FACULTAD DE INFORMATICA
UNLP

TESINA DE LICENCIATURA

TITULO: Un aporte gráfico a las herramientas para transformaciones entre modelos

AUTORES: Gabriela Alejandra Pérez

DIRECTOR: Dra. Claudia Pons

CODIRECTOR: Dra. Roxana Giandini

CARRERA: Licenciatura en Informática

Resumen

La Ingeniería de Software Conducida por Modelos (MDE) es un nuevo paradigma de software que propone mejorar la construcción de software a través de un proceso guiado por modelos. MDE promete una mejora de la productividad y de la calidad del software generado ya que reduce el salto semántico entre el dominio del problema y de la solución, reduciendo también los tiempos de desarrollo. La transformación entre modelos constituye el motor de MDE y de esta manera los modelos pasan de ser entidades meramente contemplativas a ser entidades productivas.

En esta tesis se han analizado las principales herramientas MDA disponibles actualmente. Algunas utilizan estándares, como MOF, XMI, QVT, mientras que otras definen sus propios lenguajes.

La principal dificultad que se ve en el uso de estas herramientas, además de la necesidad de definir la transformación, es también la cuestión de definir los metamodelos de entrada y de salida, y del modelo de entrada. La falta de un editor gráfico que facilite estas tareas obstaculiza gravemente el uso de las herramientas.

Líneas de Investigación

Conclusiones

Se considera que el aporte de este trabajo es valioso en el área de MDD, ya que, sobre la base de un análisis cuidadoso y extenso de las necesidades existentes y de la tecnología que le da soporte, hemos generado una propuesta que facilita el proceso de desarrollo de modelos y transformaciones (a través del uso de definiciones gráficas) y que incrementa su confiabilidad (a través de la aplicación OCL). Además la propuesta se basa completamente en la aplicación de los estándares aceptados en MDD.

Trabajos Realizados

En esta tesis se han analizado las principales herramientas MDA disponibles actualmente. Además, se ha desarrollado un plugin para el entorno Eclipse que cumple los siguientes puntos:

- Permite especificar gráficamente metamodelos, brindando una forma más intuitiva para estas construcciones.
- Posee una base formal, ya que permite la edición y evaluación de restricciones definidas en OCL sobre estos metamodelos.
- Permite la instanciación de un modelo a partir del metamodelo definido. Esto es de gran ayuda, ya que da lugar a un marco de trabajo mucho más amigable para los desarrolladores

Trabajos Futuros

Como trabajo futuro se propone:

- Completar el prototipo presentado para permitir la transformación de modelos con el lenguaje estándar para transformaciones de modelos: QVT. Para esto será necesario implementar un nuevo plugin que permita definir la transformación en QVT. Al utilizar el lenguaje estándar, será posible la reutilización de la transformación y de los modelos.

- Enriquecer el evaluador OCL para permitir la evaluación de propiedades a nivel modelo.

Fecha de la presentación: Marzo del 2008

Agradecimientos:

Quiero agradecerles muy especialmente a Claudia y Roxana, por acompañarme estos últimos años, y decirles que sin ellas mi vida no sería la misma.

A mi familia, por educarme y darme la oportunidad de estudiar.

A Luján, Wenddy y Osvaldo, por compartir tantas tardes de estudio y por estar conmigo en los momentos en que los necesité.

Y por último a Germán, mi amor, por estar a mi lado siempre, y a mi solcito Lautaro por sonreírme y hacerme sentir la persona más feliz del mundo.

INDICE

Índice de figuras

1 Introducción

En este capítulo se describen los problemas principales encontrados en los desarrollos actuales de sistemas de software, y se presenta un nuevo paradigma que promete solucionarlos. Al final del capítulo se presentan los objetivos de este trabajo.

1.1 El desarrollo tradicional de desarrollo de *software*

El desarrollo de software se suele comparar, en términos de madurez, con el desarrollo del hardware. Mientras que el desarrollo de hardware tuvo importantes avances, (por ejemplo, la velocidad de los procesadores ha crecido exponencialmente en veinte años), el progreso en el desarrollo de software parece ser mínimo. El progreso hecho en el desarrollo del software no se puede medir en términos de la velocidad o de los costos del mismo. Este progreso se hace evidente en el hecho de que es posible construir sistemas mucho más complejos y mucho más grandes que en un comienzo.

El desarrollo del software se encuentra actualmente en una etapa en la que debe superar un importante número de problemas. Hoy en día, debido a la gran cantidad de tecnologías existentes, escribir software es una tarea difícil. Con cada nueva tecnología se requiere mucho trabajo para poder emplearla teniendo en cuenta el tiempo de aprendizaje, el tiempo de desarrollo, etc. Y este trabajo debe ser hecho una y otra vez por cada una de estas nuevas tecnologías. Por otro lado, la mayoría de los sistemas actuales usan más de una tecnología, y necesitan comunicarse con otros sistemas. También hay que tener en cuenta que los requisitos cambian continuamente.

A continuación, se analizan algunos de los problemas más importantes encontrados durante el desarrollo del software y se intenta descubrir su causa. En el siguiente capítulo se presenta una propuesta de la OMG [25] que intenta solucionar los problemas planteados.

1.1.1 El problema de la productividad

El proceso del desarrollo de software, como se utiliza hoy en día, es conducido por el diseño de bajo nivel y por el código. Un proceso típico es el que se muestra en la figura 1-1, el cual incluye cinco fases:

- La conceptualización y la determinación de los requisitos del usuario
- Análisis y descripción funcional
- Diseño
- *Testing*
- *Deployment*

No importa si el proceso es iterativo e incremental, o es el tradicional en cascada. Los documentos y los diagramas se producen sólo en las primeras fases (de la uno a la tres). Estos documentos incluyen descripciones de los

requisitos escritos en lenguaje natural, dibujos, y a menudo incluyen diagramas expresados en UML [3, 29, 31, 32]. Entre estos últimos, están los diagramas de casos de uso, diagramas de clase, diagramas de interacción y diagramas de actividad.

Estos documentos y diagramas creados en las primeras tres fases rápidamente pierden el valor, tan pronto como comienza la codificación. La conexión entre los diagramas y el código se pierde gradualmente mientras se progresa en la fase de codificación. Hasta el punto en que se convierten en dibujos con poca o ninguna relación, en vez de ser una especificación exacta del código.

Cuando el sistema cambia, la relación entre el código, el texto y los diagramas producidos en las primeras tres fases se pierde. Como puede verse también en la figura 1-1, los programadores suelen hacer los cambios sólo en el código, porque no hay tiempo disponible para actualizar los diagramas y otros documentos de alto nivel. Además, el valor agregado de diagramas actualizados y los documentos es cuestionable, porque cualquier nuevo cambio se encuentra en el código de todos modos. ¿Entonces porqué utilizar tiempo valioso en construir especificaciones de alto nivel?

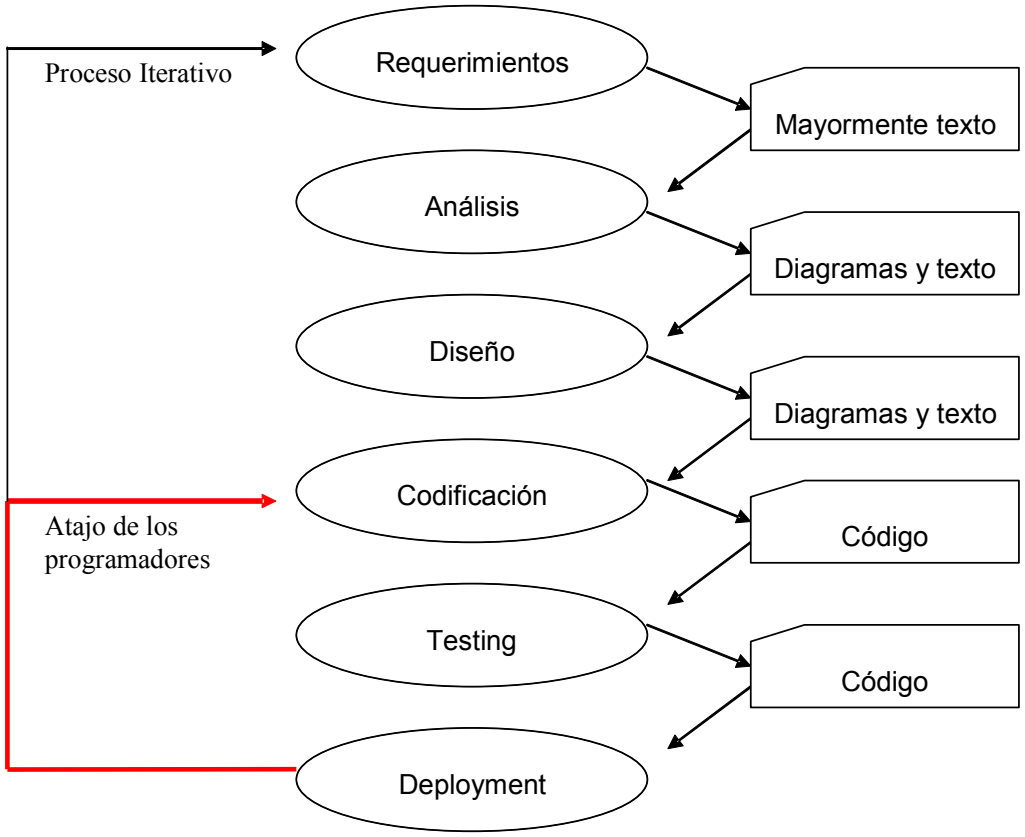


Figura 1-1 – Ciclo de vida tradicional del desarrollo de *software*

De estos problemas surgen varias soluciones. Una de ellas es la idea de *Extreme Programming* [15] (XP) que se volvió popular en muy poco tiempo. Una de las razones de esto es que reconoce al código como la fuerza

impulsora del desarrollo del software. Afirma que las únicas fases en el proceso del desarrollo que son realmente productivas son la codificación y el *testing*.

Acerca de este tema, Alistair Cockburn [6] escribió en su libro de desarrollo ágil del software, que el acercamiento de XP soluciona solamente una parte del problema. Mientras que un mismo equipo trabaja en el desarrollo de un software, existe bastante conocimiento de alto nivel en sus cabezas para entender el sistema. Y esto pasa durante el desarrollo inicial. Pero los problemas empiezan cuando se cambia parte del equipo, que sucede generalmente después de la entrega de la primera versión del software. El nuevo equipo necesita mantener, arreglar errores, mejorar o cambiar funcionalidades. Tener solamente el código hace la tarea de mantenimiento del sistema mucho más difícil. ¿Dadas quinientas mil líneas de código (o aún mucho más), dónde se comienza a intentar y a entender cómo trabaja un sistema?

Planteado así, quedan dos alternativas: o utilizamos nuestro tiempo en las primeras fases del desarrollo del software construyendo la documentación y diagramas de alto nivel, o utilizamos nuestro tiempo en la fase de mantenimiento descubriendo lo que hace realmente el software. Muchas veces se considera la tarea de la documentación como tareas adicionales. Se considera que escribir código es productivo, pero hacer modelos y documentación no lo es. No obstante, en un proyecto maduro de software estas tareas deben realizarse.

1.1.2 El problema de la portabilidad

La industria del software tiene una característica especial que la diferencia de las otras industrias. Cada año (o en menos tiempo), aparecen nuevas tecnologías que rápidamente llegan a ser populares (a modo de ejemplo, se pueden listar *Java*, *Linux*, *XML, HTML, UML, .NET, JSP, ASP, PHP, flash*[1], servicios de *Web*, etc.). Y muchas compañías necesitan aplicar estas nuevas tecnologías por buenas razones:

- Los clientes exigen esta nueva tecnología (por ejemplo, implementaciones para web).
- Son la solución para algunos problemas reales (por ejemplo, XML para el problema de intercambio de datos o Java para el problema de portabilidad).
- La empresa que desarrolla la herramienta deja de dar soporte a las viejas tecnologías y se centra sólo en las nuevas (Por ejemplo, el soporte para OMT fue reemplazado por soporte para UML).

Las nuevas tecnologías ofrecen beneficios concretos para las compañías y muchas de ellas no pueden quedarse atrás. Por lo tanto, tienen que pasar a utilizarlas los más rápido posible. Como consecuencia del cambio, las inversiones en tecnologías anteriores pierden valor. Por consiguiente, el software existente se cambia a una nueva tecnología, o a una versión nueva de la usada en su construcción. También puede darse otro caso en donde el software puede seguir sin cambiar utilizando la vieja tecnología, pero que

[1] Estos estándares están descriptos en el **Glosario de términos**.

ahora necesite comunicarse con sistemas nuevos, los cuales serán construidos usando nuevas tecnologías.

1.1.3 El problema de la interoperabilidad

Los sistemas de software raramente se encuentran aislados. La mayoría de ellos necesitan comunicarse con otros, a menudo ya existentes. Incluso cuando los sistemas se construyen desde cero, se usan múltiples tecnologías, a veces mezclando viejas y nuevas. Por ejemplo, un sistema web necesita usar una base de datos para almacenar información.

1.1.4 El problema del mantenimiento y la documentación

En las secciones anteriores, se vio el problema del mantenimiento. La documentación fue siempre un punto débil en el proceso del desarrollo del software. La sensación de la mayoría de los desarrolladores es pensar que su tarea principal es producir código. La documentación durante el desarrollo cuesta tiempo y retrasa el proceso. Se cree que documentar es una tarea aburrida que hay que hacer cuando el proyecto está prácticamente terminado y que debe ser realizada por trabajadores de bajo nivel en la organización

Pero como se explicó, la documentación ayudará la tarea a las personas que se integren al proyecto en el futuro. Así pues, escribirla se ve como algo para el futuro, no útil en el presente. Por otro lado, no hay incentivo para hacer documentación con excepción del líder de proyecto, que indica que la misma debe hacerse.

Por supuesto, los desarrolladores están equivocados. No comprenden que documentar y modelar es tal vez una de las tareas más difíciles dentro del proceso de desarrollo de software, que requiere habilidades de muy alto nivel y que debe acompañar a todo el proceso desde su comienzo. Su tarea es desarrollar sistemas que puedan cambiar y que se puedan mantener fácilmente en el futuro. A pesar de las sensaciones de muchos desarrolladores, escribir documentación es una de sus tareas esenciales.

