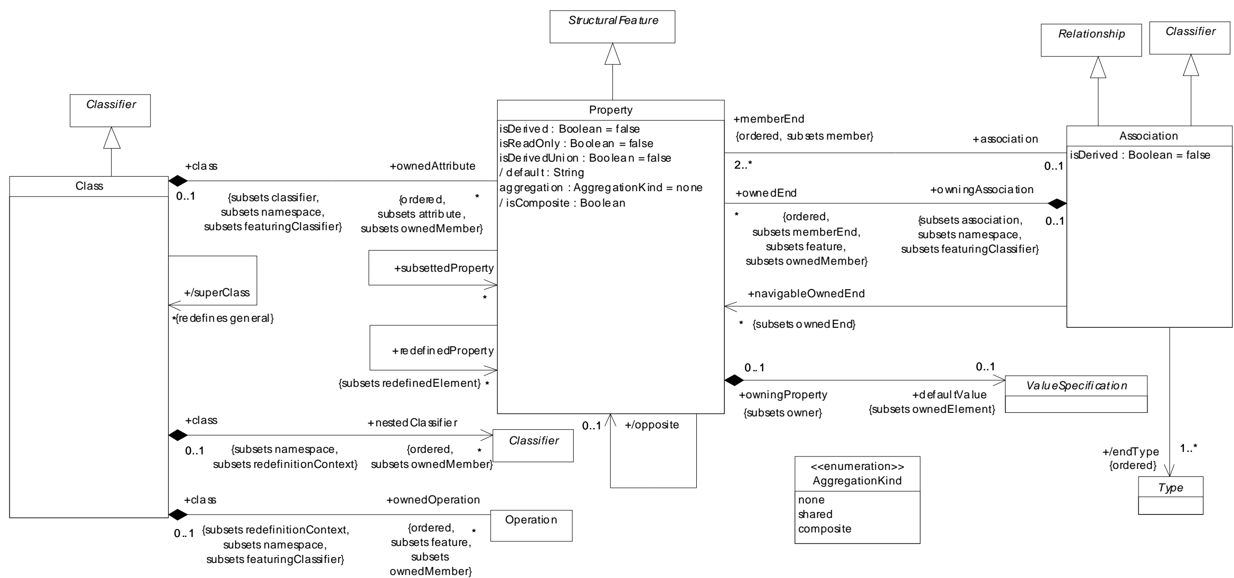

## Diagrama de clases de *Classifiers*



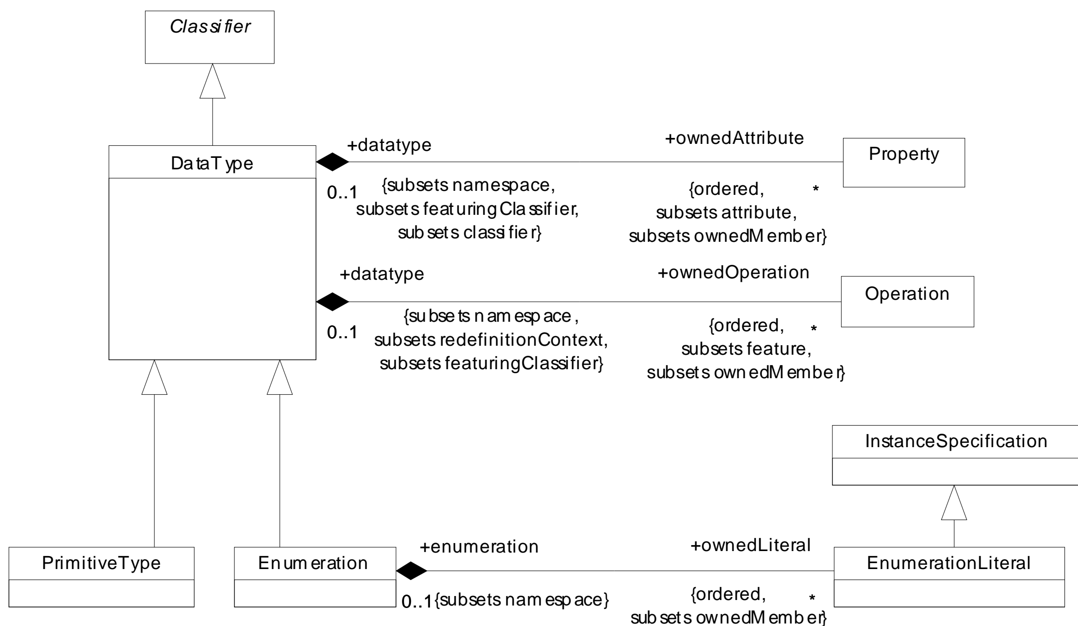

## Diagrama de clases de *Features*



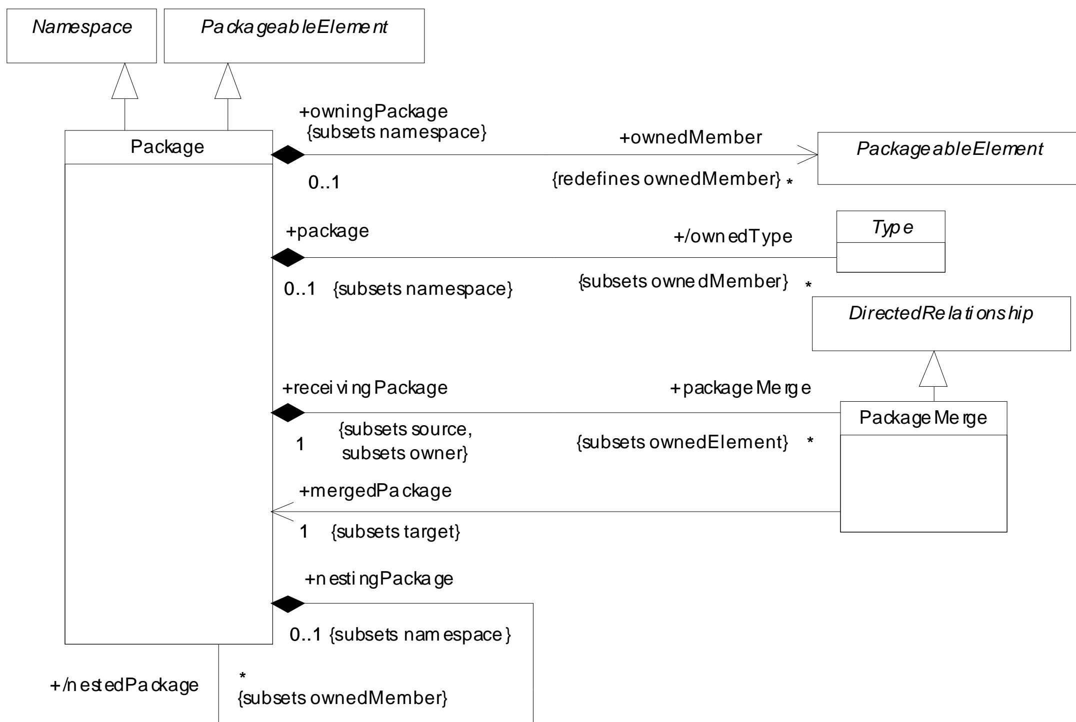

## Diagrama de clases de *Operations*




## Diagrama de clases de *Classes*

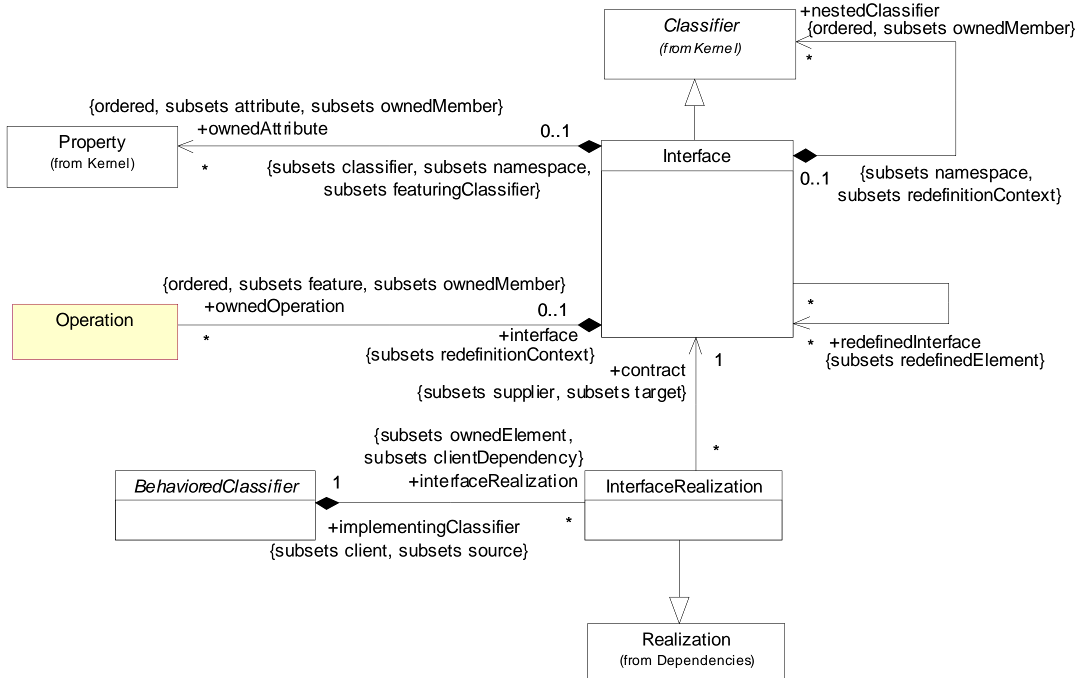



## Diagrama de clases de *Data Types*



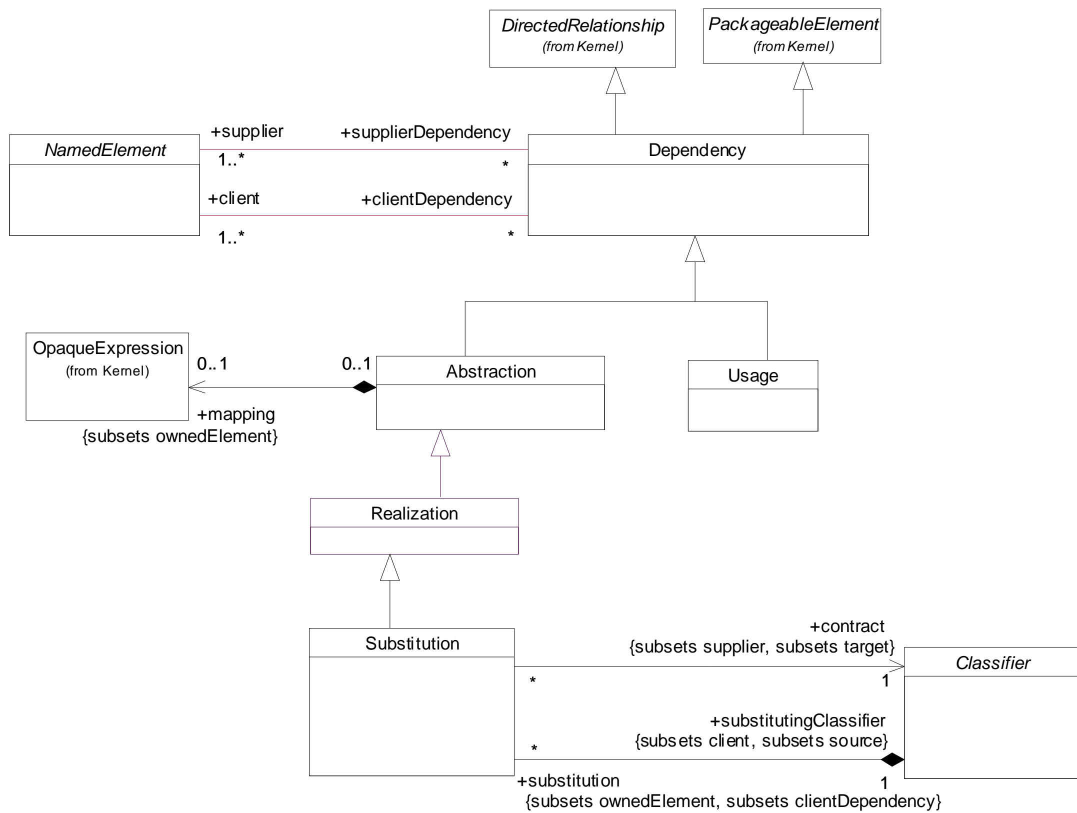

## Diagrama de clases de *Packages*

## A.2. Diagrama del paquete Interfaces




### Diagrama de clases de *Interfaces*

# A.3. Diagrama del paquete Dependencies




## Diagrama de clases de *Dependencies*

# Anexo B: Especializaciones del Metamodelo UML

En este anexo se describen parcialmente los siguientes metamodelos:

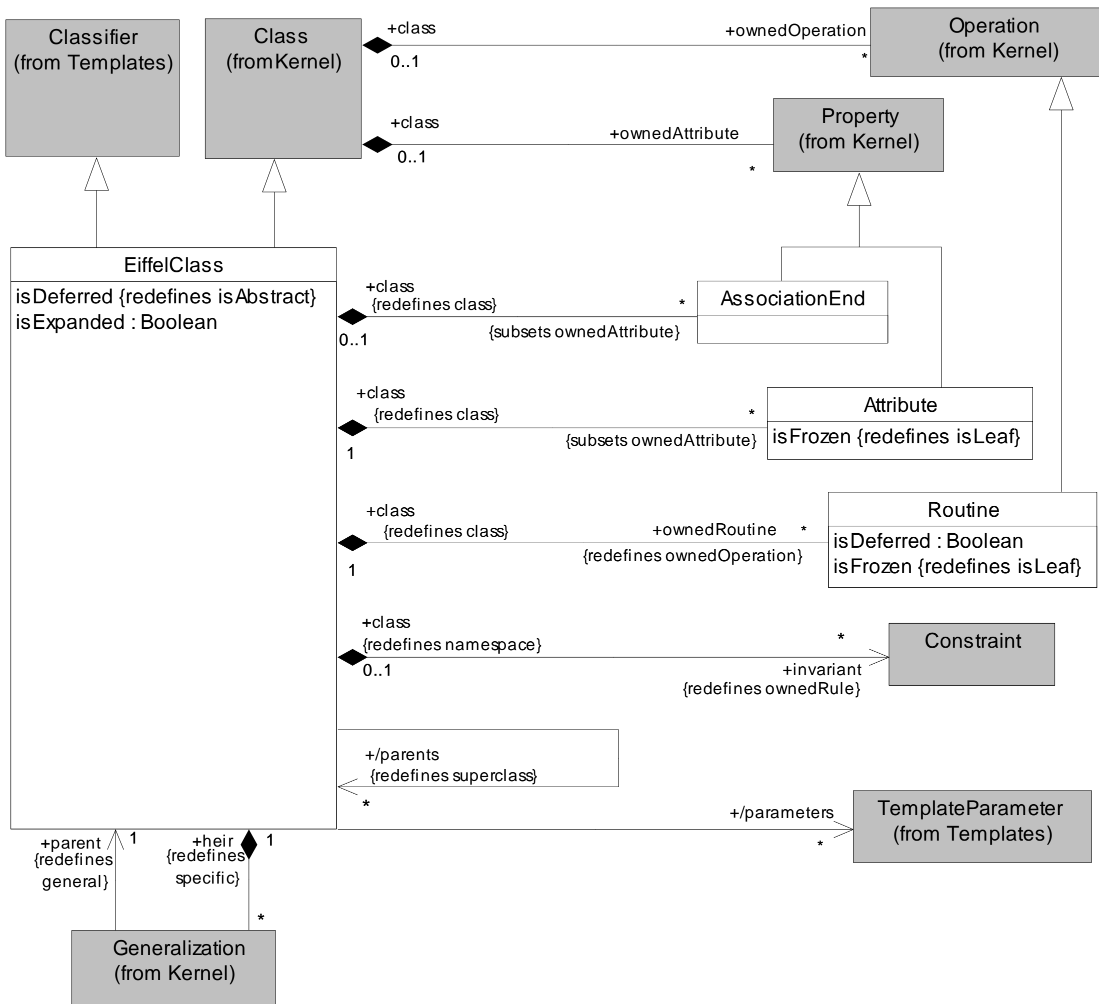
- **PSM-Eiffel:** Metamodelo específico a la Plataforma Eiffel
- **PSM-Java:** Metamodelo específico a la Plataforma Java