Una solución a este problema, basado en el código, es la facilidad de generar documentación directamente desde el código fuente, asegurándose de que esté siempre actualizada. La documentación es parte del código y no algo independiente. Esta solución, sin embargo, resuelve únicamente el problema de la documentación en niveles inferiores. En cambio, la de alto nivel (tanto textos como diagramas) todavía necesitan ser mantenidos a mano. Dada la complejidad de los sistemas que se construyen, la documentación en un nivel más alto de la abstracción es imprescindible.

1.1.5 MDD – una propuesta de solución

A lo largo de esta década el Desarrollo Dirigido por Modelos (en inglés: MDD – Model Driven Development) se convirtió en un nuevo paradigma de software. MDD promete mejorar el proceso de construcción de software basándose en un proceso guiado por modelos y soportado por potentes herramientas. La transformación entre modelos constituye el motor del MDD. Estos modelos son sucesivamente transformados, comenzando con modelos abstractos independientes de la plataforma, con el objetivo de obtener a cada paso modelos más específicos, hasta llegar al código ejecutable.

De esta manera, MDD promete solucionar los problemas planteados anteriormente y mejorar la productividad y la calidad del software generado, debido a que se reduce el salto semántico entre el dominio del problema y de la solución. Se reducen los tiempos de desarrollo y las herramientas de generación pueden aplicar frameworks, patrones y técnicas con éxito ya comprobado.

1.2 Objetivos

La presente tesis tiene como objetivo principal hacer un aporte a la propuesta planteada por MDD.
Así, el primer objetivo es analizar el concepto de transformaciones de modelos; los elementos que intervienen en ellas y su modelado. A partir de ahí se hace necesario plantear el segundo objetivo, que es analizar como se da soporte a MDD en las herramientas de ingeniería de software actuales. Por lo tanto, se realiza el estudio de lenguajes y herramientas existentes para transformación de modelos. Para cada herramienta estudiada se presentan las ventajas y desventajas de su uso.
Finalmente, el tercer objetivo es hacer un aporte a las herramientas estudiadas para facilitar su utilización. Para lograrlo, se presenta la implementación de un plugin que es capaz de interoperar con las distintas herramientas ya que utiliza el estándar XMI para guardar la información. Este plugin permite realizar las definiciones de los metamodelos y modelos necesarios para las transformaciones gráficamente. Además permite chequear la buena formación de los metamodelos definidos. Por último, contribuye también a disminuir la dificultad encontrada en la utilización de la mayoría de las herramientas de transformación.

1.3 Publicaciones

Algunos de los artículos publicados en congresos, relacionados con el tema de este trabajo de tesis son:

A Minimal OCL-based Profile for Model Transformation.
Roxana Giandini, Gabriela Pérez, Claudia Pons.
Publicado en las actas de las VI Jornadas Iberoamericanas de Ingeniería de software e Ingeniería del Conocimiento (JIISIC 07) ISBN: 978-9972-2885-2-4. Lima, Perú, Febrero de 2007.

PAMPERO: Precise Assistant for the Modeling Process in an Environment with Refinement Orientation.
C. Pons, R.Giandini, G. Pérez, P. Pesce, V.Becker, J. Longinotti, J.Cengia.
UML Modeling Languages and Applications: UML 2004 Satellite Activities, Revised Selected Papers. Lecture Notes in ComputerScience number 3297. New York: Springer-Verlag. Lisbon, Portugal, October 11-15, 2004 . ISBN: 3-540-25081-6

Claudia Pons, Roxana Giandini, Gabriela Pérez, Gabriel Baum.
ATEM 2007 - 4th International Workshop on (Software) Language Engineering, at MoDELS 2007 Conference. Nashville, USA. October 2007

Además, Pampero fue el ganador del concurso internacional "ECI: Internacional Challenge for Eclipse" (http://www.scs.carleton.ca/~deugo/ice/) organizado por IBM, en la categoría estudiantil, en el año 2004

1.4 Organización de la tesina

El resto de este trabajo se estructura como sigue:
En el capítulo 2 se presenta el paradigma MDA [19, 20], como es su ciclo de desarrollo y como promete este paradigma resolver los problemas vistos anteriormente. Al final del capítulo se explica brevemente el concepto de transformación.

En el capítulo 3 se describe la arquitectura de cuatro capas de modelado y se analizan cada una de las capas. También se estudia la capa más abstracta, el lenguaje MOF [21] y su implementación más utilizada actualmente: Ecore. Al final del capítulo se explica claramente la necesidad del lenguaje MOF para definir metamodelos para las transformaciones de modelos.

En el capítulo 4 se estudia el lenguaje OCL [24], necesario para definir restricciones sobre metamodelos y modelos. Se define invariante, pre y post condición y se dan ejemplos de restricciones definidas en tres niveles: modelo, metamodelo y meta-metamodelo.

En el capítulo 5 se estudian distintas herramientas existentes que dan soporte a MDA. Se describen especialmente aquellas implementadas para la plataforma Eclipse [30]. Se analiza en particular las facilidades ofrecidas por las herramientas para definir los metamodelos y modelos necesarios para realizar transformaciones.

En el capítulo 6 se estudian dos de estas herramientas en más detalle: ATL [2, 10, 14] y MofScript [23], y se enumeran los pasos necesarios para ejecutar una transformación con ellas.

En el capítulo 7 se presenta ePlatero [8], una herramienta de modelado con base formal, implementada en esta facultad por el grupo de investigación del que formo parte. Seguidamente se extiende ePlatero con la creación de un plugin que le permite interoperar con algunas de las herramientas de transformación estudiadas, permitiendo dar soporte al paradigma MDA. Por último, el capítulo 8 expone las conclusiones de la tesis y los trabajos futuros.

2 MDA – Arquitectura Dirigida por Modelos

En este capítulo se explicará brevemente las características y el ciclo de desarrollo de un sistema usando MDA. Luego se exponen los beneficios que MDA provee, mostrando como se solucionan cada uno de los problemas expuestos anteriormente. Al final del capítulo se explican algunos conceptos básicos de las transformaciones entre modelos.

2.1 Introducción

Como se mencionó, MDD se convirtió en un nuevo paradigma de software que promete mejorar su proceso de construcción. MDD define un proceso guiado por modelos y soportado por potentes herramientas. Su objetivo es separar la especificación de la funcionalidad del sistema de su implementación sobre una plataforma concreta. Para ello, el MDD identifica dos tipos principales de modelos: modelos con alto nivel de abstracción e independientes de cualquier tecnología de implementación, y modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica.

Han aparecido varios enfoques dentro del ámbito de MDD, pero sin duda la iniciativa más conocida y extendida es la Model Driven Architecture (MDA), presentada por el OMG en noviembre de 2000 con el objetivo de afrontar los desafíos de integración de las aplicaciones y los continuos cambios tecnológicos. MDA propone el uso de un conjunto de estándares como MOF, UML y XMI.

En la actualidad existen varias herramientas que soportan MDA. Cada herramienta incorpora un mecanismo propio para el manejo de transformaciones. Una iniciativa de OMG es la definición de un lenguaje de transformaciones estándar denominado QVT [22, 28] pero en las herramientas aún no se ha adoptado, por lo que cada una de ellas define su propio lenguaje de transformación, y solo algunos de estos se basan en QVT.

2.2 ¿Qué es MDA?

MDA es el acrónimo de *Model Driven Architecture* (Arquitectura Dirigida por Modelos), un concepto promovido (pero no creado) por la OMG, que propone basar el desarrollo de software en modelos. De esta forma, a partir de esos modelos, se podrán realizar trasformaciones que generen código u otros modelos con características de una tecnología particular (o con menor nivel de abstracción). El punto clave en MDA es la importancia de los modelos en el proceso de desarrollo de software.

¿Qué no es MDA?

Hoy MDA es uno de los tantos acrónimos de moda y como el concepto puede tender a malinterpretarse, se enumeran rápidamente algunos puntos de lo que no es MDA.

- MDA no es un proceso de desarrollo
- MDA no es una especificación
- MDA no es una implementación
- MDA no es una implementación de referencia de ningún estándar particular.
- MDA no es un concepto maduro (todavía)
- MDA no es simplemente generar código
- MDA no tiene, aún, una visión unificada en la industria.
- MDA no es una arquitectura ni un "arquitectural style o pattern"

2.3 El ciclo de vida de desarrollo en MDA

El ciclo de vida de desarrollo de software usando MDA se muestra en la figura 2-1. Este ciclo de vida no se ve muy distinto del ciclo de vida tradicional. Se identifican las mismas fases. Una de las mayores diferencias está en el tipo de los artefactos que se crean durante el proceso de desarrollo. Los artefactos son modelos formales, es decir, modelos que pueden ser comprendidos por una computadora. Los siguientes tres modelos son el corazón del MDA.

- Modelo Independiente de la plataforma: es un modelo con un alto nivel de abstracción que es independiente de cualquier tecnología o lenguaje de implementación. Se lo llama PIM o Platform Independent Model.
  Un PIM representa el modelo de procesos de negocio a ser implementado. Dentro del PIM el sistema se modela desde el punto de vista de cómo se soporta mejor al negocio, sin tener en cuenta como va a ser implementado: ignora los sistemas operativos, los lenguajes de programación, el hardware y la topología de red, etc.
- Modelo específico de la plataforma: Como siguiente paso, un PIM se transforma en uno o más PSM (Platform Specific Models). Un PIM representa la proyección de los PIMs en una plataforma específica. Un PIM puede generar múltiples PSMs, cada uno para una tecnología distinta. Generalmente, los PSMs deben colaborar entre sí para una solución completa y consistente. Por ejemplo, un PSM para JAVA contiene términos como clase, interfase, etc. Un PSM para una base de datos relacional contiene términos como tabla, columna, clave foránea, etc.
- Código: El paso final en el desarrollo es la transformación de cada PSM a código. Ya que el PSM habla de elementos del código de un lenguaje especifico, la transformación es relativamente directa.

MDA define los PIM, los PSM y el código como modelos. Además define como se relacionan entre ellos. El primer paso es crear un PIM y luego transformarlo. A medida que este PIM es transformado, los nuevos modelos tendrán menos abstracción y eventualmente se obtendrá código ejecutable. Uno de los pasos más complejos en el proceso de desarrollo de MDA es la transformación de los PIM a uno o más PSMs.

2.4 Automatización de los pasos de transformación

Como ya se mencionó, el proceso MDA es parecido al proceso tradicional. Sin embargo, hay una diferencia crucial. Tradicionalmente, las transformaciones de modelo a modelo, o de modelo a código son hechas mayormente con intervención humana. Muchas herramientas pueden generar código a partir de modelos, pero generalmente no van mas allá de la generación de algún esqueleto de código que luego se debe completar a mano.

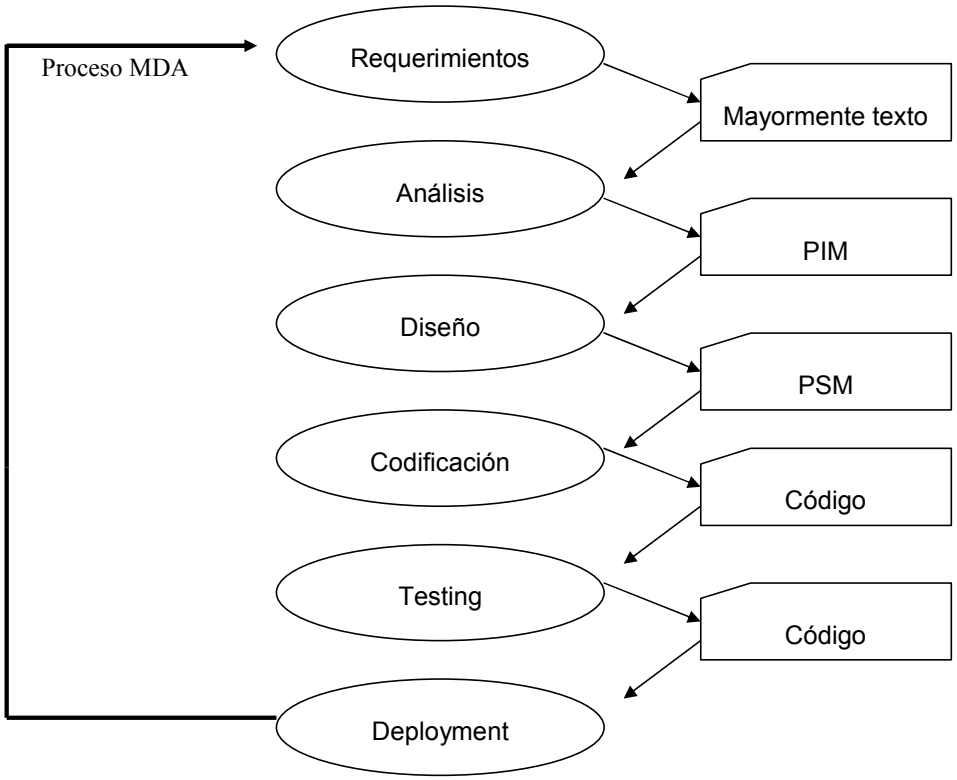


Figura 2-1 – Ciclo de vida del desarrollo de software del paradigma MDA

En contraste, las transformaciones MDA son siempre ejecutadas por herramientas, como se muestra en la figura 2-2. Muchas herramientas pueden transformar un PSM a código; no hay nada nuevo en eso. Dado que un PSM es un modelo muy cercano al código, esta transformación no es demasiado compleja. Lo nuevo que propone MDA es que la transformación de un PIM a PSMs esté automatizada. Ese es el principal beneficio de MDA.

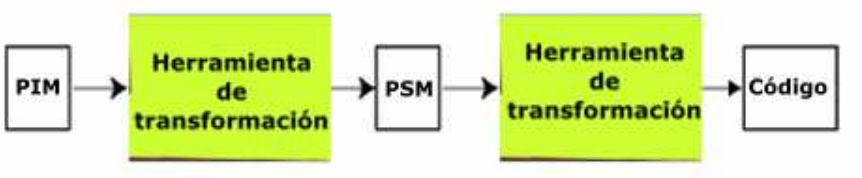


Figura 2-2 – Los tres pasos principales en el proceso de desarrollo MDA.

2.5 Beneficios de MDA

Esta sección explica los beneficios que brinda la aplicación de MDA en el proceso de desarrollo de software.

2.5.1 Productividad

En MDA el foco está en el desarrollo de un PIM. Los PSMs necesarios son generados por una transformación desde un PIM. Por supuesto, es necesario que alguien defina la transformación, que es una tarea difícil y especializada. Pero esa transformación necesita ser definida solo una vez, y puede ser aplicada en el desarrollo de muchos sistemas.

Ya que los desarrolladores necesitan enfocarse solo en los PIM, pueden trabajar independientemente de los detalles de las plataformas. Esos detalles técnicos serán agregados automáticamente por la transformación del PIM al PSM. Esto mejora la productividad de tres maneras:

- En primer lugar, los desarrolladores del PIM tienen menos trabajo que hacer, ya que los detalles específicos de la plataforma no necesitan ser diseñados, sino que ya están en la definición de la transformación. Pensando en el código, habrá mucho menos que escribir ya que una gran cantidad de ese código será generado a partir del PIM.
- En segundo lugar, los desarrolladores pueden enfocarse en el PIM en vez de hacerlo en el código. Esto hace que el sistema desarrollado se acerque más a las necesidades del usuario final, el cual tendrá mejor funcionalidad en menor tiempo.
- En tercer lugar, habrá una mejora en la implementación de modificaciones: se observa que aplicando MDA en las primeras fases del proceso las mismas son menos productivas respecto a otras metodologías (por ejemplo XP). Sin embargo esto es compensado en las etapas posteriores, especialmente en la de mantenimiento, ya que introducir modificaciones sobre los modelos es mucho más rápido y seguro que hacerlo sobre el código final.

Estas mejoras en la productividad se pueden alcanzar con el uso de herramientas que automaticen completamente la generación de un PSM a partir de un PIM.

2.5.2 Portabilidad

Dentro de MDA la portabilidad se lleva a cabo enfocándose en el desarrollo de PIMs que son independientes de la plataforma. El mismo PIM puede ser automáticamente transformado en muchos PSMs para diferentes plataformas. Por lo tanto, todo lo que se especifica en el nivel de PIM es completamente portable, solo depende de las herramientas de transformación disponibles.

2.5.3 Interoperabilidad

Como se muestra en la figura 2-3, se pueden generar muchos PSMs a partir del mismo PIM, y estos pueden estar relacionados. En MDA estas relaciones se llaman *bridges* o puentes. Los PSMs se transformarán en múltiples PIMs, los cuales frecuentemente están relacionados.

Por ejemplo, se podría transformar el mismo PIM a dos plataformas distintas modeladas con dos PSMs. Para cada elemento del primer PSM se puede conocer de qué elemento del PIM fue generado. Desde el PIM sabemos que elementos se generan en el segundo PSM. Por lo tanto, se puede deducir como se relacionan los elementos del primer PSM con los elementos del segundo PSM. Ya que conocemos todos los detalles específicos de ambas plataformas (de los dos PSMs) podemos generar el *bridge* entre los dos PSMs.

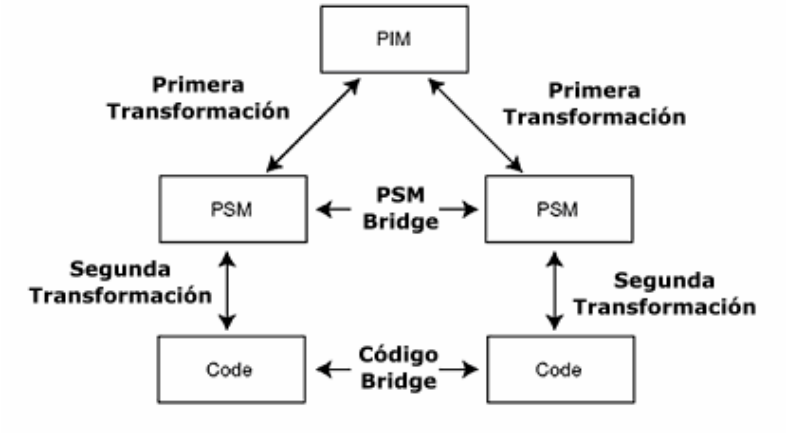


Figura 2-3 – La interoperabilidad en MDA usando puentes.

Como ejemplo, se tiene la transformación de un PIM a dos PSMs, el primero a un modelo Java y el segundo a un modelo de base de datos relacional. Para la clase "cliente" en el PIM, sabemos qué clase Java genera y qué tabla genera. Por consiguiente, construir un *bridge* entre ambos elementos es fácil. Para retornar un objeto de la base de datos, se deberían leer los datos de la tabla Cliente e instanciar un objeto de la clase Cliente.

La interoperabilidad cross-platform puede ser realizada por herramientas que generen, además de PSMs, *bridges* entre ellos y sus distintas plataformas.

2.5.4 Mantenimiento y documentación

Con MDA los desarrolladores pueden enfocarse solamente en el PIM, el cual tiene un nivel más abstracto que el código. El PIM es usado entonces para generar PSMs, el cual posteriormente será usado para generar código. Por lo tanto, el modelo será una exacta representación del código. Así el PIM cumplirá la función de documentación de alto nivel necesaria en cualquier sistema de software.

La gran diferencia es que el PIM no es abandonado luego de ser escrito. Los cambios que deben ser realizados sobre el sistema deberán ser hechos cambiando el PIM y regenerando el PSM y el código.

2.6 Conceptos básicos sobre transformación de modelos

En esta sección se explica con mas detalle que es una transformación y se da una idea de cómo se la debe especificar

2.6.1 ¿Qué es una transformación?

El proceso MDA, descrito anteriormente, muestra el rol de varios modelos, PIM, PSM y código dentro del framework MDA. Una herramienta que soporte MDA, toma un PIM como entrada y lo transforma en un PSM. La misma herramienta u otra, tomará el PSM y lo transformará a código. Estas transformaciones son esenciales en el proceso de desarrollo de MDA. En la figura 2-2 se muestra la herramienta de transformación como una caja negra, que toma un modelo de entrada y produce otro modelo como salida.

Si abriéramos la herramienta de transformación y mirásemos dentro, podríamos ver qué elementos están involucrados en la ejecución de la transformación. En algún lugar dentro de la herramienta hay una definición que describe como se debe transformar el modelo de entrada. Esta es la definición de la transformación. La figura 2-4 muestra la estructura de la herramienta de transformación.

Hay que notar que hay una diferencia entre la transformación misma, que es el proceso de generar un nuevo modelo a partir de otro modelo, y la definición de la transformación.

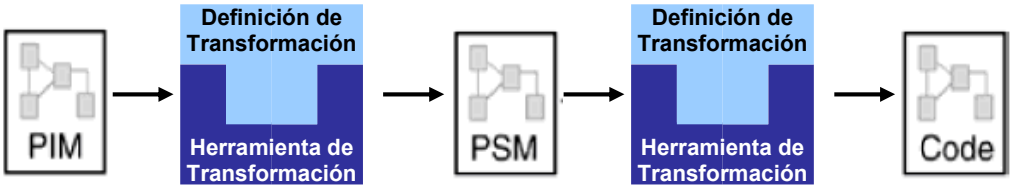


Figura 2-4 – Las definiciones de transformaciones dentro de las herramientas de transformación

Para especificar la transformación, (que será aplicada muchas veces, independientemente del modelo fuente al que será aplicada) se relacionan construcciones de un lenguaje fuente en construcciones de un lenguaje destino. Se podría, por ejemplo, definir una transformación que relaciona elementos de UML a elementos JAVA, la cual describiría como los elementos

JAVA pueden ser generados a partir de cualquier modelo UML. Esta situación se muestra en la figura 2-5.

En general, se puede decir que una definición de transformación consiste en una colección de reglas, las cuales son especificaciones no ambiguas de las formas en que un modelo (o parte de él) puede ser usado para crear otro modelo (o parte de él).

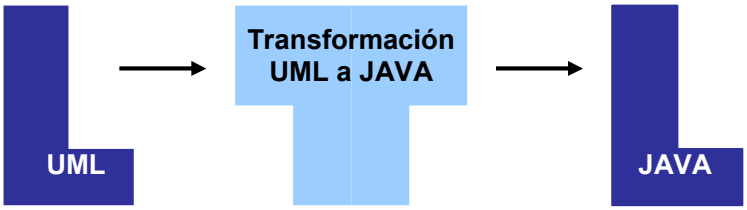


Figura 2-5 – Definición de transformaciones entre lenguajes.

Para concluir, estas definiciones fueron extraídas del libro de Anneke Kepple [16]:

- *Una transformación es la generación automática de un modelo destino desde un modelo fuente, de acuerdo a una definición de transformación.*

- *Una definición de transformación es un conjunto de reglas de transformación que juntas describen como un modelo en el lenguaje fuente puede ser transformado en un modelo en el lenguaje destino.*

- *Una regla de transformación es una descripción de como una o más construcciones en el lenguaje fuente pueden ser transformadas en una o más construcciones en el lenguaje destino.*

2.6.2 ¿Cómo se especifica una transformación?

Para especificar una transformación, la primera cuestión que se plantea es en que lenguaje se va a definir. Una primera propuesta es hacerlo en un lenguaje textual, donde la transformación estará compuesta por numerosas reglas cada una de las cuales transformará un subconjunto de elementos del modelo fuente en un subconjunto de elementos del modelo destino. Acá aparecen algunas propuestas, como QVT que intentan cubrir este tema.

Una segunda cuestión es determinar cuales son los conceptos a transformar. Por un lado, en el ejemplo se quiere transformar las clases, del primer modelo (UML), en clases del segundo modelo (JAVA). Hay que tener en cuenta que la transformación debe involucrar conceptos de ambos mundos y tratar estos elementos de manera uniforme. Para poder hacerlo, se necesita un lenguaje más abstracto aún que el lenguaje de los elementos que se quiere transformar. Y que estos lenguajes (UML y JAVA) sean instancias de este lenguaje más abstracto. Es necesario entonces, la definición de un meta-lenguaje que permita predicar sobre los elementos, y poder establecer las relaciones entre éstos. Este meta-lenguaje es MOF.

En el siguiente capítulo se presenta la arquitectura cuatro capas de modelado, la cual proporciona un marco para la definición de las

transformaciones entre modelos. En el capítulo 4 se introduce OCL, lenguaje basado en lógica de primer orden, útil para especificar reglas de buena formación sobre metamodelos y modelos.

3 La arquitectura cuatro capas de modelado

En este capítulo se explica brevemente el objetivo del metamodelado, la arquitectura cuatro capas de metamodelado definida por la OMG y por qué el metamodelado es tan importante en el contexto de MDA.

3.1 Introducción

Hace algunos años, los lenguajes se definían frecuentemente usando la gramática Backus Naur Form (BNF), en la cual se describe que secuencia de tokens forman una expresión correcta dentro del lenguaje. Este método es útil para lenguajes textuales, como lo son los lenguajes de programación. Ya que los lenguajes de modelado no tienen que estar basados en texto, y generalmente no lo están (ya que tienen una sintaxis gráfica, como UML) se necesita un mecanismo diferente para definirlos. Este mecanismo de definición se llama metamodelado.

Usando un lenguaje de modelado, podemos crear modelos; un modelo especifica que elementos pueden existir en un sistema. Si se define la clase Persona en un modelo, se pueden tener instancias de Persona como Juan, Pedro, etc. Por otro lado, la definición de un lenguaje de modelado especifica que elementos pueden existir en un modelo. Por ejemplo, el lenguaje UML especifica que dentro de un modelo se pueden usar los conceptos Clase, Estado, Paquete, etc. Debido a esta similitud, se puede describir un lenguaje por medio de un modelo, usualmente llamado metamodelo. El metamodelo de un lenguaje describe que elementos pueden ser usados en el lenguaje.

En UML se pueden usar, entre otros elementos, clases, atributos y asociaciones, ya que el metamodelo de UML define que es una clase, que es un atributo y que es una asociación.

Como un metamodelo es también un modelo, el metamodelo en sí mismo debe estar escrito en un lenguaje bien definido. Este lenguaje se llama metalenguaje. Desde este punto de vista, BNF es un metalenguaje. En la Figura 3-1 se muestra gráficamente esta relación.

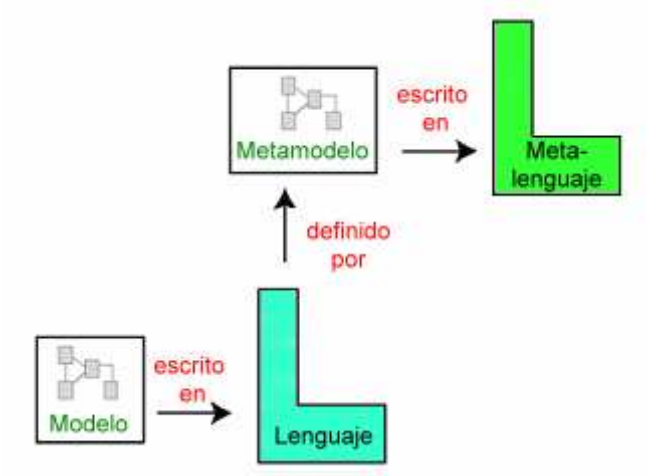


Figura 3-1 - Modelos, Lenguajes, Metamodelos y Metalenguajes

3.2 Modelos y metamodelos

El metamodelado entonces es un mecanismo que permite definir formalmente lenguajes de modelado, como por ejemplo, UML y OCL. La Arquitectura cuatro capas de Modelado es la propuesta de la OMG orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos.

Los cuatro niveles definidos en esta arquitectura se denominan comúnmente: M3, M2, M1, M0:

Nivel M0: Instancias

En el nivel M0 se encuentran todas las instancias "reales" del sistema, es decir, los objetos de la aplicación. Aquí no se habla de clases, ni atributos, sino de entidades físicas que existen en el sistema.

Para entender mejor la relación entre los elementos en las distintas capas, se presenta un ejemplo de cómo se modela un sistema con UML.

Ejemplo

Supongamos que se tiene un sistema de un videoclub, donde se maneja información acerca de los clientes, y las películas de las cuales se dispone.

En la Figura 3-2 se muestra un diagrama de objetos UML donde puede verse las distintas entidades que almacenan los datos necesarios para este sistema. Se tiene un videoclub, llamado "Videomanía", del cual se conoce el nombre y la dirección. Además, este videoclub tiene un cliente, Juan García, del cual queremos guardar su dirección y su dirección de email. Por último, el videoclub tiene una película con título "The Ring" de la cual se conoce la duración.