- **ISM-Eiffel:** Metamodelo específico a la Implementación Eiffel
- **ISM-Java:** Metamodelo específico a la Implementación Java

Cada uno de los metamodelos fue especificado utilizando la notación UML, de manera semi-formal usando la combinación de notación gráfica, lenguaje natural (español) y lenguaje formal:

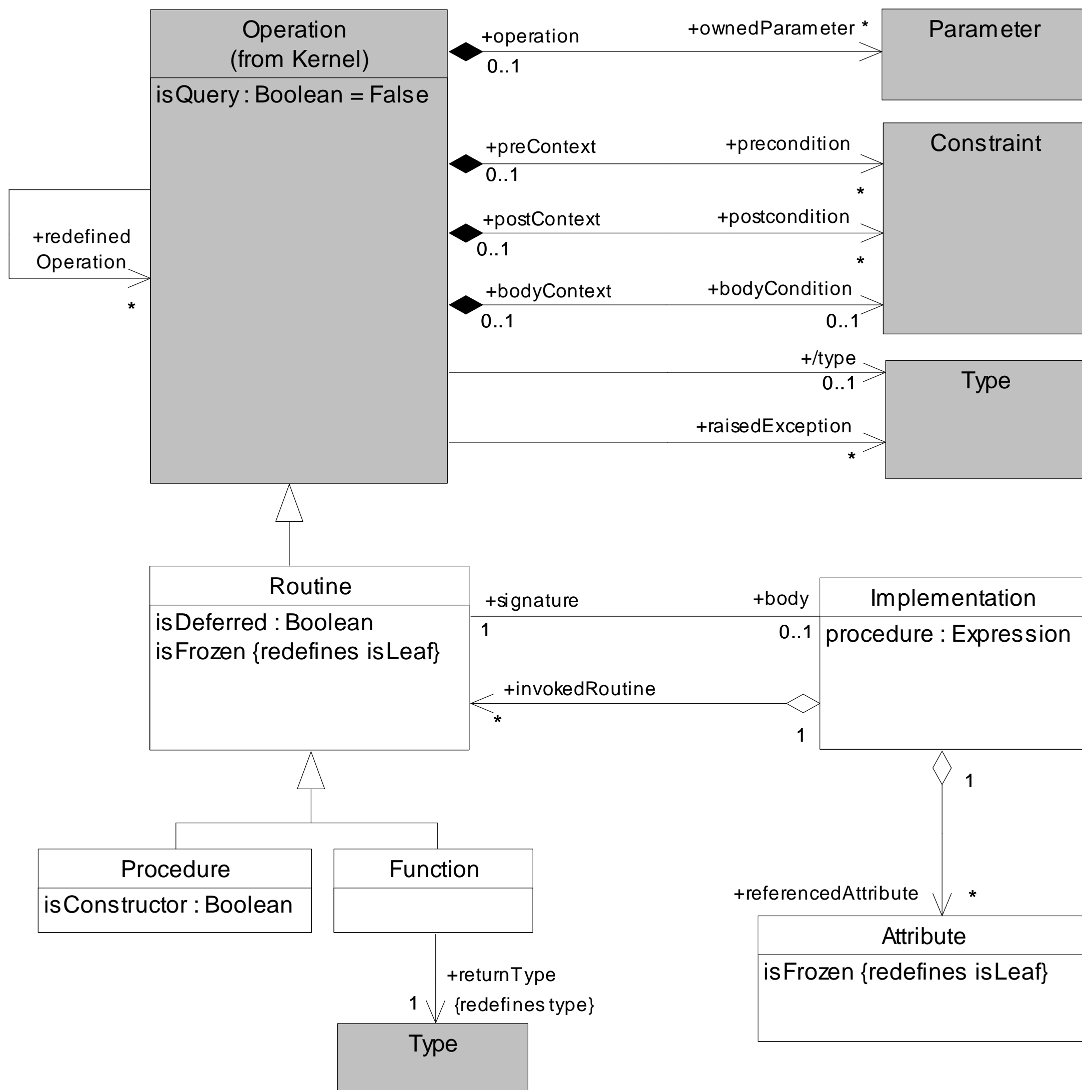
- *Sintaxis abstracta*: consiste de uno o más diagramas de clases UML que muestran las metaclases que definen las construcciones y sus relaciones. Las metaclases que aparecen en color gris oscuro corresponden a metaclases propias del metamodelo UML.
- *Descripción de las metaclases*: se utiliza el lenguaje natural para describir cada metaclase, sus generalizaciones y sus asociaciones y lenguaje formal (OCL) para escribir sus restricciones. Las metaclases son presentadas en orden alfabético.