Todas estas entidades son instancias pertenecientes a la capa M0.

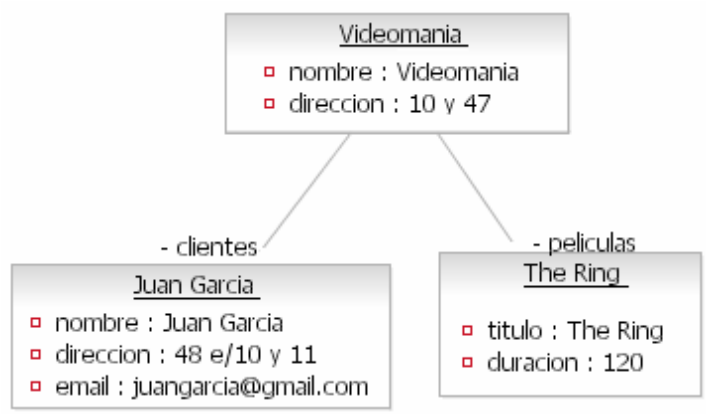


Figura 3-2 - Entidades de la capa M0 del modelo de cuatro capas

Nivel M1 : Modelo del sistema

Por encima de la capa M0 se sitúa la capa M1, que representa el modelo de un sistema de software. Los conceptos del nivel M1 representan categorías de las instancias de M0. Es decir, cada elemento de M0 es una instancia de un elemento de M1. Sus elementos son modelos de datos, por ejemplo entidades como "Cliente", "Videoclub", atributos como "nombre" y relaciones entre estas entidades.

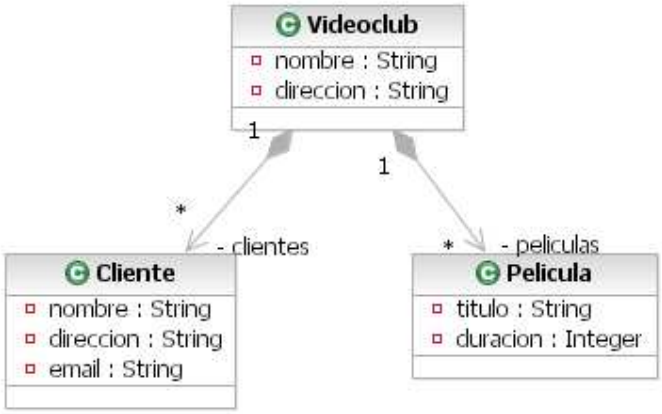


Figura 3-3 – Modelo del sistema

En el nivel M1 aparece entonces la entidad Videoclub la cual representa los videoclubs del sistema, tales como "Videomanía", con los atributos nombre y dirección. Lo mismo ocurre con la entidad Cliente y Película. En la figura 3-3 se muestra un modelo de clases UML para este ejemplo.

Videomanía puede verse ahora como una instancia de Videoclub. De la misma manera, el cliente de nombre Juan García puede verse como una instancia de la entidad Cliente y "The Ring" como una instancia de Película. En la figura 3-4 se muestra la relación entre el nivel M0 y el nivel M1.

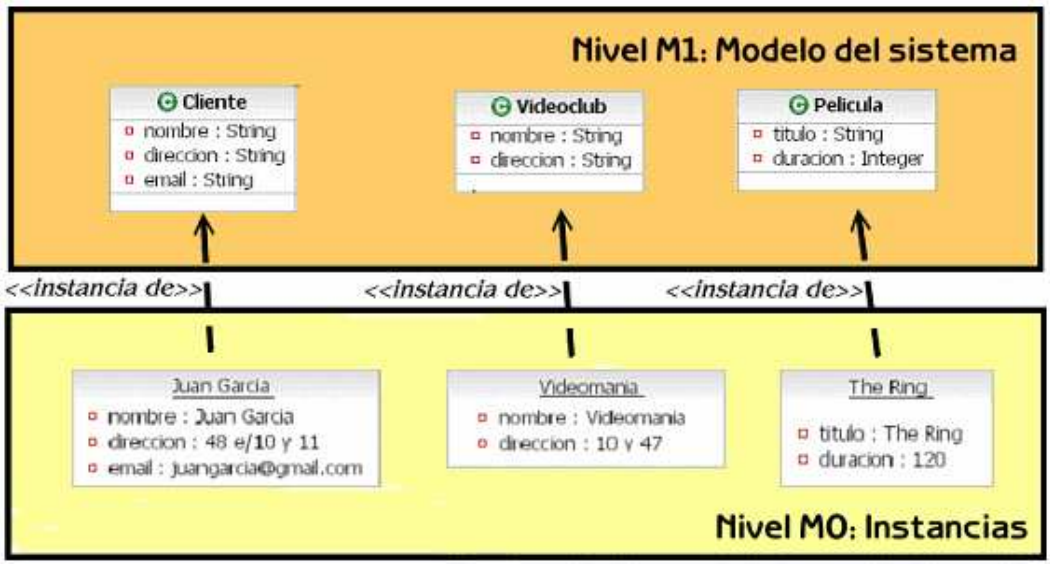


Figura 3-4- Relación entre el nivel M0 y el nivel M1

Nivel M2: Metamodelo

Análogamente a como ocurre con las capas M0 y M1, los elementos del nivel M1 son a su vez instancias del nivel M2. Esta capa recibe el nombre de metamodelo. La figura 3-5 muestra una parte del metamodelo UML. En este nivel aparecen conceptos tales como Clase, Atributo o Asociación.

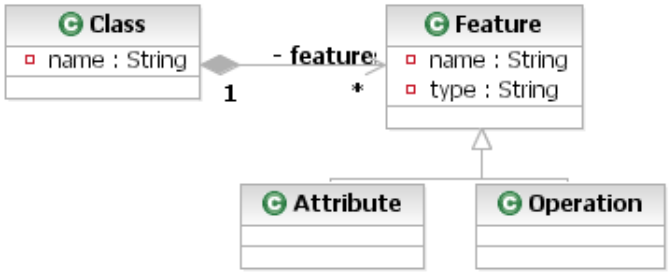


Figura 3-5 - Parte del metamodelo UML

Siguiendo el ejemplo, la entidad Cliente será una instancia de la metaclase Class del metamodelo UML. Sus atributos, nombre y dirección serán instancias de la metaclase Attribute. En la figura 3-6 se muestra los elementos del modelo Videoclub instanciados a partir de las metaclases del metamodelo UML.

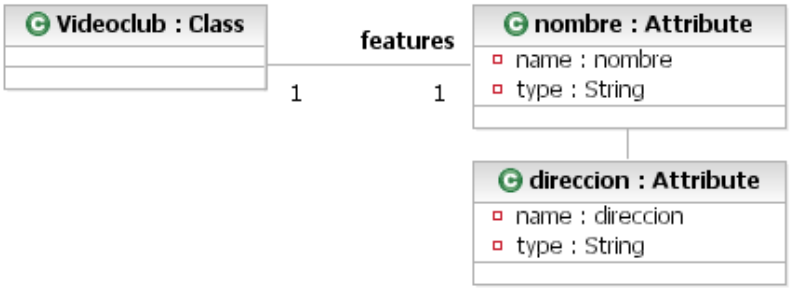


Figura 3-6 – Modelo instanciado a partir del metamodelo UML

La figura 3-7 muestra la relación entre los elementos del nivel M1 con los elementos del nivel M2.

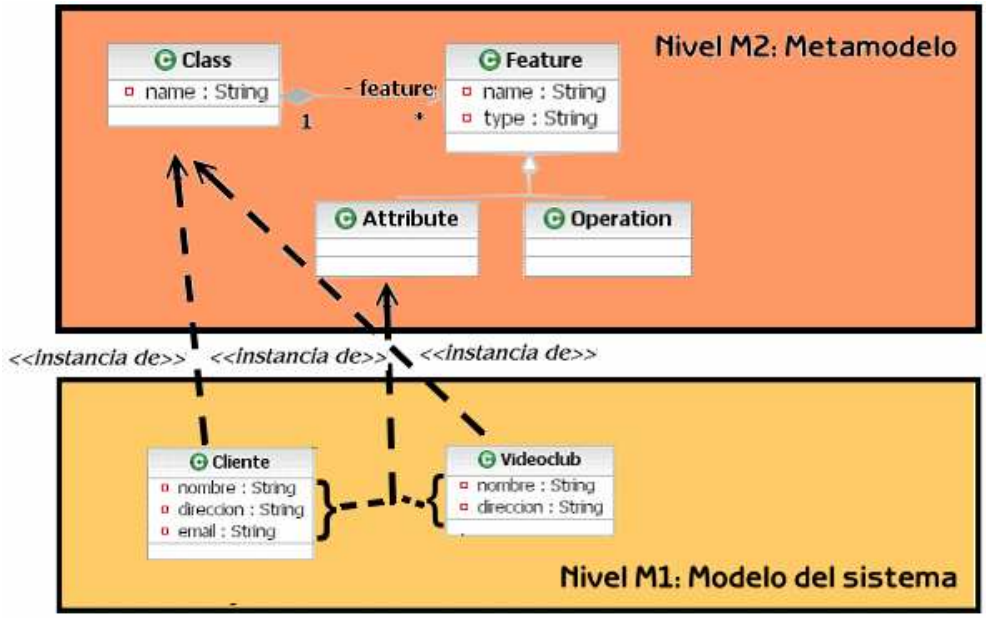


Figura 3-7 - Relación entre el nivel M1 y el nivel M2

Nivel M3: Meta-metamodelo

De la misma manera podemos ver los elementos de M2 como instancias de otra capa, la capa M3 o capa de meta-metamodelo. Un meta-metamodelo (OMG, 2003) es un modelo que define el lenguaje para representar un metamodelo. La relación entre un meta-metamodelo y un metamodelo es análoga a la relación entre un metamodelo y modelo.

Es el nivel más abstracto, que permite definir metamodelos concretos. Dentro de la OMG, MOF es el lenguaje estándar de la capa M3. Esto supone que todos los metamodelos de la capa M2, son instancias de MOF. MOF puede ser usado para definir metamodelos orientados a objetos, como es el caso de UML, y también para otros metamodelos, como es el caso de las redes de Petri o metamodelos para servicios web.

La figura 3-8 muestra la relación entre los elementos del metamodelo UML (Nivel M2) con los elementos del metamodelo de MOF (Nivel M3). Puede verse que las entidades de la capa M2 son instancias de las metaclases MOF de la capa M3

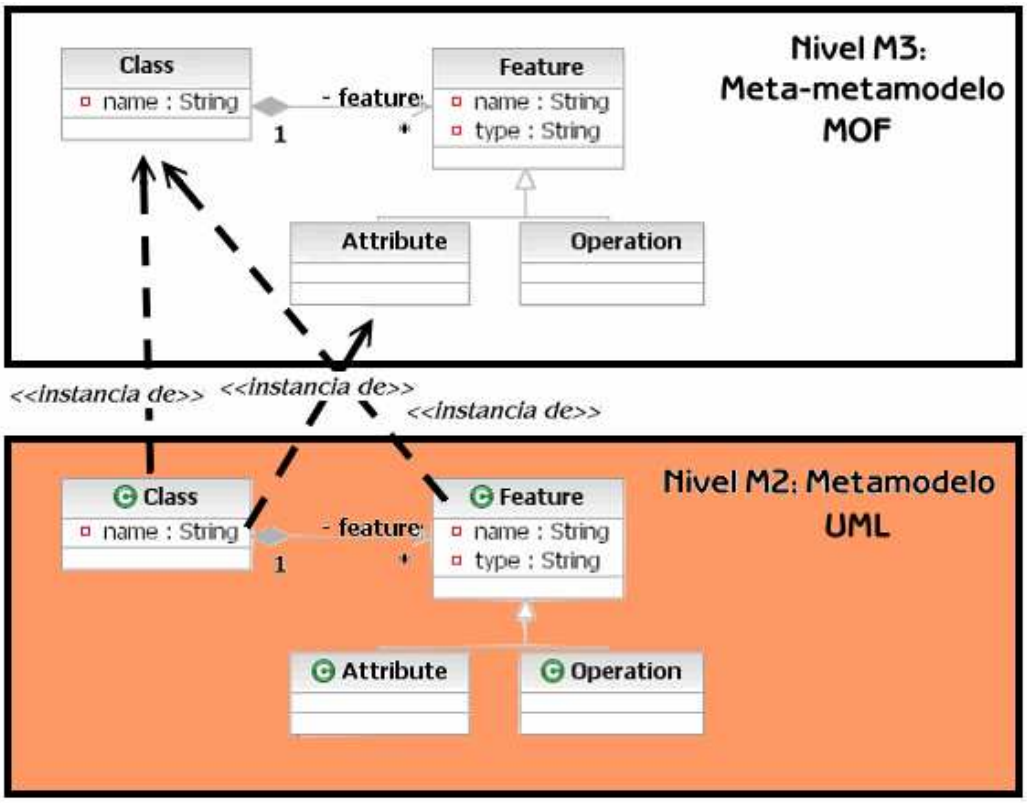


Figura 3-8 - Relación entre el nivel M2 y el nivel M3

Por último, como vista general, la figura 3-9 muestra las cuatro capas de la arquitectura de modelado, indicando las relaciones entre los elementos en las diferentes capas. Puede verse que en la capa M3 se encuentra el meta-metamodelo MOF, a partir del cual se pueden definir distintos metamodelos en el nivel M2, como UML, JAVA, OCL, etc. Instancias de estos metamodelos serán los elementos del nivel M1, como modelos UML, o modelos Java. A su vez, instancias de los elementos M1 serán los objetos, como los objetos UML.

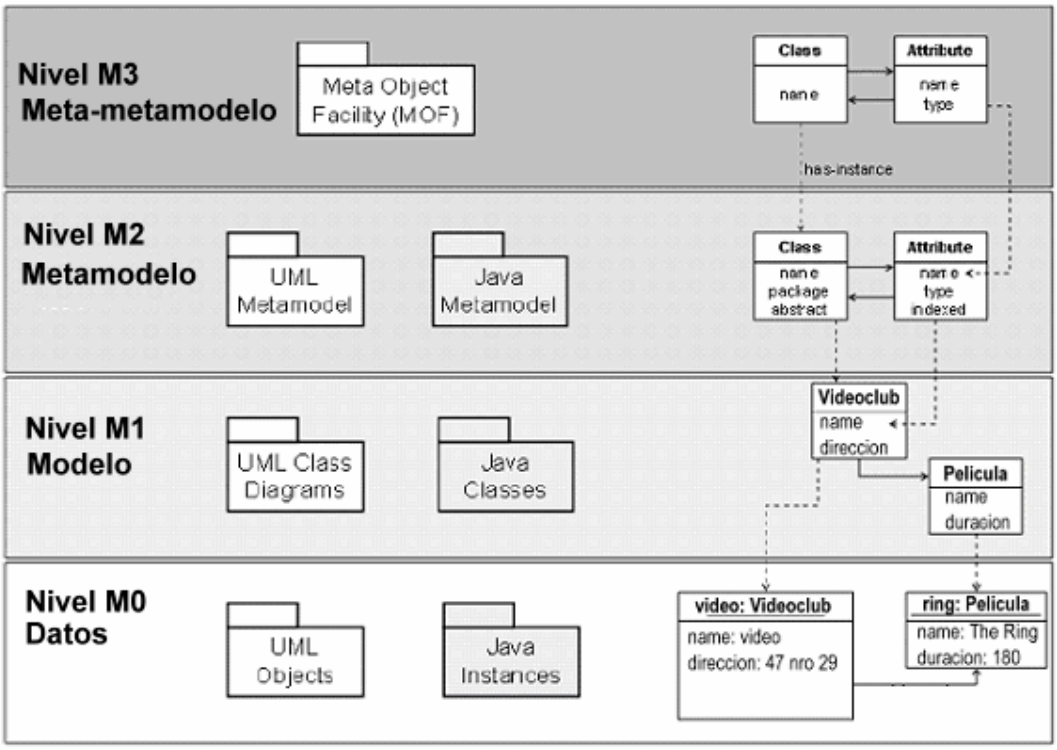


Figura 3-9 – Vista general de las relaciones entre los 4 niveles

3.3 El uso del metamodelado en MDA

La razón por la que el metamodelado es tan importante dentro de MDA es, primero, la necesidad de contar con un mecanismo para definir lenguajes de modelado sin ambigüedades y permitir que una herramienta de transformación pueda leer, escribir y entender los modelos. Además, las reglas de transformación que constituyen una definición de una transformación describen como un modelo en un lenguaje fuente puede ser transformado a un modelo en un lenguaje destino. Estas reglas usan los metamodelos de los lenguajes fuente y destino para definir la transformación.

La figura 3-10 muestra como se completa MDA con la capa de metamodelado. La parte baja de la figura es con lo que la mayoría de los desarrolladores trabaja habitualmente. En el centro se introduce el metalenguaje para definir nuevos lenguajes. Un pequeño grupo de desarrolladores, usualmente con más experiencia, necesitarán definir lenguajes y las transformaciones entre estos lenguajes. Para este grupo, entender el metanivel será esencial a la hora de definir transformaciones.

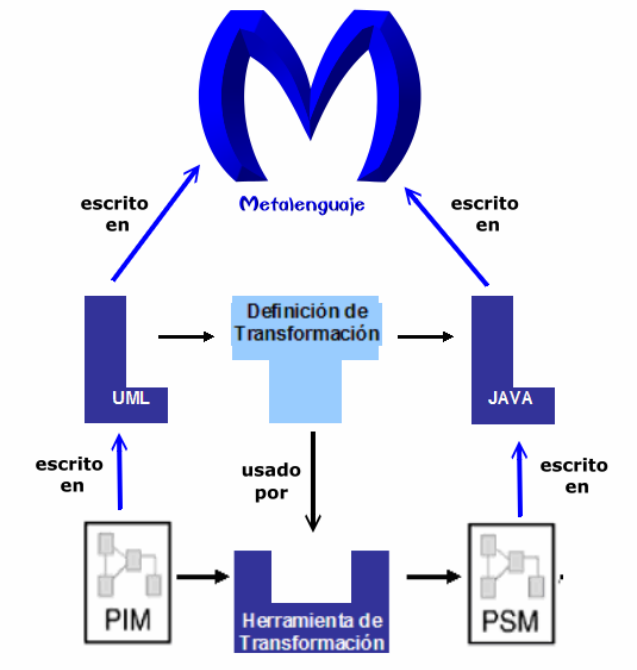


Figura 3-10 – MDA incluyendo el metalenguaje

3.4 La capa más abstracta : MOF (Meta-Object Facility)

El lenguaje MOF, acrónimo de Meta-Object Facility, es un estándar de la OMG para la ingeniería conducida por modelos. Como se vió en el capítulo anterior, MOF se encuentra en la capa superior de la arquitectura de 4 capas. Provee un meta-meta lenguaje en la capa superior que permite definir metamodelos en la capa M2. El ejemplo más popular de un lenguaje en la capa M2 es el metamodelo UML, que describe el lenguaje UML.

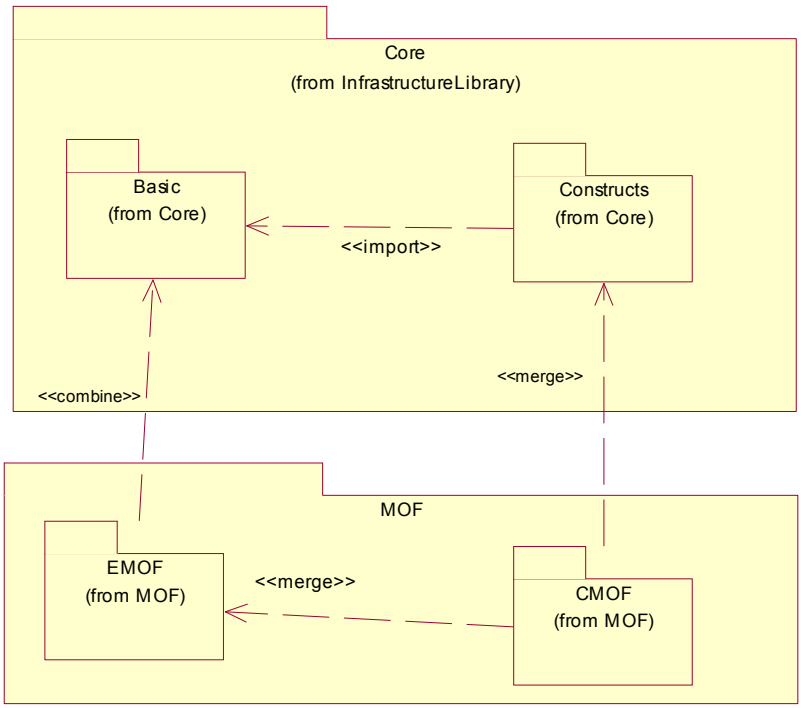


Figura 3-11 – Relación entre EMOF y CMOF

Mof es una arquitectura de metamodelado cerrada. Esto significa que el metamodelo MOF se define en términos de si mismo. MOF permite una arquitectura de metamodelado estricta, es decir, cada elemento de un modelo en cualquiera de las capas tiene una correspondencia estricta con un elemento del modelo de la capa superior.

Actualmente, la definición de MOF esta separada en dos partes fundamentales, EMOF (*Essential MOF*) y CMOF (*Complete MOF*), y se espera que en el futuro se agregue SMOF (*Semantic MOF*). En la figura 3-11 puede verse la relación entre EMOF y CMOF. Ambos paquetes importan los elementos de un paquete en común, del cual utilizan los constructores básicos y lo extienden con los elementos necesarios para definir metamodelos simples, en el caso de EMOF y metamodelos mas sofisticados, en el caso de CMOF.

3.4.1 EMOF - Essential MOF

La importancia de EMOF es la de proveer un framework para mapear modelos MOF a implementaciones como XMI para metamodelos simples. Un primer objetivo de EMOF es permitir definir metamodelos simples usando un conjunto de conceptos básicos pero soportando extensiones para definición de metamodelos más sofisticados (usando CMOF).

EMOF utiliza el paquete *Basic y Construct,* e incluye capacidades adicionales definidas en su especificación. EMOF, como todos los metamodelos en MOF esta especificado con CMOF.

La figura 3-12 muestra en un diagrama de clases una vista general de las principales clases definidas en EMOF. Puede verse que la raíz de la jerarquía es la metaclase *Element*.

La figura 3-13 muestra un diagrama de clases definiendo la metaclase Class, donde puede verse que una Class puede tener atributos, operaciones, etc.

La figura 3-14 muestra la definición de los tipos de datos (*Datatype*) y por último la figura 3-15 muestra la definición de los paquetes.

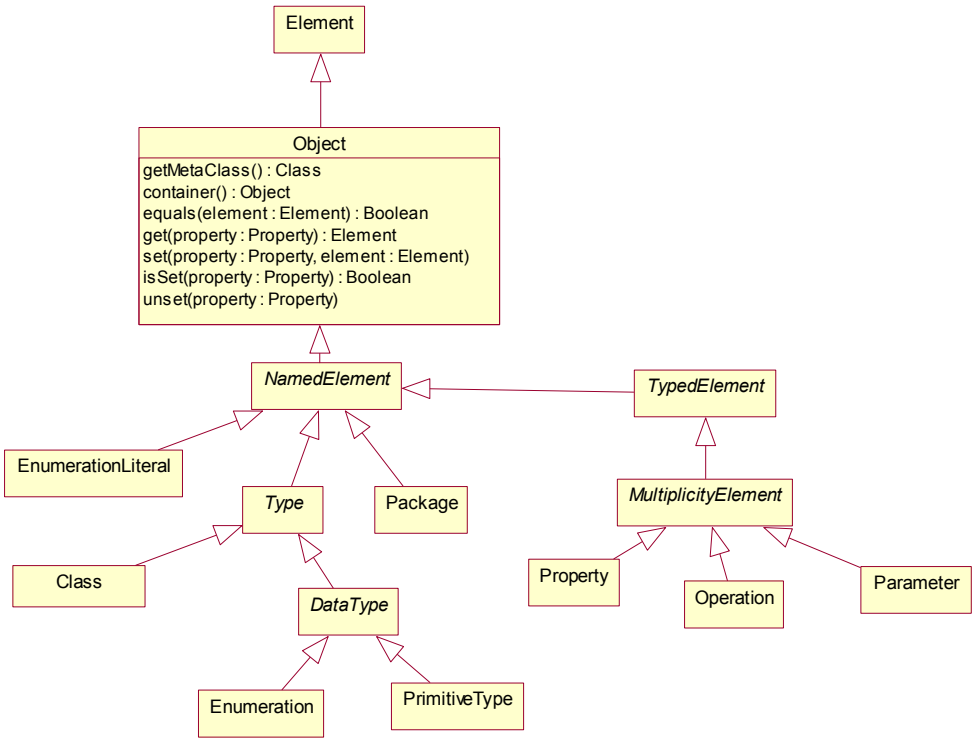


Figura 3-12 – EMOF – Overview

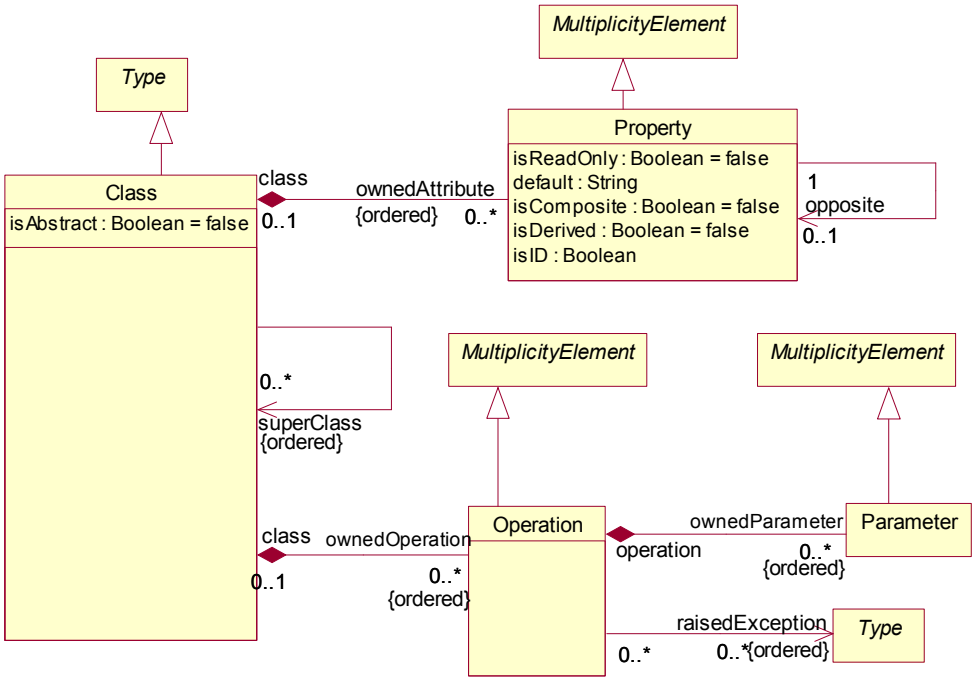


Figura 3-13 – EMOF – Diagrama para Class

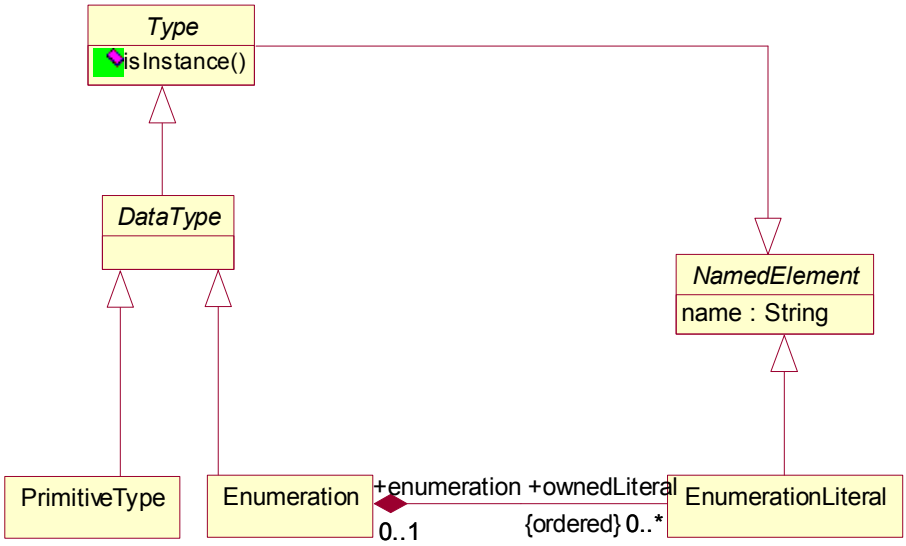


Figura 3-14 – EMOF – Diagrama para tipos de datos

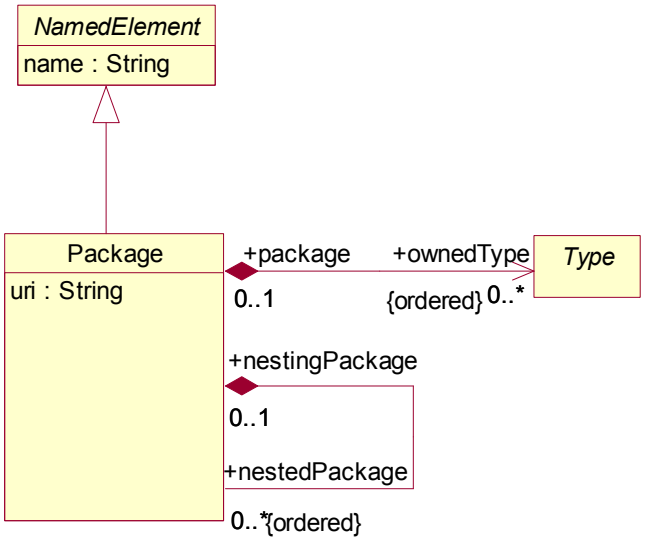


Figura 3-15 – EMOF – Diagrama para paquetes

Los siguientes ítems explican el objetivo de cada una de los elementos definidos en los diagramas de clases anteriores. Para cada uno de estos elementos se explica su descripción, los atributos que define y la semántica del mismo. Las relaciones con los otros elementos pueden verse en los diagramas de clases presentados.