# B.1. Metamodelo específico a la plataforma Eiffel

## Sintaxis abstracta




## Diagrama de Clases




## Diagrama de Operaciones

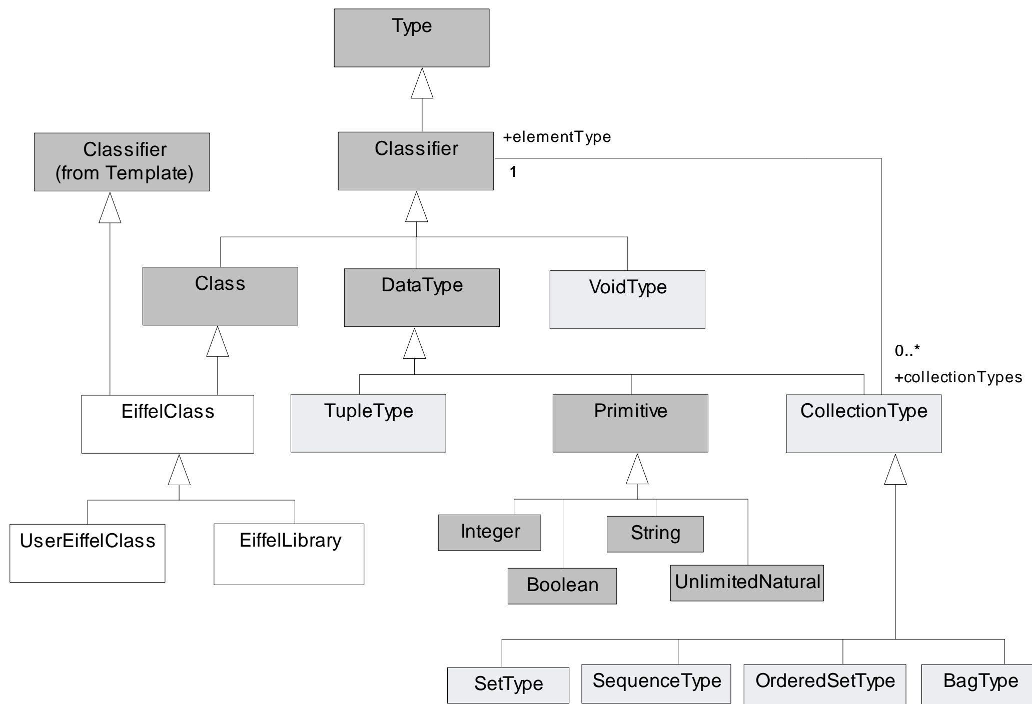



**Diagrama de Tipos UML, OCL y Eiffel**

## Descripción de las clases

## AssociationEnd

### Generalización

- Property (de Kernel)

### Descripción

Representa a los extremos de asociación propios de la clase.

### Atributos

No tiene atributos adicionales.

### Asociaciones

- class: EiffelClass [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 miembro de una asociación.
self.association->size() = 1

## Attribute

### Generalización

- Property (de Kernel)

### Descripción

Representa los atributos declarados en una clase Eiffel.

### Atributos

- isFrozen: Boolean [1] — Especifica si un atributo es frozen, es decir, si es una constante. Si es frozen debe tener valor inicial obligatoriamente. Redefine RedefinableElement::isLeaf.

### Asociaciones

- class: EiffelClass [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()

## EiffelClass

### Generalizaciones

- Class (de Kernel), Classifier (de Templates).

### Descripción

Una clase Eiffel describe un conjunto de objetos que comparten las mismas especificaciones de features, restricciones y semántica.

### Atributos

- isDeferred: Boolean [1] — Especifica si una clase es diferida, es decir, si incluye uno o más *features* especificados pero no implementados. Redefine Classifier::isAbstract.
- isExpanded: Boolean [1] — Especifica si la clase es expandida, es decir, sus instancias son objetos y no referencias a objetos.

### Asociaciones

- associationEnd: AssociationEnd [*] — Referencia a los extremos de asociación propios de la clase Eiffel. Subconjunto de *Class::ownedAttribute.*
- attribute: Attribute [*] — Referencia a las variables propias de la clase Eiffel. Subconjunto de *Class::ownedAttribute*.
- generalization: Generalization [*] — Especifica las relaciones de generalización para esta clase.

- invariant: Constraint [*] — Referencia las invariantes de la clase. Redefines *NameSpace::ownedRule*.
- /parameters: TemplateParameter [*] — Referencia el conjunto de parámetros de la clase. Es derivado.
- /parents: EiffelClass [*] — Referencia a las superclases de una clase Eiffel. Redefine *Class::superClass*. Es derivado.
- routine: Routine [*] — Referencia las operaciones propias de la clase. Redefine *Class::ownedOperation.*

### Restricciones

[1] Una clase que tiene alguna rutina diferida debe ser declarada diferida.
self.ownedRoutine -> exists (r | r.isDeferred) **implies** self. isDeferred

[2] Las rutinas privadas de una clase no pueden ser declarados abstractos.
self.ownedRoutine -> forAll (r | r.visibility = #private **implies not** r.isAbstract)

[3] Las rutinas frozen de una clase no pueden ser declaradas diferidas.
self.ownedRoutine -> forAll (r | r.isFrozen **implies not** r.isDeferred)

[4] Una clase Eiffel no tiene clases anidadas.
self.nestedClassifier-> isEmpty()

[5] parents se deriva de la relación de generalización.
parents= self.generalization.parent

[6] parameters se deriva a partir de los parámetros de la signatura template redefinible.
parameters= ownedSignature.parameter

## Function

### Generalizaciones

- Routine

### Descripción

Declara una función que puede ser invocada pasando una cantidad fija de argumentos.

### Atributos

No tiene atributos adicionales.

### Asociaciones

- returnType: Type[1] — Referencia el tipo de retorno de la función. Redefine *Operation::type*.

### Restricciones

[1] Una función debe tener un tipo de retorno, por lo tanto existe en el conjunto de argumentos de la misma, uno cuyo tipo es de retorno.

self.ownedParameter-> select (p | p.direction= #return) -> size= 1

## Implementation

### Generalización

- Element (de Kernel)

### Descripción

Especifica un procedimiento que lleva a cabo el resultado de una rutina.

### Atributos

- procedure: Expression [0..1] Referencia el procedimiento de la rutina.

### Asociaciones

- invokedRoutine: Routine [*] Referencia a las rutinas invocadas en esta implementación.
- referencedAttribute: Field [*] Especifica las variables referenciadas en esta implementación.
- signature: Routine [1] Referencia la rutina a la que corresponde esta implementación.

### Restricciones

[1] Una rutina no puede invocar a un constructor
self.invokedRoutine->select(r | r.oclIsTypeOf(Procedure)) -> forAll(p |
**not** p.oclAsType(Procedure).isConstructor)

## Procedure

### Generalizaciones

- Routine

### Descripción

Declara un procedimiento que puede ser invocado pasando una cantidad fija de argumentos.

### Atributos

- isConstructor: Boolean [1] Determina si el procedimiento es constructor.

### Asociaciones

No tiene asociaciones adicionales.

### Restricciones

[1] Un procedimiento no tiene un tipo de retorno
self.ownedParameter->select(p | p.direction= #return)-> isEmpty()

[2] El constructor de una clase no puede ser abstracto
self.isConstructor **implies not** self.isDeferred

## Routine

### Generalizaciones

- Operation (de Kernel)

**Descripción**
Especifica las características de una rutina Eiffel.

**Atributos**

- isDeferred: Boolean [1] — Especifica si una rutina es diferida, es decir, si no tiene implementación.
- isFrozen: Boolean [1] — Especifica si una rutina es final, es decir, si no puede ser redefinida en una clase descendiente. Redefine *RedefinableElement::isLeaf.*

**Asociaciones**

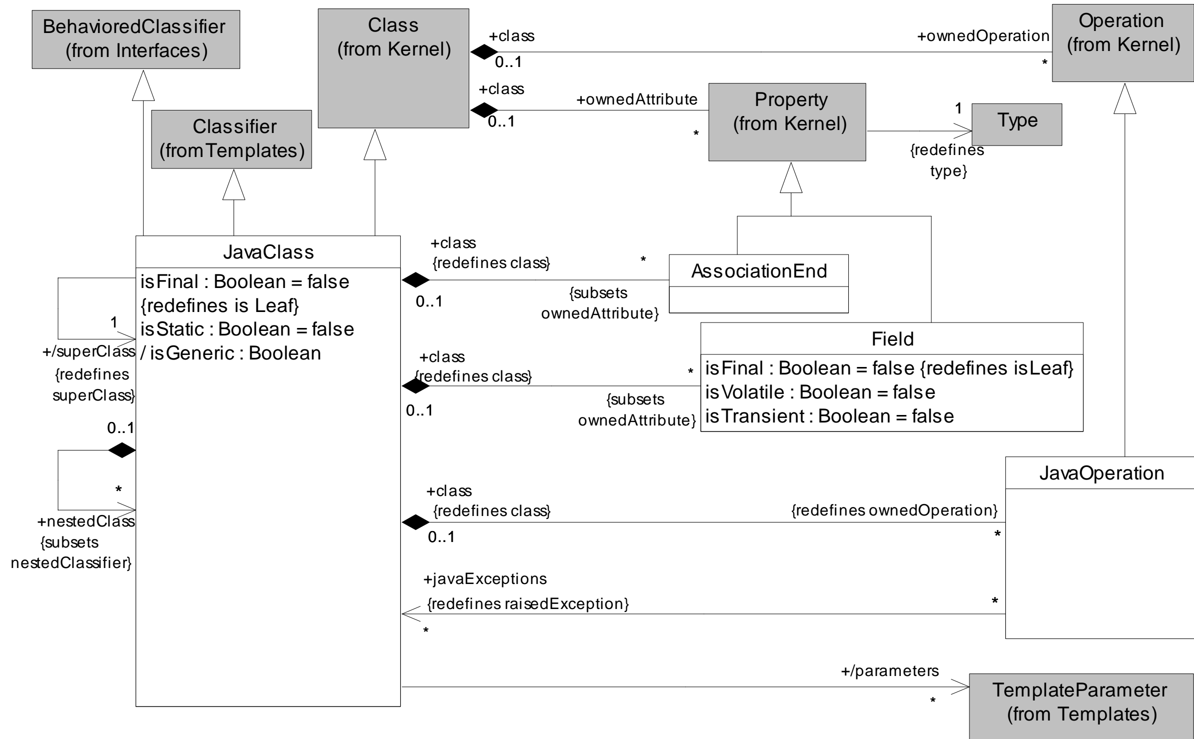
- body: Implementation [0..1] — Referencia a la implementación de la rutina.
- class: EiffelClass [1] — Referencia a la clase que declara esta rutina. Redefine *Operation::class*.