- **Class**

  Una clase es la descripción de un conjunto de objetos (que son sus instancias)

  Generalizaciones:
  Type

  Atributos:

  - isAbstract: es un booleano que indica si la clase es abstracta. El valor por defecto es false.
  - ownedAttribute: Property [*]. Representa el conjunto de atributos pertenecientes a la clase. No incluye los atributos heredados. Los atributos están representados por instancias de la metaclase Property.
  - ownedOperation: Operation [*]. Representan las operaciones pertenecientes a la clase. Este conjunto no incluye las operaciones heredadas.
  - superclass: Class[*] . Es el conjunto de las superclases directas de la clase, de las cuales la clase hereda.

  Semántica:
  Las clases tienen atributos y operaciones y participan en jerarquías de herencia. La herencia múltiple esta permitida. Las instancias de

una clase son objetos. Cuando una clase es abstracta no puede tener instancias directas. Cualquier instancia de una clase no abstracta (concreta) es también una instancia indirecta de su superclase.
Un objeto permite la invocación de operaciones definidas en su clase y en su superclases.

- **Datatype**

  Datatype es una clase abstracta que actúa como una superclase común de distintos tipos de datos.

  Generalizaciones:
  Type

  Atributos:
  - No tiene atributos adicionales.

  Semántica:
  Datatype es una clase abstracta que representa la noción general de ser un tipo de datos, es decir, un tipo cuyas instancias son identificadas solo por su valor.

- **Enumeration**

  Un enumerativo define un conjunto de literales que pueden ser usados como sus valores.

  Generalizaciones:
  Datatype

  Atributos:
  - ownedLiteral: EnumerationLiteral [*] {ordered, composite}. Es una colección ordenada para los valores del enumerativo.

  Semántica:
  Un enumerativo define un conjunto finito de valores.

- **EnumerationLiteral**

  Un literal es un valor de un enumerativo.

  Generalizaciones:
  NamedElement

  Atributos:
  - enumeration: Enumeration [0..1]. Referencia al enumerativo al cual pertenece el literal.

- **Multiplicity**

  Una multiplicidad es una definición de un intervalo de enteros no negativos que comienza con un límite inferior y termina con un límite superior posiblemente infinito. Especifica la cardinalidad permitida de la instanciación del elemento.

  Generalizaciones:
  Element

  Atributos:
  - isOrdered: Boolean. Para una multiplicidad multivaluada, este atributo especifica cuando los valores en una instanciación de los elementos están ordenados. El valor por defecto es false.
  - isUnique: Boolean. Para la multiplicidad multivaluada, este atributo especifica cuando el valor en una instanciación de este elemento es única. El valor por defecto es true.
  - Lower: Integer [0..1]. Especifica el límite inferior del intervalo de multiplicidad. El valor por defecto es 1.
  - Upper: UnlimitedNatural [0..1]. Especifica el límite superior del intervalo. El valor por defecto es 1.

- **NamedElement**

  Un namedElement representa a los elementos con nombre

  Generalizaciones:
  Element

  Atributos:
  - name: String [0..1]. El nombre del elemento.

  Semántica:
  Los elementos con nombre son instancias indirectas de NamedElement. El nombre del elemento es opcional, si se especifica debe ser un string válido, incluyendo los vacíos.

- **Operation**

  Una operación es propiedad de una clase y puede ser invocada en el contexto de los objetos que son instancias de esa clase.
  Generalizaciones:
  TypedElement
  MultiplicityElement

  Atributos:
  - class: Class [0..1]. La clase a la cual pertenece la operación.
  - ownedParameter: Parameter [*] {ordered, composite }. Representa los parámetros de la operación.

- raisedException: Type [*]. Las excepciones que se declaran como posibles a ser levantadas durante la invocación de la operación.

Semántica:
Una operación pertenece a una clase. Es posible invocar una operación sobre cualquier objeto que es directa o indirectamente instancia de la clase. Dentro de la invocación, el contexto de ejecución incluye al objeto y los valores de los parámetros. El tipo de la operación, si tiene, es el del valor retornado por la operación y la multiplicidad es la del resultado. Una operación puede estar asociada con un conjunto de tipos que representan posibles excepciones que la operación puede alcanzar.

➢ Package

Un paquete es un contenedor para tipos y otros paquetes.

Generalizaciones:
NamedElement

Atributos:

- nestedPackage: Package [*] {composite}. Representa al conjunto de los paquetes contenidos en este paquete.
- nestingPackage: Package [0..1]. El paquete que contiene al paquete actual.
- ownedType: Type [*] {composite}. El conjunto de los types contenidos en este paquete.

Semántica:
Los paquetes proveen una forma de agrupar tipos y paquetes juntos, lo cual es una estrategia que puede ser muy útil para entender y manipular un modelo. Un paquete no puede estar contenido en si mismo.

➢ Parameter

Un parámetro representa un parámetro de una operación

Generalizaciones:
NamedElement

Atributos:

- operation: Operation [0..1]. La operación a la cual pertenece el parámetro

Semántica:
Cuando se invoca una operación, se pasa por cada parámetro un argumento. Cada parámetro tiene un tipo y una multiplicidad.

- **PrimitiveType**

  Un tipo primitivo es un tipo de dato implementado por la infraestructura subyacente y esta disponible para usarlo en el modelado.

  Generalizaciones:
  Datatype

  Atributos:
  - No tiene atributos adicionales

  Semántica:
  Los tipos de datos primitivos son Integer, Boolean, String y UnlimitedNatural.

- **Property**

  Una propiedad es un elemento con tipo que representa un atributo de una clase.

  Generalizaciones:
  TypedElement
  MultiplicityElement

  Atributos:
  - class: Class [0..1]. Referencia a la clase a la cual pertenece la propiedad cuando la propiedad es un atributo.
  - default: String [0..1]. Un string para representar el valor por defecto para el atributo de un objeto cuando la clase es instanciada.
  - isComposite: Boolean. Si isComposite tiene el valor true, el objeto que contiene el atributo es contenedor para el objeto o el valor contenido en el atributo. El valor por defecto es false.
  - isDerived: Boolean. Si el atributo isDerived es true, el valor del atributo es derivado de otros atributos. El valor por defecto es false.
  - isReadOnly: Boolean. Si el atributo es true, el atributo no podrá ser escrito después de la inicialización. El valor por defecto es false.
  - opposite: Property [0..1]: dos atributos, atributo1 y atributo2 de dos objetos o1 y o2 pueden ser opuestos si o1.atributo1 referencia a o2 y o2.atributo2 referencia a o1. En este caso, atributo1 es opuesto a atributo2.

  Semántica:
  Una propiedad representa un atributo de una clase. Una propiedad tiene tipo y multiplicidad. Cuando una propiedad tiene opuesto, representa dos atributos mutuamente restringidos. La semántica de dos propiedades que son mutuamente opuestas es la misma que para asociaciones navegables bidirecionales con la excepción que la

asociación no tiene links explícitos como instancias y no tiene nombre.

- **Type**

  Un tipo restringe los valores representados por un elemento tipado (TypedElement). Un tipo sirve como restricción a un rango de valores representados por un elemento con tipo. Type es una metaclase abstracta.

  Generalizaciones:
  NamedElement

  Atributos:
  - No tiene atributos adicionales.

  Semántica:
  Un tipo representa un conjunto de valores. Un elemento tipado restringido por este tipo tiene valores dentro de este conjunto.

- **TypedElement**

  Un elemento tipado (typedElement) tiene un tipo que sirve como restricción del tipo de valores que el elemento puede representar. TypedElement es una metaclase abstracta.

  Generalizaciones:
  NamedElement

  Atributos:
  - type: Type [0..1] El tipo del TypedElement.

  Semántica:
  Los valores representados por el typedElement deben ser instancias de su tipo. Un elemento sin tipo asociado puede representar valores de cualquier tipo.

3.4.2 CMOF - Complete MOF

El modelo CMOF es el metamodelo usado para especificar otros metamodelos, como UML2. Esta construido a partir de EMOF y el paquete Core::Constructs de UML2. El paquete Model no define ninguna clase por si mismo, solo mezcla y extiende paquetes y de esa manera define capacidades básicas de metamodelado.

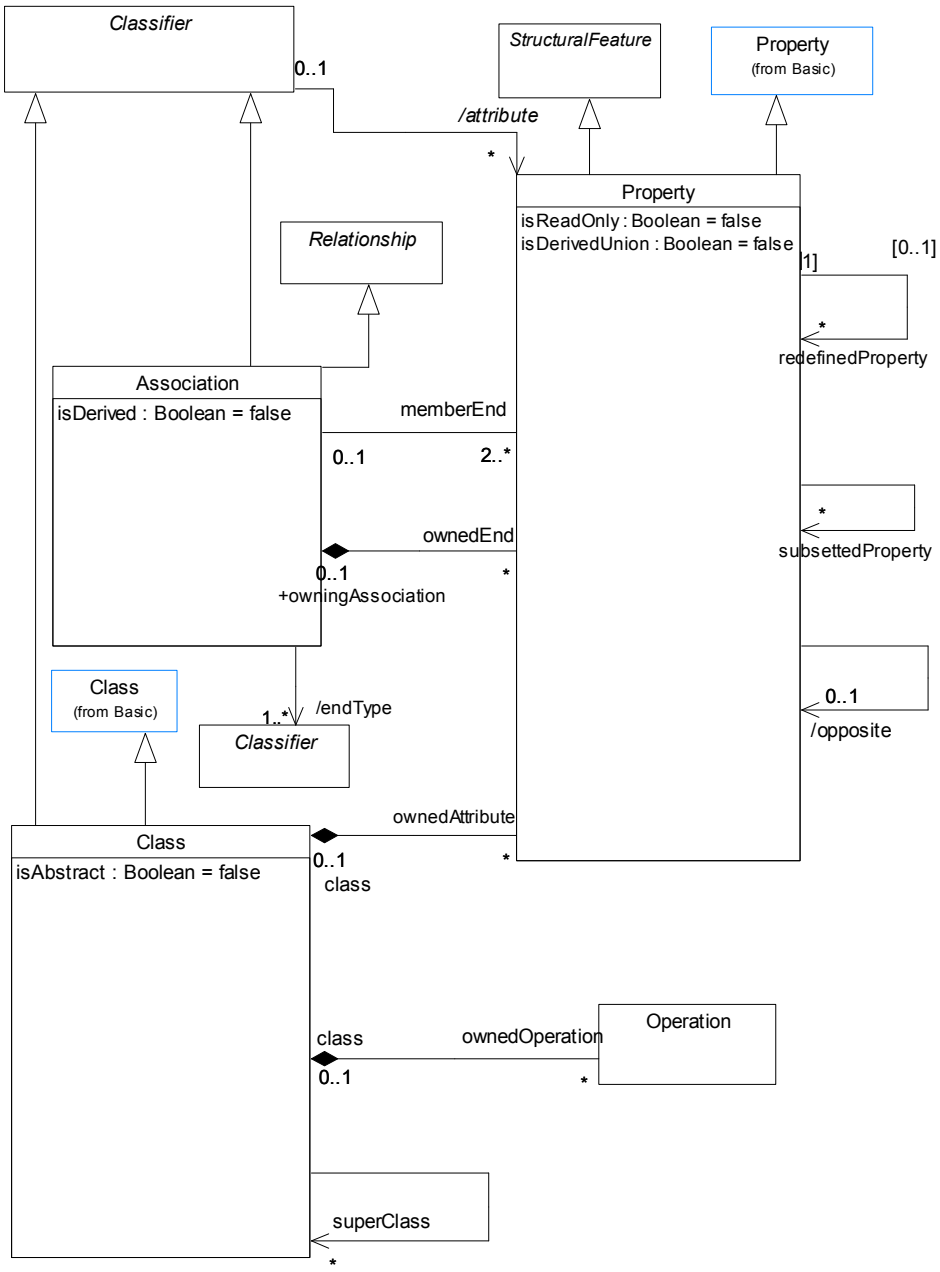


Figura 3-16 – CMOF

La figura 3-16 muestra algunas de las clases concretas y asociaciones importadas por CMOF del paquete *Basic*. Estos elementos son los básicos para proveer el modelado de clases. Este diagrama agrega la definición de asociaciones entre metaclases, y especifica que una asociación puede tener dos o más finales de asociación (*property*) que especifican las clases que se asocian. Con este diagrama se completan las principales clases de MOF,

necesarias en la definición de metamodelos simples. La definición completa de MOF se encuentra disponible en el sitio de OMG.

- **Association**

  Una asociación describe un conjunto de tuplas cuyos valores referencian a los elementos que relacionan las instancias. Una instancia de una asociación es un link. Tiene al menos dos finales de asociación, representados como propiedades, cada uno de los cuales esta relacionado con un tipo, el tipo del final de asociación. Una asociación puede tener más de un final de asociación referenciando al mismo tipo.

  Generalizaciones:
  Classifier
  Relationship

  Atributos:

  - isDerived: Boolean. Especifica cuando la asociación es derivada de otros elementos del modelo, como de otras asociaciones. El valor por defecto es false.
  - memberEnd: Property [2..*]. Cada final de asociación representa la participación de instancias del classifier conectado al final del link de la asociación.
  - ownedEnd: Property [*]. Representan los finales de asociación que son propios de la asociación. Es un conjunto ordenado.
  - / endType: Type [1..*]. Referencia los classifiers que son usados como tipos para los finales de asociación.
  - navigableOwnedEnd: Property [*]. Los finales de asociación navegables que son propios de la asociación. Es un subconjunto de ownedEnd.

  Semántica:

  Una asociación declara que pueden existir links entre instancias de los tipos asociados. Un link es una tupla con un valor para cada final de asociación, donde el valor es una instancia del tipo declarado por el final de asociación.
  Una asociación puede representar una agregación, es decir, una relación todo/parte. Solo asociaciones binarias pueden ser agregaciones. Una agregación compuesta es una forma más fuerte de agregación la cual requiere que la instancia que es parte este incluida en a lo sumo un elemento compuesto por vez. Si el elemento compuesto es borrado, normalmente todas sus partes deben ser borradas con él.

- **Classifier**

  Un classifier es una clasificación de instancias, describe un conjunto de instancias que tienen características en común. Un classifier es un espacio de nombres o namespace cuyos miembros pueden incluir features. Classifier es una metaclase abstracta.

Generalizaciones:
Namespace

Atributos:

- / feature: Feature [*]. Es un atributo derivado. Especifica cada feature definido en el classifier, los cuales pueden ser atributos u operaciones.

- **StructuralFeature**

  Un "feature de estructura" es un feature con tipo de un classifier que especifica la estructura de las instancias de ese classifier. StructuralFeature es una metaclase abstracta.

  Generalizaciones:
  TypedElement
  Feature

  Atributos:

  - No tiene atributos adicionales.

  Semántica:
  Un structuralFeature especifica que las instancias del classifier al que pertenece tienen un valor cuyo valor o valores son del tipo especificado.

- **Relationship**

  Relationship es un concepto abstracto que especifica una relación entre elementos. Una relationship referencia uno o mas elementos relacionados. Relationship es una metaclase abstracta.

  Generalizaciones:
  Element

  Atributos:

  - • / relatedElement: Element [1..*]. Especifica los elementos relacionados por la relationship

  Semántica:
  Relationship no tiene una semántica específica. Las subclases de relationship agregarán la semántica apropiada al concepto que representan.

3.5 Implementación de MOF - Ecore

El metamodelo MOF esta implementado mediante un plugin para eclipse llamado Ecore. Este plugin respeta las metaclases definidas por MOF. Todas las metaclases mantienen el nombre del elemento que implementan y

agregan como prefijo la letra "E", indicando que pertenecen al metamodelo Ecore. Por ejemplo, la metaclase EClass implementa la metaclase Class de MOF.

En las siguientes figuras pueden verse diagramas de clase que definen los elementos de Ecore. En la figura 3-17 puede verse el diagrama de clases para la definición de los *features* de estructura, como las referencias y los atributos. La figura 3-18 y 3-22 muestra la definición de las *EClass*. La figura 3-19 la definición de los atributos y la figura 3-20 la definición de las referencias. Para la definición de las operaciones y los parámetros de las operaciones se presenta la figura 3-18 y 3-21.

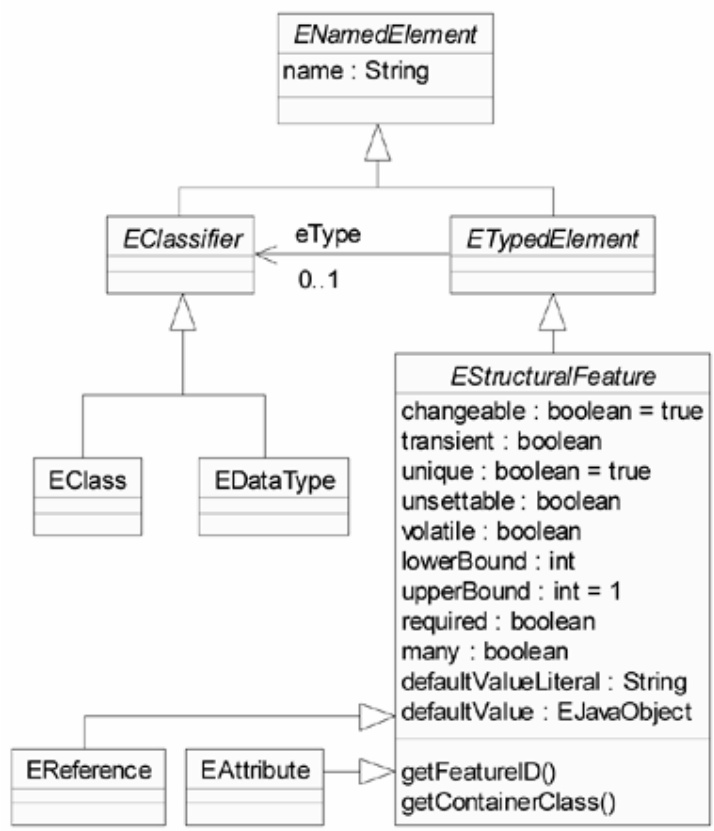


Figura 3-17 - Ecore - Features de estructura

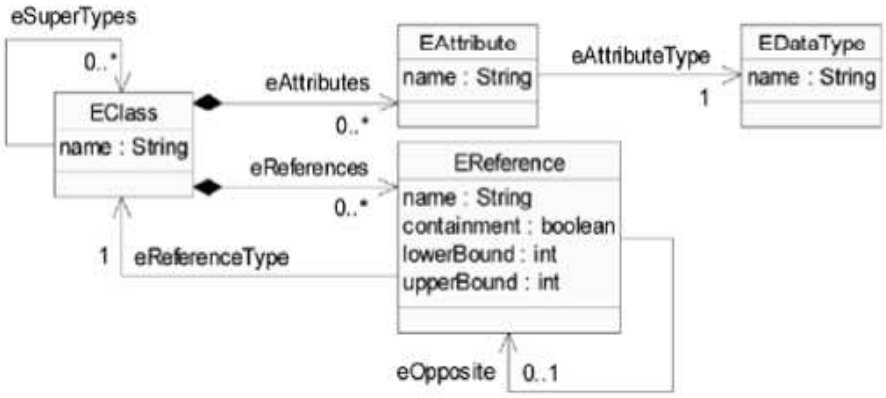


Figura 3-18 – Ecore - Kernel

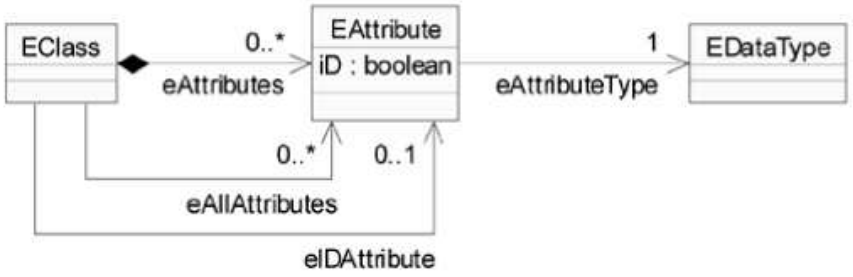


Figura 3-19 – Ecore **-** Atributos

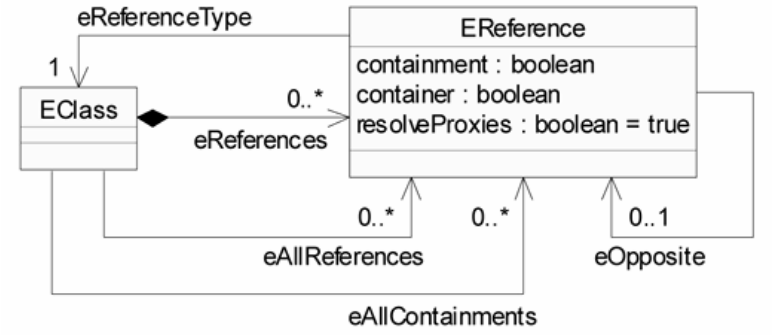


Figura 3-20 – Ecore **-** Referencias

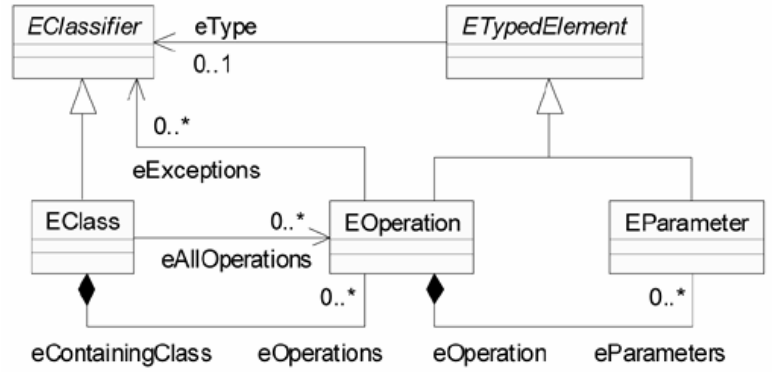


Figura 3-21 – Ecore **-** Operaciones y parámetros

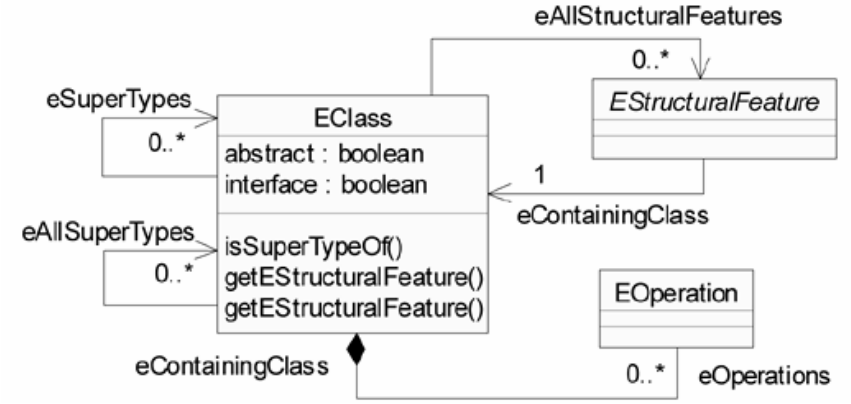


Figura 3-22 – Ecore **-** Clases

- **EAttribute**

  Un EAttribute representa un atributo de una clase.

  Generalizaciones:
  EStructuralFeature

  Atributos:
  - iD: boolean. Especifica si el valor del atributo puede ser usado para identificar inequívocamente la clase que lo contiene.

  EReferences:
  - eAttributeType: EDataType. Define el tipo del atributo.

- **EClass**

  Una EClass modela las clases. Las clases se identifican por nombre y pueden contener atributos y referencias. Para soportar la herencia, una clase puede tener otras clases como sus supertipos.

  Generalizaciones:

EClassifier

Atributos:

- abstract: boolean. Especifica si la clase es abstracta o no.
- interface: boolean. Especifica si la clase representa una interface. Declarará solo operaciones que no podrán ser implementadas.

EReferences:

- eAttributes: EAttribute [*]. Referencia a los atributos propios de la EClass.
- eAllAtributes: EAttribute[*]. Referencia a todos los atributos, los propios de la EClass y los atributos heredados por los supertipos de la EClass.
- eIDAttribute: EAttribute. Referencia al primer atributo en la lista de eAllAttributes cuyo valor en el campo iD es true.
- eReferences: EReference[*]. Referencia al conjunto de EReferences propias de la clase.
- eAllReferences: EReference[*]. Referencia al conjunto de EReferences propias de la clase y además las heredadas por las superclases.
- eAllContainmetns: EReference[*]. Referencia al conjunto de EReferences cuyo atributo containment es true.
- eOperations: EOperation[*]. Referencia al conjunto de operaciones definidas en la clase.
- eAllOperations: EOperation[*]. Referencia al conjunto de operaciones definidas en la clase más las operaciones definidas en las superclases.
- eSuperTypes: EClass[*]. Referencia al conjunto de superclases directas de la clase.
- eAllSuperTypes: EClass[*]. Referencia a todas las superclases de la clase.

➢ EClassifier

Un eClassifier es una clase abstracta.

Generalizaciones:
ENamedElement

Atributos:

- instanceClassName: String. Relaciona al Classifier con alguna clase propia de Java.
- instanceClass: EJavaClass. Relaciona al Classifier con alguna clase propia de Java.
- defaultValue: EJavaObject. Define el valor por defecto el cual debe ser un elemento de la clase de Java especificada.

- **EDataType**

  Un EDataType modela los tipos de atributos, representando los tipos de datos primitivos y los tipos de objetos que son definidos en JAVA. Los EDataType también se los identifica por nombre. Representan un dato simple.

  Generalizaciones:
  EClassifier
  Atributos:
  - serializable: boolean. Indica si el valor puede o no ser serializado.

- **EEnum**

  Un tipo de dato enumerativo es un tipo especial de datos el cual esta definido con una lista explicita de valores, llamadas literales.

  Generalizaciones:
  EDataType

  Atributos:
  - eLiterals: EEnumLiteral[*]. Referencia a la lista de literales, que son posibles valores del enumerativo.

- **EEnumLiteral**

  Representa un posible valor para un dato enumerativo

  Generalizaciones:
  ENamedElement

  Atributos:
  - value: int
  - instance: EEnumerator. Este atributo se usa para especificar el objeto estático que representa al literal.

- **ENamedElement**

  Un ENamedElement representa un elemento con nombre

  Generalizaciones:

  Atributos:
  - name:String. Representa el nombre del elemento.

- **EOperation**

  Una eOperation representa las características de comportamiento de una clase. Actualmente, las EOperation no tienen semántica, es decir, solo se modela las interfaces de las operaciones, las cuales deberán ser codificadas a mano.