**Restricciones**

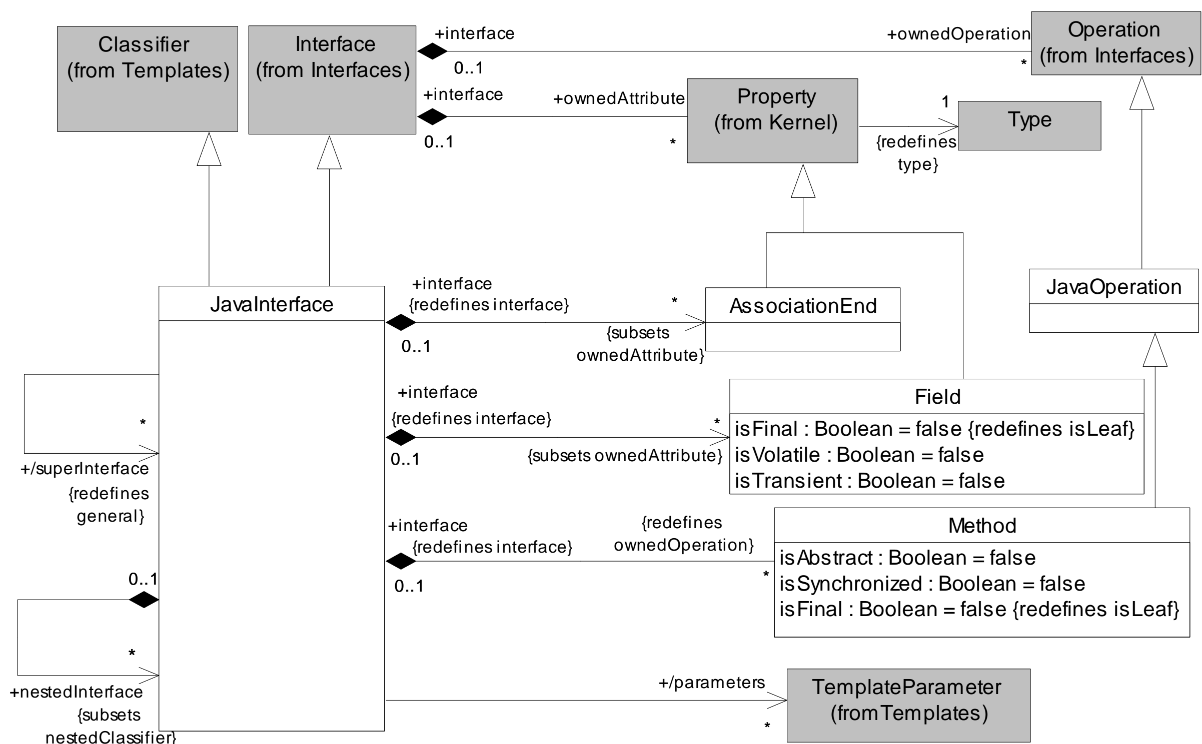
[1] Si una rutina es diferida no tiene implementación.
self.isDeferred **implies** self.body-> isEmpty()

## B.2. Metamodelo específico a la plataforma Java

### Sintaxis abstracta




### Diagrama de Clases Java




### Diagrama de Interfaces Java

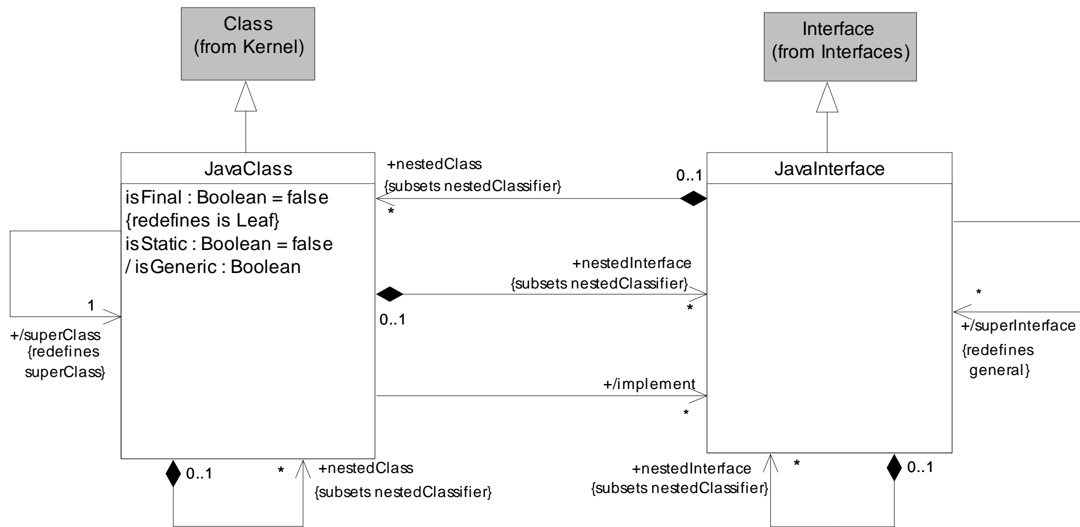



**Diagrama de relaciones entre clases e interfaces Java**

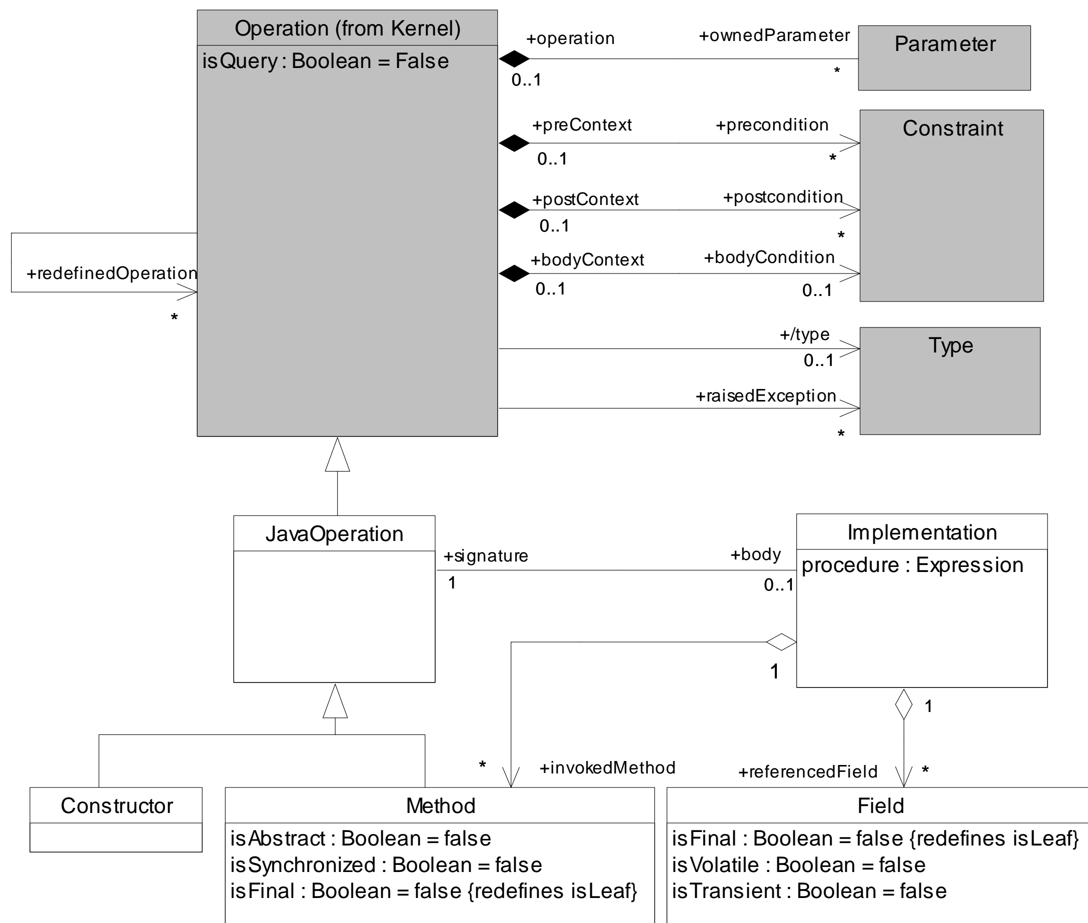
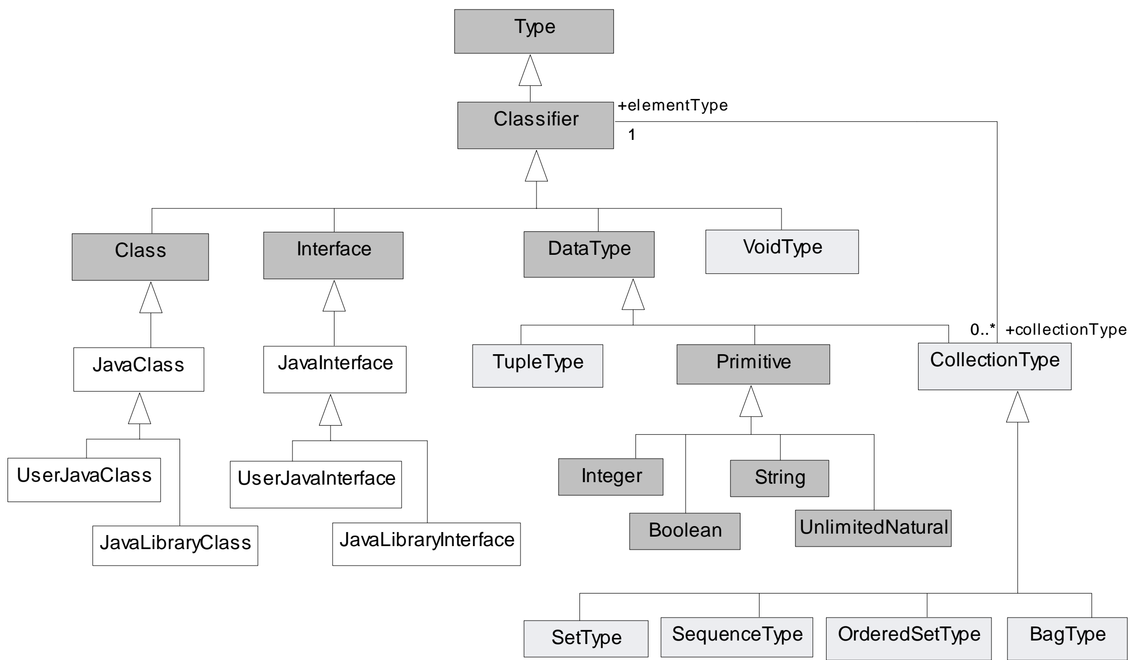
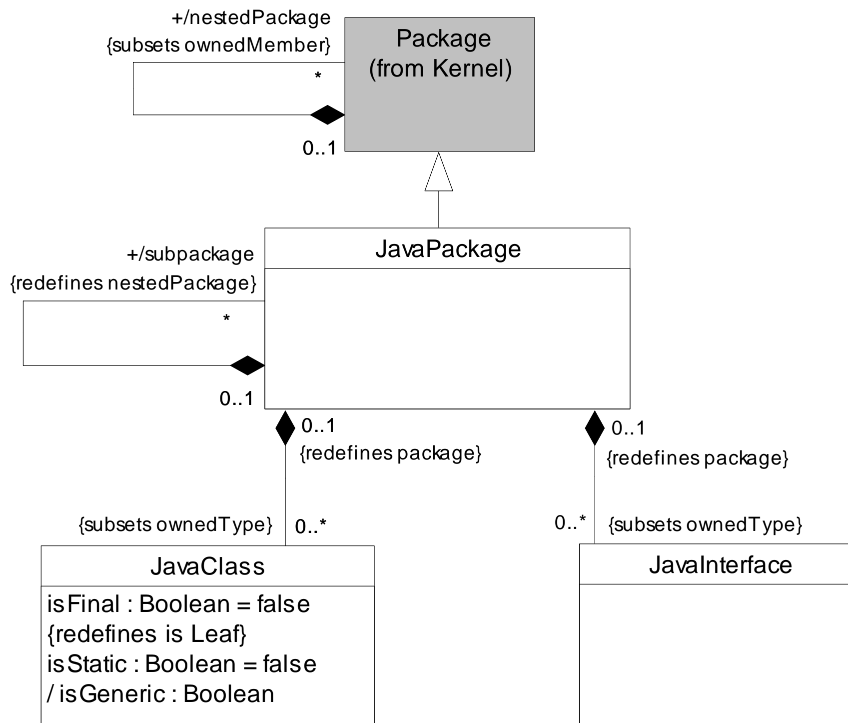



**Diagrama de Operaciones Java**




## Diagrama de Tipos (UML, OCL y Java)




## Diagrama de Paquete Java

## Descripción de las clases

A continuación se describen las principales metaclases del metamodelo específico a la plataforma Java.

### AssociationEnd

**Generalización**
- Property (de Kernel)

**Descripción**
Representa a los extremos de asociación propios de la clase.

**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

**Generalización**
- JavaOperation

**Descripción**
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.

**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

### Generalización
- Property (de Kernel)

### Descripción
Especifica los atributos declarados en una clase o en una interfaz.

### 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.*
- isTransient: Boolean [1] — Especifica si un atributo es parte del estado persistente del objeto.
- isVolatile: Boolean [1] — Especifica si un atributo es volátil, es decir, si es accedido de forma asincrónica.

### 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

### Generalización
- Element (de Kernel)

### Descripción
Especifica el procedimiento de la operación.

### Atributos
- Procedure: Expression [0..1] — Referencia el procedimiento de la operación.

### Asociaciones
- 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.
- signature: JavaOperation [1] — Especifica la operación que ésta implementa.

### Restricciones
No hay restricciones adicionales.

## JavaClass

### Generalizaciones

- Class (de Kernel), Classifier (de Templates), BehavioredClassifier (de Interfaces)

### Descripción

Una clase Java describe un conjunto de objetos que comparten las mismas especificaciones de *features*, restricciones y semántica.

### 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.
- isStatic: Boolean — Especifica si la clase es estática.

### Asociaciones

- associationEnd: AssociationEnd [*] — Referencia a los extremos de asociación propios de la clase Java. Subconjunto de C*lass::ownedAttribute.*
- field: Field [*] — Referencia a las variables propias de la clase Java. Subconjunto de C*lass::ownedAttribute*.
- /implement: JavaInterface [*] — Referencia a las interfaces Java implementadas por esta clase. Es derivado.
- javaOperation: JavaOperation [*] — Referencia las operaciones propias de la clase. Redefine C*lass::ownedOperation*.
- javaPackage: JavaPackage [0..1] — Referencia el paquete en el cual está declarada. Redefine *Type::package.*.
- nestedClass: JavaClass [*] — Referencia a las clases Java que están declaradas dentro del cuerpo de una JavaClass (clases anidadas).Subconjunto de C*lass::nestedClassifier.*
- nestedInterface: JavaInterface [*] — Referencia a las interfaces Java que están declaradas dentro del cuerpo de una JavaClass (interfaces anidadas). Subconjunto de C*lass::nestedClassifier.*
- /parameters: TemplateParameter [*] — Referencia el conjunto de parámetros de la clase. Es derivado.
- /superClass: JavaClass [1] — Referencia a la superclase de una clase Java. Redefine *Class::superClass*. Es derivado.

### Restricciones

[1] Los clasificadores anidados de una clase o de una interfaz Java solo 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

### Generalizaciones

- Interface (de Interfaces), Classifier (de Templates)

### Descripción

Describe las características de las interfaces en la plataforma Java.

### Atributos

No tiene atributos adicionales.

### Asociaciones

- 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*.
- javaPackage: JavaPackage [0..1] — Referencia el paquete en el cual está declarada. Redefine *Type::package..*
- method: Method [*] — Referencia los métodos propios de una JavaInterface. Redefine *Interface::ownedOperation*.
- 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.*

### 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 solo pueden ser del tipo JavaClass o JavaInterface.
```
self.nestedClassifier->forAll(c | c.oclIsTypeOf(JavaClass) or c.oclIsTypeOf(JavaInterface))
```

[4] Una interfaz solo 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 solo 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

### Generalizaciones

- Operation (de Kernel, de Interfaces)

**Descripción**
Es una operación de una clase Java.

**Atributos**
No tiene atributos adicionales.

**Asociaciones**

- class: JavaClass [0..1] — Referencia a la clase que declara esta operación. Redefine *Operation::class*.
- body: Implementation [0..1] — Referencia a la implementación de la operación.
- javaException: JavaClass [*] — Referencia a los tipos que representan las excepciones que pueden surgir durante una invocación de esta operación.

**Restricciones**

[1] Si una operación es abstracta no tiene implementación
self.isAbstract **implies** self.body->isEmpty()

## JavaPackage

**Generalizaciones**
- Package (de Kernel)

**Descripción**
Es utilizado para agrupar elementos. Sus miembros pueden ser tipos clases o interfaces y subpaquetes.

**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

**Generalizaciones**
- JavaOperation

**Descripción**
Declara una operación que puede ser invocada pasando una cantidad fija de argumentos.

### Atributos

- isAbstract: Boolean [1] — Especifica si un método es abstracto. Es verdadero si no tiene implementación.
- isFinal: Boolean [1] — Especifica si un método es final. Si es verdadero no puede ser sobrescrito en una clase derivada. Redefine *RedefinableElement::isLeaf.*
- isSyncronized: Boolean [1] — Especifica si un método es sincronizado. Es verdadero si adquiere un *lock* antes de su ejecución.

### Asociaciones

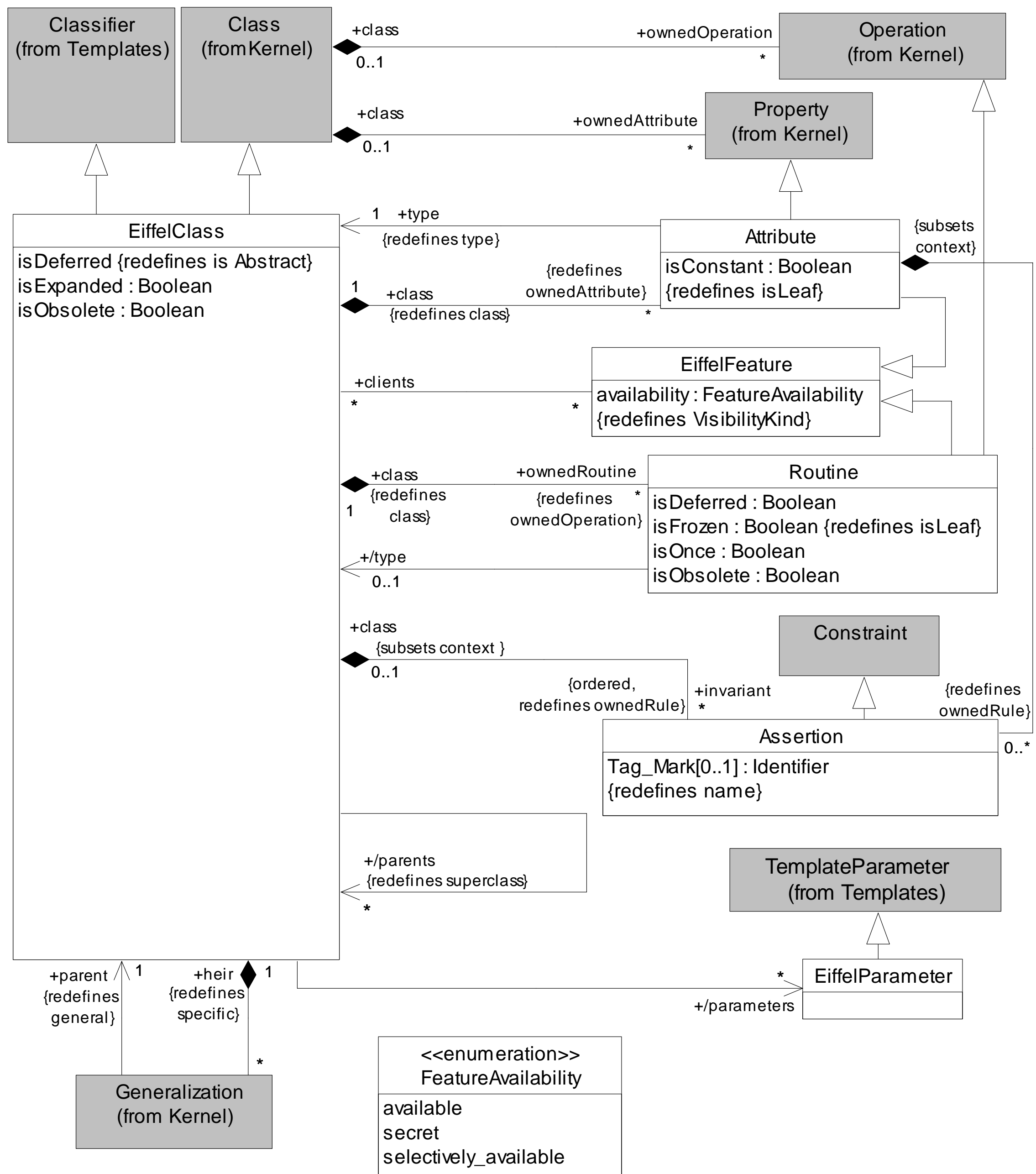
- interface: JavaInterface [0..1] — Declara a la interfaz que declara este método. Redefine *Operation::interface*.

### Restricciones

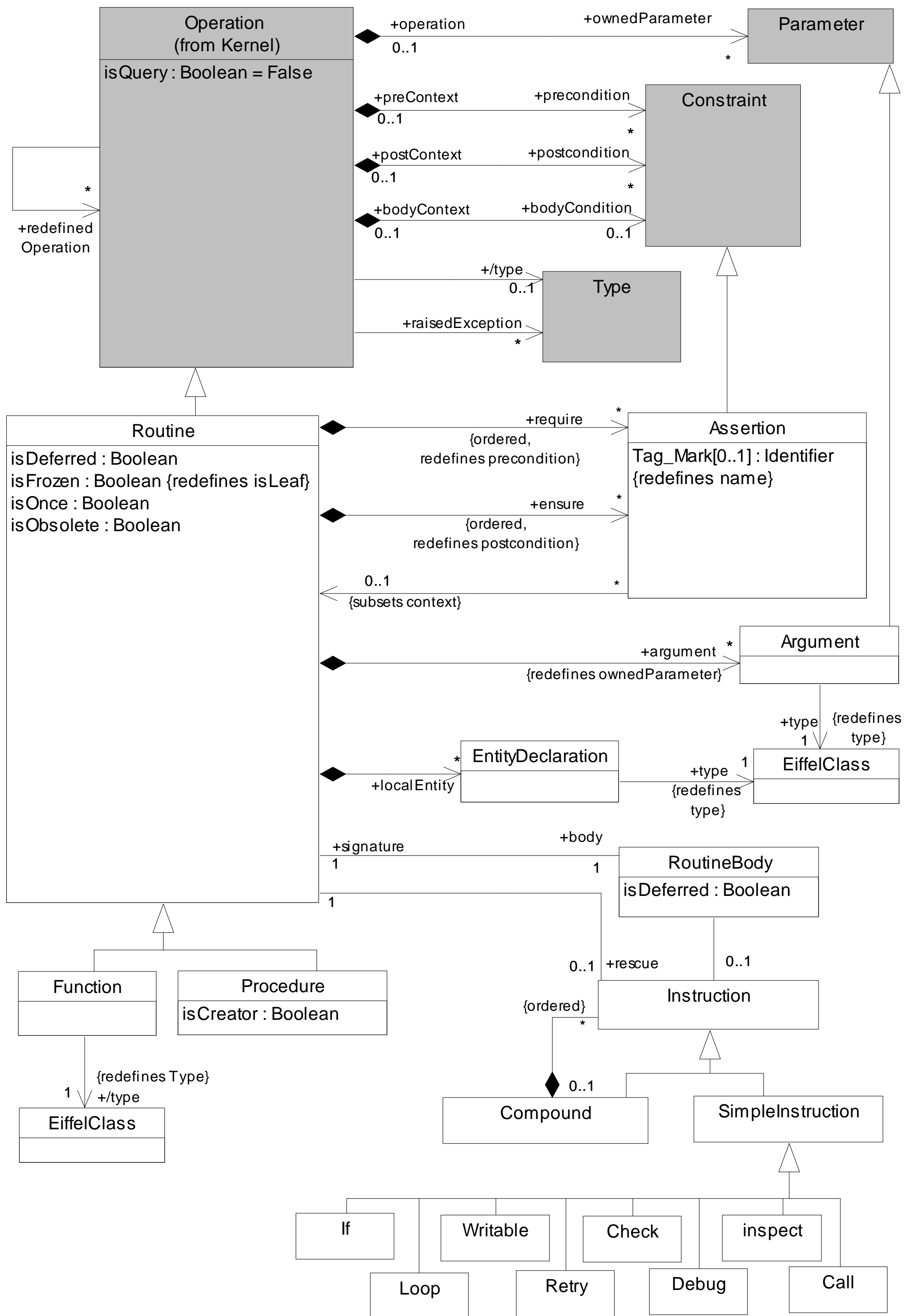
No hay restricciones adicionales.

## B.3. Metamodelo específico a la implementación Eiffel

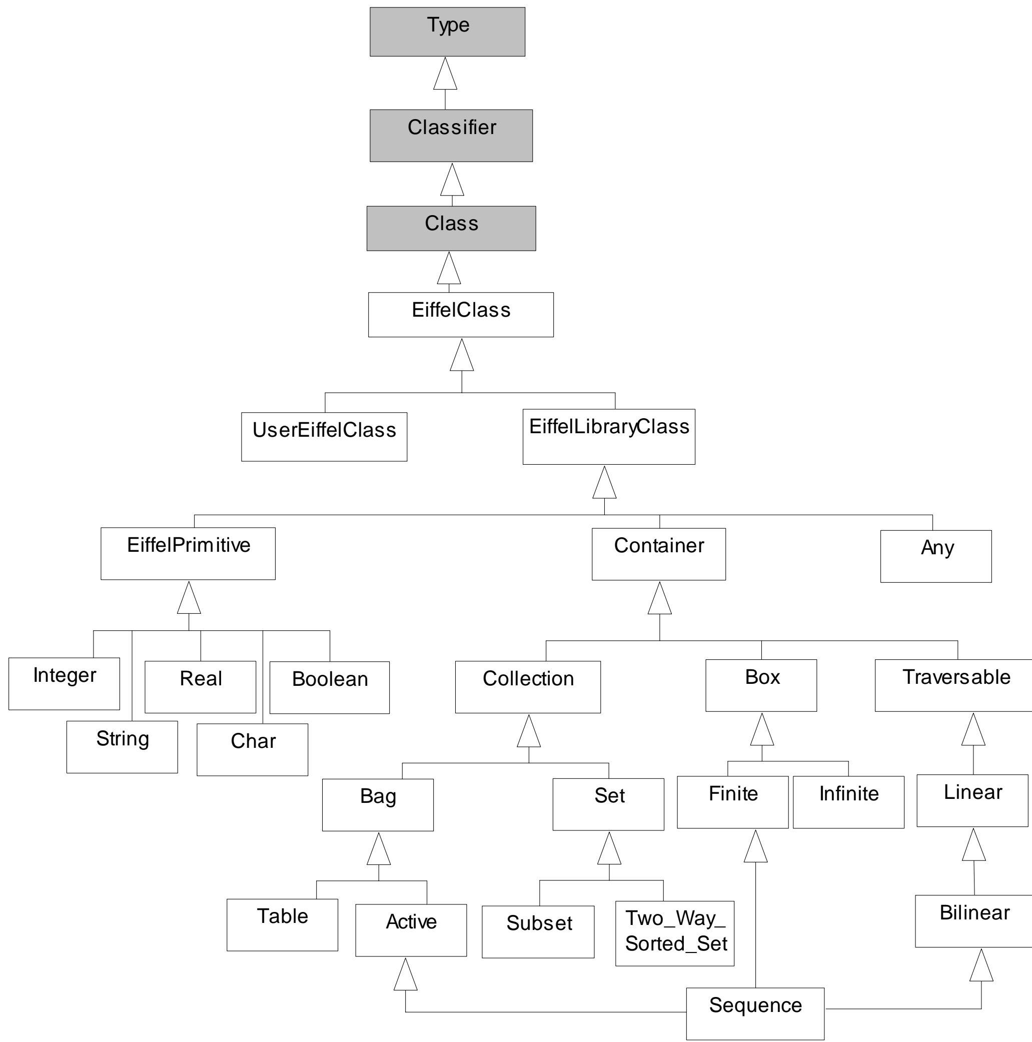
### Sintaxis abstracta



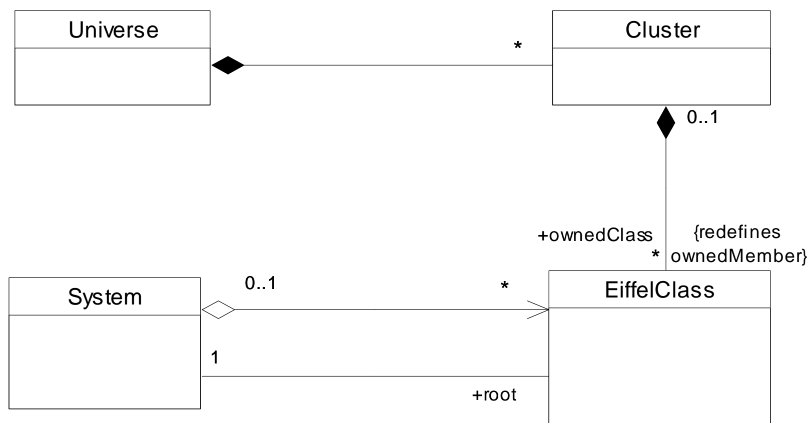

### Diagrama de Clases Eiffel




## Diagrama de Operaciones Eiffel




## Diagrama de Tipos




## Diagrama de Clusters

## Descripción de las clases

## Attribute

### Generalización
- Property (de Kernel).

### Descripción
Representa los atributos declarados en una clase, según la especificación del lenguaje Eiffel.

### Atributos
- isConstant: Boolean [1] Especifica si un atributo es constante. Si es constante debe tener valor inicial obligatoriamente. Redefine *RedefinableElement::isLeaf.*

### Asociaciones
- class: EiffelClass [1] Referencia a la clase que declara este atributo. Redefine *Property::class*.
- type: EiffelClass [1] Referencia el tipo de este atributo. Redefine *TypedElement::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()

## Assertion

### Generalización
- Constraint (de Kernel)

### Descripción
Describe las aserciones, según la especificación del lenguaje Eiffel.

### Atributos
- Tag_Mark: Identifier [0..1] Referencia el identificador de la aserción. Redefine *NamedElement::name*.

### Asociaciones
- class [0..1]: EiffelClass Referencia a la clase que es el contexto en el cual se evalúa esta restricción. Subconjunto de *Constraint::context.*
- routine [0..1]: Routine Referencia a la rutina que es el contexto en el cual se evalúa esta restricción. Subconjunto de *Constraint::context.*

### Restricciones
No tiene restricciones adicionales.

## Argument

### Generalización
- Parameter (de Kernel)

**Descripción**
Describe los argumentos de una rutina, según la especificación del lenguaje Eiffel.

**Atributos**
No tiene atributos adicionales.

**Asociaciones**
- type: EiffelClass [1] Referencia el tipo de este argumento. Redefine *TypedElement::type*.

**Restricciones**
No tiene restricciones adicionales.

## Cluster

**Generalización**
- Package (de Kernel)

**Descripción**
Se utilizan para agrupar clases. Los clusters son esenciales para la organización de las clases en Eiffel.

**Atributos**
No tiene atributos adicionales.

**Asociaciones**
- ownedClass: EiffelClass [*] Referencia las clases Eiffel que son miembros de este cluster. Redefine *Package::ownedType*.

**Restricciones**
No tiene restricciones adicionales.

## Compound

**Generalización**
- Element (de Kernel)

**Descripción**
Describe un conjunto de instrucciones, según la especificación del lenguaje Eiffel.

**Atributos**
No tiene atributos adicionales.

**Asociaciones**
- instruction: Instruction [*] Especifica el conjunto de instrucciones que forman el *compound*. Es ordenado.

**Restricciones**
No tiene restricciones adicionales.

## EntityDeclaration

### Generalización
- TypedElement (de Kernel)

### Descripción
Describe una entidad local de una rutina, según la especificación del lenguaje Eiffel.

### Atributos
No tiene atributos adicionales.

### Asociaciones
- type [1]: EiffelClass — Especifica el tipo de la entidad. Redefine *TypedElement::type*.

### Restricciones
No tiene restricciones adicionales.

## EiffelClass

### Generalizaciones
- Class (de Kernel), Classifier (de Templates)

### Descripción
Una clase Eiffel describe un conjunto de objetos que comparten las mismas especificaciones de features, restricciones y semántica, según la especificación del lenguaje Eiffel.

### Atributos
- isDeferred: Boolean [1] — Especifica si una clase es diferida, es decir, si incluye uno o más *features* especificados pero no implementados. Redefine *Classifier::isAbstract*.
- isExpanded: Boolean [1] — Especifica si la clase es expandida, es decir, sus instancias son objetos y no referencias a objetos.
- isObsolete: Boolean [1] — Especifica si la clase es obsoleta.

### Asociaciones
- attribute: Attribute [*] — Referencia los atributos propios de la clase Eiffel. Redefine *Class::ownedAttribute*.
- eiffelFeatures: EiffelFeature [*] — Referencia los features de los cuales esta clase es cliente.
- generalization: Generalization [*] — Especifica las relaciones de generalización para esta clase.
- invariant: Assertion [*] — Referencia las invariantes de la clase. Redefines *NameSpace::ownedRule*.
- ownedRoutine: Routine [*] — Referencia las rutinas propias de la clase. Redefine *Class::ownedOperation.*
- /parameters: EiffelParameter [*] — Referencia el conjunto de parámetros de la clase. Es derivado.
- /parent: EiffelClass [*] — Referencia las clases padres de una clase Eiffel. Redefine *Class::superClass*. Es derivado.

### Restricciones

[1] Una clase que tiene alguna rutina diferida debe ser declarada diferida.
self.ownedRoutine -> exists (r | r.isDeferred) **implies** self. isDeferred

[2] Las rutinas secretas de una clase no pueden ser declaradas diferidas.
self.ownedRoutine -> forAll (r | r.availability = #secret **implies not** r.isDeferred)

[3] Las rutinas frozen de una clase no pueden ser declaradas diferidas.
self.ownedRoutine -> forAll (r | r.isFrozen **implies not** r.isDeferred)

[4] Una clase Eiffel no tiene clases anidadas.
self.nestedClassifier-> isEmpty()

[5] ancestors se deriva de la relación de generalización.
ancestors= self.generalization.parent

[6] parameters se deriva a partir de los parámetros de la signatura template redefinible.
parameters= ownedSignature.parameter

[7] Los parámetros de una clase son del tipo EiffelClass
self.parameters.parameteredElement->forAll (p| p.oclIsTypeOf(EiffelClass))

[8] Una clase diferida no tiene procedimiento de creación.
self.class.isDeferred **implies**
self.ownedRoutine->select(p| p.oclIsTypOf (Procedure) **and** p.isCreator)-> isEmpty()

[9] Una clase expandida tiene un solo procedimiento de creación y éste no tiene argumentos.
self.class.isExpanded **implies**
self.ownedRoutine->select(p| p.oclIsTypeOf (Procedure) **and** p.isCreator)-> size() = 1
**and**
self.ownedRoutine->select(p| p.isCreator and p.argument->isEmpty()) -> size() = 1

[10] Una clase expandida no tiene parámetros.
self.class.isExpanded **implies** self.parameter -> isEmpty()

## EiffelParameter

### Generalizaciones

- TemplateParameter (de Templates)

### Descripción

Especifica los parámetros de una clase, según la especificación del lenguaje Eiffel.

### Atributos

No tiene atributos adicionales.

### Asociaciones

No tiene asociaciones adicionales.

### Restricciones

[1] El tipo de los parámetros de una clase son EiffelClass
self.parameteredElement->forAll (p| p.oclIsTypeOf(EiffelClass))

## EiffelFeature

### Generalizaciones
- NamedElement

### Descripción
Declara un *feature*, según la especificación del lenguaje Eiffel.

### Atributos
- availability: FeatureAvailability [1] — Referencia la disponibilidad del *feature*. Redefine *NamedElement::visibility*.

### Asociaciones
- clients: EiffelClass[*] — Referencia las clases para las cuales este *feature* es disponible.

### Restricciones
[1] Si el *feature* es disponible selectivamente, entonces debe tener asociada una lista de clientes, de lo contrario la lista de clientes es vacía.

```
if self.availability = #selectively_available
    then  self.client->size() > 0
else
    self.client->isEmpty()
endif
```

## FeatureAvailability

### Generalizaciones
- Ninguna

### Descripción
FeatureAvailability es una enumeración de los siguientes valores:
- available
- secret
- selectively_available

Los cuales determinan si un *feature* es disponible a todas las clases, a ninguna o a algunas respectivamente, según la especificación del lenguaje Eiffel.

## Function

### Generalizaciones
- Routine

### Descripción
Declara una función, según la especificación del lenguaje Eiffel.

### Atributos
No tiene atributos adicionales.

**Asociaciones**

- /type: EiffelClass[1] — Referencia el tipo de retorno de la función. Redefine *TypedElement::type*.

**Restricciones**

No tiene restricciones adicionales.

## Instruction

**Generalizaciones**

- NamedElement (de Kernel)

**Descripción**

Describe una instrucción, según la especificación del lenguaje Eiffel.

**Atributos**

No tiene atributos adicionales.

**Asociaciones**

- routineBody: RoutineBody — Referencia el cuerpo de la rutina de la cual esta instrucción forma parte.
- routine: Routine — Referencia la rutina que declara la cláusula rescue de la cual esta instrucción es parte.

**Restricciones**

No tiene restricciones adicionales.

## Routinebody

**Generalización**

- Element (de Kernel)

**Descripción**

Especifica el cuerpo de la rutina, según la especificación del lenguaje Eiffel.

**Atributos**

- is Deferred: Boolean — Especifica si el cuerpo de la rutina es diferido, es decir, si no está implementado.

**Asociaciones**

- signature: Routine [1] — Referencia la rutina a la que corresponde esta implementación.
- instruction:Instruction[0..1] — Referencia la instrucción que conforma el cuerpo de la rutina.

**Restricciones**

[1] Si el cuerpo de la rutina es diferido, la rutina que la declara también es diferida.

self.isDeferred **implies** self.signature.isDeferred

## Procedure

### Generalizaciones
- Routine

### Descripción
Declara un procedimiento, según la especificación del lenguaje Eiffel.

### Atributos
- isCreator: Boolean [1] Determina si el procedimiento es de creación.

### Asociaciones
No tiene asociaciones adicionales.

### Restricciones
[1] Un procedimiento no tiene un tipo de retorno
self.ownedParameter->select(p | p.direction= #return)->isEmpty()

[2] Si un procedimiento es de creación no puede ser diferido.
self.isCreator **implies not** self.isDeferred

## Routine

### Generalizaciones
- Operation (de Kernel), Feature

### Descripción
Especifica las características de una rutina Eiffel, según la especificación del lenguaje Eiffel.

### Atributos
- isDeferred: Boolean [1] Especifica si una rutina es diferida, es decir, si no tiene implementación.
- isFrozen: Boolean [1] Especifica si una rutina es final, es decir, si no puede ser redefinida en una clase descendiente. Redefine *RedefinableElement::isLeaf.*
- isOnce: Boolean [1] Especifica si la rutina es ejecutada sólo una vez.
- isObsolete: Boolean [1] Especifica si la rutina es obsoleta.

### Asociaciones
- argument: Argument [*] Referencia los argumentos formales de la rutina. Redefine *Operation::OwnedParameter*.
- body: RoutineBody [1] Referencia a la implementación de la rutina.
- class: EiffelClass [1] Referencia a la clase que declara esta rutina. Redefine *Operation::class*.
- ensure: Assertion [*] Especifica las postcondiciones de la rutina. Redefine *Operation::postcondition.*
- localEntity: EntityDeclaration [*] Especifica las entidades locales de la rutina.

- require: Assertion [*] — Especifica las precondiciones de la rutina. Redefine *Operation::precondition.*
- rescue: Instruction [0..1] — Especifica la respuesta a una excepción ocurrida durante la ejecución de la rutina.

### Restricciones

[1] Si una rutina es diferida no tiene implementación.
self.isDeferred **implies** self.body->isEmpty()

[2] Si una rutina es frozen no puede ser diferida.
self.isFrozen **implies** self.isDeferred

## SimpleInstruction

### Generalizaciones

- NamedElement (de Kernel)

### Descripción

Describe una instrucción simple, según la especificación del lenguaje Eiffel.

### Atributos

No tiene atributos adicionales.

### Asociaciones

No tiene asociaciones adicionales.

### Restricciones

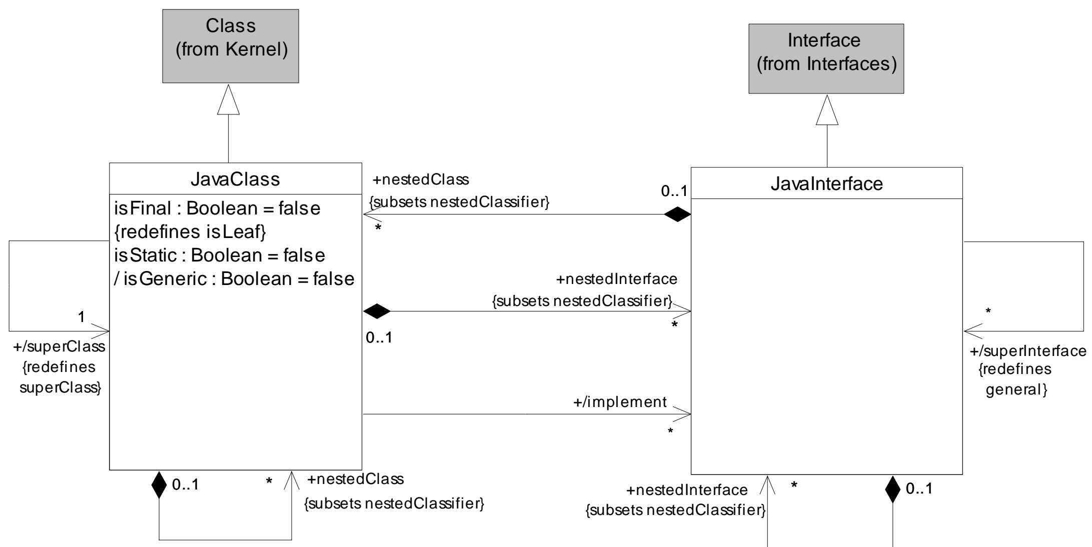
No tiene restricciones adicionales.

## B.4. Metamodelo específico a la implementación Java

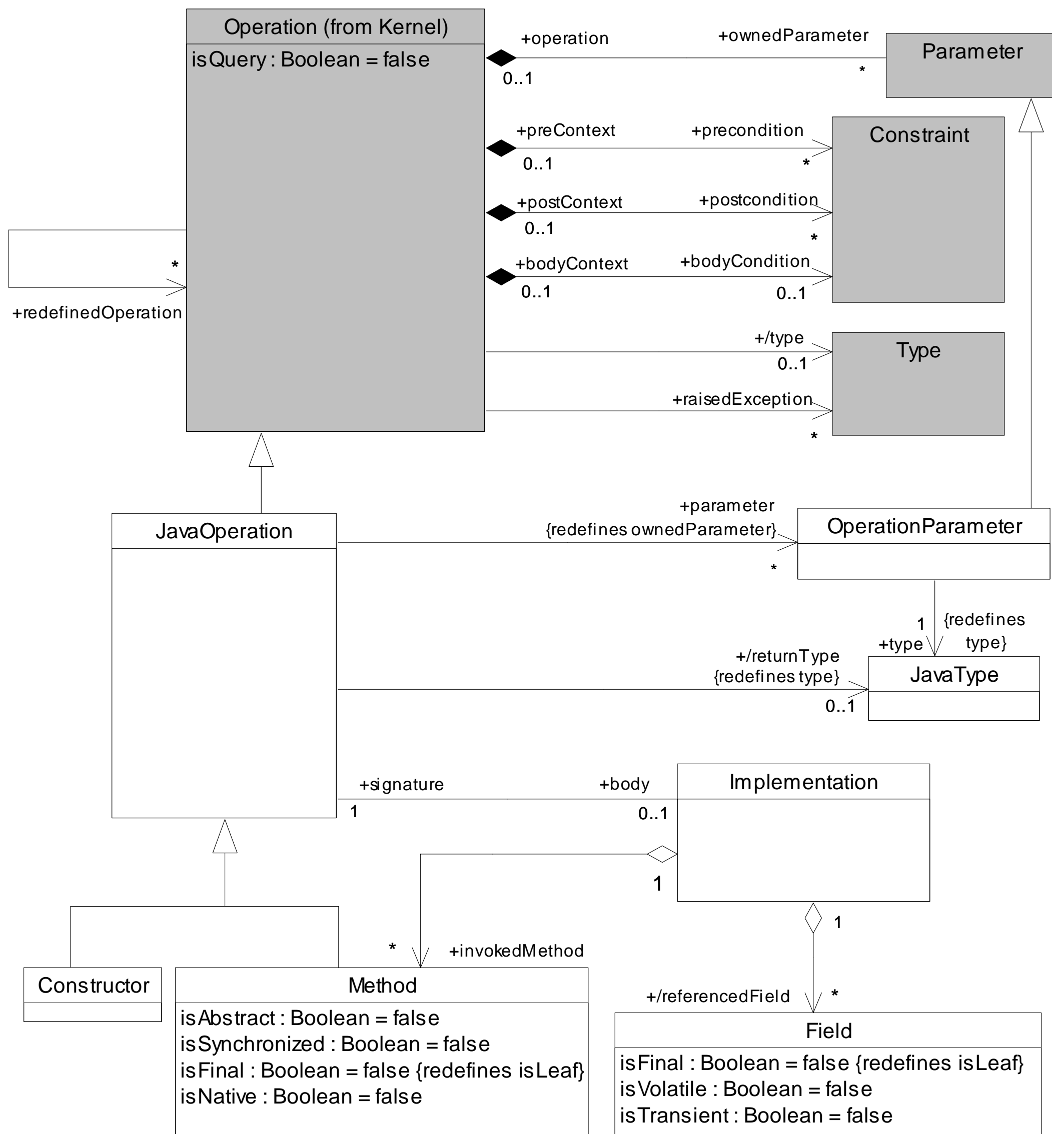
### Sintaxis abstracta

**Diagrama de Clases**

**Diagrama de Interfaces**




## Diagrama de relaciones entre clases e interfaces Java




## Diagrama de operaciones

## Diagrama de Implementación

## Diagrama de Tipos

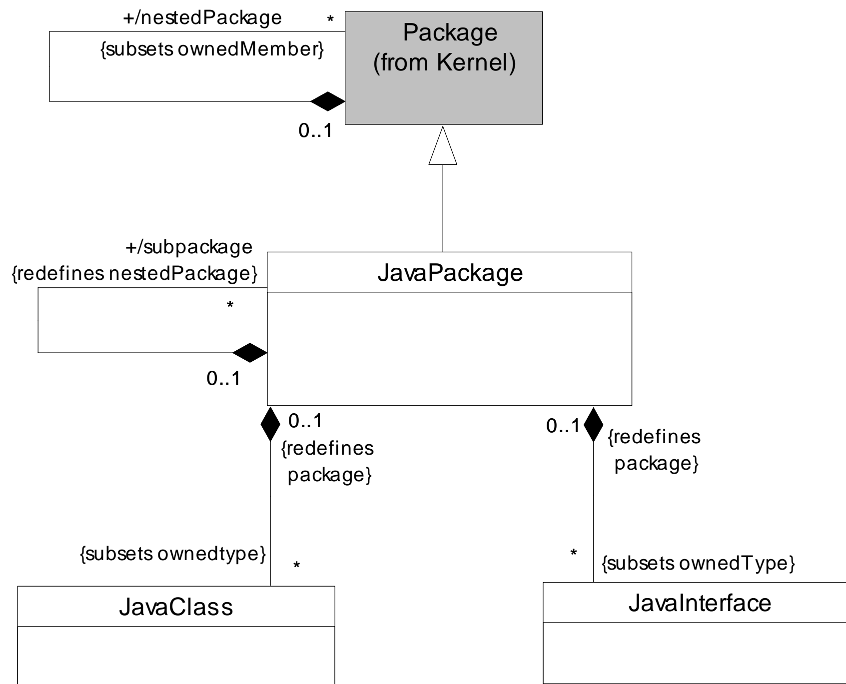



**Diagrama de Paquetes**

## 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

### Generalizaciones

- Action (de Action)

### Descripción

Especifica el bloque de código que implementa una operación, como está definido en la Especificación del Lenguaje Java.

### Atributos

No tiene atributos adicionales

### Asociaciones

- blockStatement: blockStatement [0..1] Referencia el bloque de sentencias de la cual ésta es parte.
- implementation: Implementation [1] Referencia la implementación de la cual ésta es parte.

### Restricciones

No tiene restricciones adicionales.

## Constructor

### Generalización
- JavaOperation

### Descripción
Un constructor, como está definido en la Especificación del Lenguaje Java.

### 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

### Generalización
- Property (de Kernel)

### Descripción
Representa un atributo, como está definido en la Especificación del Lenguaje Java.

### 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.*
- isTransient: Boolean [1] — Especifica si un atributo es parte del estado persistente del objeto.
- isVolatile: Boolean [1] — Especifica si un atributo es volátil, es decir, si es accedido de forma asincrónica.

### 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

**Generalización**
- Element (de Kernel)

**Descripción**
Especifica un procedimiento que lleva a cabo el resultado de la operación.

**Atributos**
No tiene atributos adicionales

**Asociaciones**

| | |
|---|---|
| • block: Block [1] | Especifica el bloque de código de la implementación. |
| • 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. |
| • signature: JavaOperation [1] | Especifica la operación que ésta implementa. |

**Restricciones**
No hay restricciones adicionales.

## JavaClass

**Generalizaciones**
- Class (de Kernel), Classifier (de Templates), BehavioredClassifier (de Interfaces)

**Descripción**
Una clase Java, como está definida en la Especificación del Lenguaje Java.

**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. |
| • isStatic: Boolean | Especifica si la clase es estática. |

**Asociaciones**

| | |
|---|---|
| • field: Field [*] | Referencia a las variables propias de la clase Java. Redefine de *Class::ownedAttribute*. |
| • /implement: JavaInterface [*] | Referencia a las interfaces Java implementadas por esta clase. 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.* |
| • nestedClass: JavaClass [*] | Referencia a las clases Java que están declaradas dentro del |

- (continuación) 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.*
- /parameters: JavaParametes [*] — Referencia el conjunto de parámetros de una clase. Es derivado.
- /superClass: JavaClass [1] — Referencia a la superclase de una clase Java. Redefine *Class::superClass*. Es derivado.

**Restricciones**

[1] Los clasificadores anidados de una clase o de una interfaz Java solo 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

### Generalizaciones

- Interface (de Interfaces), Classifier (de Templates).

### Descripción

Describe las características de una interfaz según la Especificación del Lenguaje Java.

### Atributos

No tiene atributos adicionales.

### Asociaciones

- field: Field [*] — Referencia los campos propios de una JavaInterface. Redefine de *Interface::ownedAttribute*.
- javaPackage: JavaPackage [0..1] — Referencia el paquete en el cual está declarada. Subsets *Type::package*.
- method: Method [*] — Referencia los métodos propios de una JavaInterface. Redefine *Interface::ownedOperation*.
- 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.*
- /parameter: JavaParameter [1] — Referencia el conjunto de parámetros de una interfaz. Es derivado
- /superInterface: JavaInterface [*] — Referencia las superinterfaces de una JavaInterface. Es derivada. Redefine *Classifier::general.*

### 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 solo pueden ser del tipo JavaClass o JavaInterface.
self.nestedClassifier->forAll(c | c.oclIsTypeOf(JavaClass) **or** c.oclIsTypeOf(JavaInterface))

[4] Una interfaz solo 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 solo 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

### Generalizaciones

- Operation (de Kernel)

### Descripción

Describe un método según la Especificación del Lenguaje Java.

### Asociaciones

- body: Implementation [0..1] Referencia a la implementación de la operación.

- 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*.
- 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

### Generalizaciones

- Package (de Kernel)

### Descripción

Es un paquete Java, como está definido en la Especificación del Lenguaje Java.

### 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 solo pueden ser clases, interfaces o subpaquetes.
self.ownedMember -> forAll (m | m.oclIsTypeOf ( JavaInterface) **or**
m.oclIsTypeOf ( JavaClass) **or** m.oclIsTypeOf ( JavaPackage) )

## Method

### Generalizaciones

- JavaOperation

### Descripción

Describe un método según está definido en la Especificación del Lenguaje Java.

### Atributos

- isAbstract: Boolean [1] — Especifica si un método es abstracto. Es verdadero si no tiene implementación.

- 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.
- isSyncronized: Boolean [1] — Especifica si un método es sincronizado. Es verdadero si adquiere un *lock* antes de su ejecución.

**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

**Generalization**

- Parameter (de Kernel)

**Descripción**

Especifica los parámetros de una operación según está definido en la Especificación del Lenguaje Java.

**Atributos**

No tiene atributos adicionales.

**Asociaciones**

- type: JavaType [1] — Referencia el tipo del parámetro. Redefine *TypedElement::type*.

**Restricciones**

No tiene restricciones adicionales.

# ANEXO C: Especificación parcial del metamodelo UML en NEREUS

A continuación se muestra la formalización en NEREUS de las principales clases de los paquetes Kernel e Interfaces del metamodelo UML. Las mismas aparecen en orden alfabético.

**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 …
**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
**AXIOMS** b: Boolean
get_isQuery(create(∗, b, _)) = b
**END-CLASS**

**CLASS** Package
**IS-SUBTYPE-OF** Namespace, PackageableElemnt
**ASSOCIATES** <<Package-PackageableElement>>, <<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
**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**