  Generalizaciones:
  ETypedElement

  References:
  - eContainingClass: EClass. Referencia a la clase que define la operación.
  - eParameters: EParameter[*]. Referencia los parámetros de entrada a la operación.
  - eExceptions: EClassifier [*]. Referencia al conjunto de elementos que la operación puede alcanzar.

- **EParameter**

  Un EParameter representa un parámetro de una operación.

  Generalizaciones:
  ETypedElement

  References:
  - eOperation: EOperation. Referencia la operación a la cual pertenece el parámetro.

- **EReference**

  Una EReference es usada para modelar asociaciones entre clases. Modela un final de la asociación. Como los atributos, las referencias se identifican por nombre y tienen un tipo. Sin embargo, el tipo debe ser una EClass. Si la asociación es navegable en la dirección opuesta, debe existir otra referencia. Una referencia especifica los límites inferiores y superiores en su multiplicidad.

  Generalizaciones:

  Atributos:

  -containment: boolean. Indica que la EReference se utiliza para indicar una relación containment, es decir, una relación todo parte.

  - resolveProxies: boolean. Este atributo se utiliza para la serialización del modelo. Determina si el atributo puede ser manejado con proxies o no.

  EReferences:

  -eReferenceType: EClass. Referencia la EClass extremo de la EReference.

  -eOpposite: EReference. Especifica la EReference opuesta, para indicar una asociación bidireccional. De

esta manera, una asociación bidireccional se representa como dos EReferences entre las mismas EClasses, donde la primera EReference indica como eOpposite a la segunda, y viceversa.

- **EStructuralFeature**

  EStructuralFeature es una metaclase abstracta.
  Un feature de estructura es un feature que puede ser un atributo o una referencia.

  Generalizaciones:
  ETypedElement

  Atributos:

  - changeable: boolean. Especifica cuando el valor del feature puede ser cambiado después de la inicialización.
  - transient: boolean. Especifica cuando el feature debe o no ser serializado.
  - unique: boolean. Determina si la cardinalidad del feature es más de uno.
  - unsettable: boolean.
  - volatile: boolean. Especifica que el feature no tiene un lugar para almacenarlo ligado con este.
  - lowerBound: int. Especifica el valor inferior de la multiplicidad.
  - upperBound: int. Especifica el valor superior de la multiplicidad. Si el valor es n, se especifica con -1.
  - required: boolean. Especifica que el feature es necesario, es decir, que la multiplicidad debe ser por lo menos 1.
  - many: boolean. Especifica si la cardinalidad del feature es varios, es decir, si es mayor que 1.
  - defaultValueLiteral: String.
  - defaultValue: EJavaObject.

- **ETypedElement**

  ETypedElement es una metaclase abstracta.

  Generalizaciones:
  ENamedElement

  References:

  - eType: EClassifier. Referencia al tipo del elemento. Puede referirse a una EClass o un EDataType. Si el elemento tipado es una operación, referencia el tipo de retorno. Si en cambio es un parámetro, referencia el tipo del parámetro.

3.6 Relación entre el meta-metamodelo MOF y el meta-metamodelo Ecore

La primera implementación de Ecore fue terminada en Junio del 2002. Esta primera versión usaba como meta-metamodelo la definición estándar de MOF (v1.4).
Gracias a las sucesivas implementaciones de Ecore y basado en la experiencia ganada con el uso de esta implementación en varias herramientas, el metalenguaje evolucionó. Como conclusión, se logró una implementación de Ecore realizada en Java eficiente, con solo un subconjunto de MOF, y no con todos los elementos como manipulaban las implementaciones hechas hasta ese momento.
A partir de este conocimiento, el grupo de desarrollo de Ecore colaboró más activamente con la nueva definición de MOF, y se estableció un subconjunto de elementos como esenciales, llegando a la definición de EMOF, que fue incluida como parte del MOF (v2.0).

3.7 Diferencias entre MOF y Ecore

MOF y Ecore son conceptualmente muy similares, ambos basados en el concepto de clases con atributos tipados y operaciones con parámetros y excepciones. Además ambos soportan herencia múltiple y utilizan paquetes como mecanismo de agrupamiento.
La principal diferencia está en el tratamiento de las relaciones entre las clases. MOF tiene a la asociación como concepto primario, definiéndola como una relación binaria entre clases. Tiene finales de asociaciones, con la propiedad de navegabilidad. En cambio, Ecore define solamente EReferences, como un rol de una asociación, sin finales de asociación ni Association como metaclases. Dos EReferences pueden definirse como opuestas para establecer una relación navegable para ambos sentidos. Existen ventajas y desventajas para esta implementación. Como ventaja puede verse que las relaciones simétricas, como por ejemplo "esposoDe", implementadas con Association, son difíciles de mantener ya que debe hacerse consistentemente. En cambio con Ecore, al ser solo una referencia, ella misma es su opuesto, es decir, se captura mejor la semántica de las asociaciones simétricas, y no es necesario mantener la consistencia en el otro sentido.

Ambos, MOF y Ecore, soportan el reuso de conceptos en otros paquetes. MOF soporta varios mecanismos de reutilización de elementos como "import", herencia de paquetes, no disponibles en Ecore. Sin embargo, Ecore permite tanto la definición recursiva entre paquetes, como que sean leídos conjuntamente, mientras que MOF prohíbe las dependencias cíclicas entre paquetes debido a los problemas prácticos para mantener la consistencia mutua en un ambiente distribuido.

MOF es mucho más rico y expresivo. Por ejemplo, define un concepto para Constant, mientras en Ecore se define como un EAtributo que no puede cambiar (*unchangeable*) y solo puede ser definido dentro de una EClass. Respecto a las Constraint de MOF, Ecore no tiene una equivalente.
Para una vista completa de las diferencias entre MOF y Ecore, ver el trabajo de Anna Gerber [11].

3.8 ¿Cómo se usan los metamodelos en una transformación? Ejemplo

Para entender mejor la necesidad de los metamodelos en una transformación, se presenta como ejemplo una transformación de un modelo UML a un modelo Java.
Se quiere transformar un modelo de un videoclub escrito en UML en las clases de Java correspondientes a ese modelo.

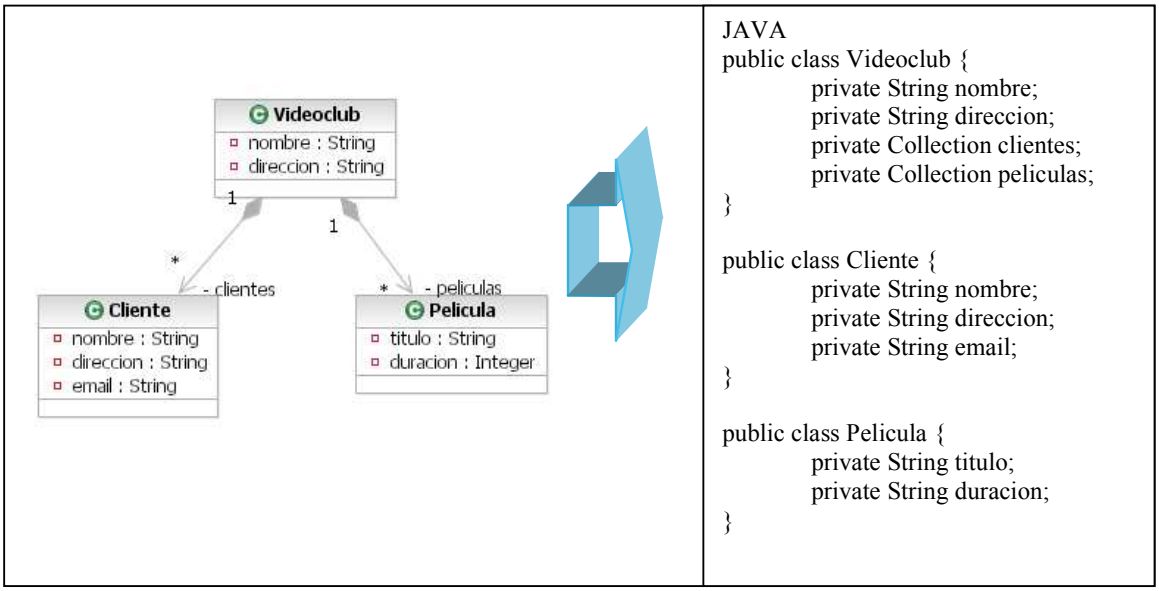


Figura 3-23 – Ejemplo de transformación

La figura 3-23 muestra gráficamente la transformación que se intenta realizar. En palabras, se quiere que la clase Videoclub de UML, se transforme en la clase Videoclub de Java. Asimismo, los atributos de Videoclub, nombre y dirección, se deben transformar en los atributos nombre y dirección de la clase Java Videoclub. Lo mismo se quiere que ocurra con las otras clases. Por lo tanto, se quiere que cada una de las clases de UML se transforme en una clase Java, y que ocurra algo similar con los atributos, los tipos de estos, y los demás elementos.

La primera cuestión que se plantea entonces, es en que lenguaje se escribirá esta transformación. Se podría escribir en un lenguaje textual, donde la transformación estará compuesta por numerosas reglas cada una de las cuales transformará un subconjunto de elementos. Acá aparecen algunas propuestas, como QVT que intentan cubrir esta cuestión.

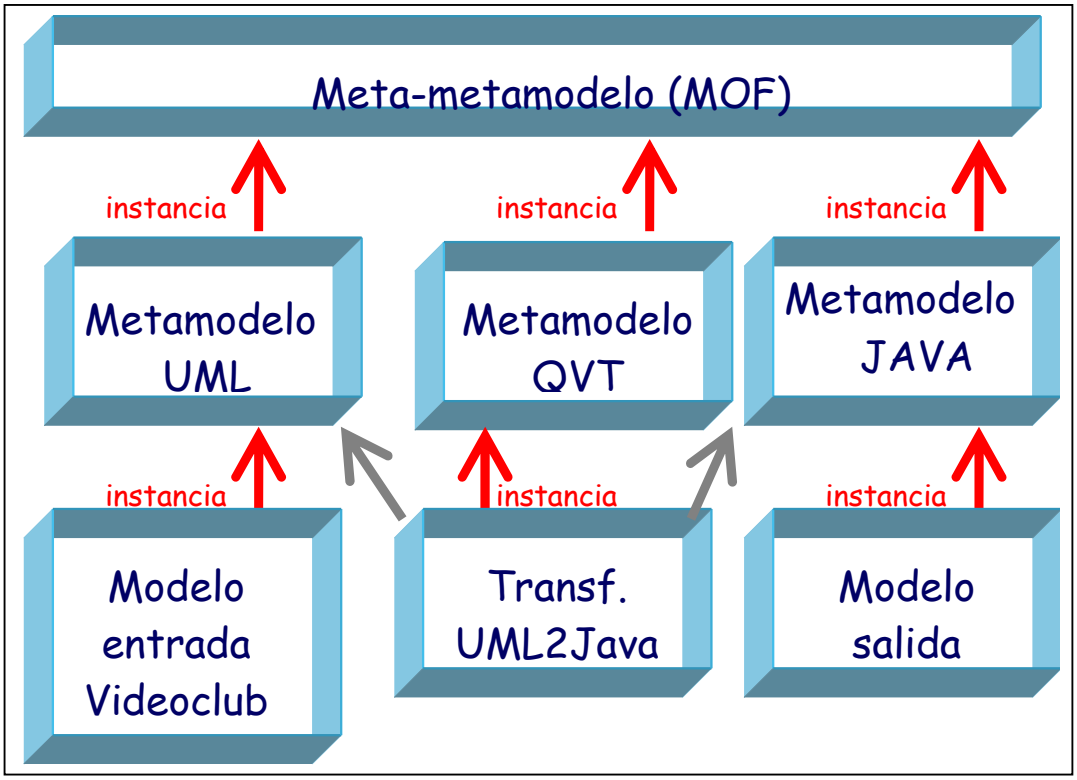


Figura 3-24 – Elementos necesarios en una transformación

Una segunda cuestión es determinar cuales son los conceptos a transformar. Por un lado, se quiere transformar las clases, del primer modelo (UML), en clases del segundo modelo (JAVA). Hay que tener en cuenta que la transformación debe involucrar conceptos de ambos mundos, y tratar estos elementos de la misma manera. Para poder hacerlo, se necesita un lenguaje más abstracto aun que el lenguaje de los elementos que se quiere transformar. Y que estos lenguajes (UML y JAVA) sean instancias de este lenguaje más abstracto. Un meta-lenguaje que permita predicar sobre los elementos y pueda establecer las relaciones entre estos. Este lenguaje es MOF.

En la figura 3-24 puede verse más claramente los elementos que están involucrados en la transformación que se intenta realizar. Para poder realizar la transformación, cada uno de estos modelos debe ser instancia de su correspondiente metamodelo, el cual como se dijo anteriormente, debe ser instancia de MOF.

Videoclub es un modelo instancia del metamodelo UML que fue a su vez definido en el lenguaje de metamodelado MOF. El modelo de salida por su parte, es una instancia del metamodelo JAVA definido también con MOF.

Siguiendo con el ejemplo, para transformar elementos de UML en elementos JAVA se necesita una transformación, o un programa de transformación, en este caso la transformación *UML2Java* que predique sobre los elementos de los respectivos metamodelos. La transformación, al ser ejecutada, teniendo como entrada al modelo Videoclub, generará automáticamente el modelo

JAVA buscado. La transformación escrita en QVT para este ejemplo, tendrá la siguiente forma:

```
Transformation UML2Java (Uml:UML2.0,Java:JAVA) {

 TopLevel Relation UMLClass2JavaClass {
    checkonly domain Uml c: Class
    enforced domain Java jc: JavaClass

    when { }
    where { c.name = jc.name and c. ownedAttribute -> forAll (
p : Property | ( jc.ownedFields -> exists(jf:JavaField |
p.name = jf.name) and jc.ownedMethods -> exists (jm1, jm2 : JavaMethod
| Attr2Getter (p, jm1) and Attr2Setter (p, jm2)) }
}
Relation Attr2Getter{
   checkonly domain Uml a: Property
   enforced domain Java jm: JavaMethod

   when { }
   where { 'get' + p.name.firstUpperCase() = jm.name }
}

Relation Attr2Setter{
   checkonly domain Uml p: Property
   enforced domain Java jm: JavaMethod

   when { ! p.isReadOnly }
   where {jm.name = 'set' + p.name.firstToUpperCase () and
jm.ownedParameter -> first().name = 'a' + p.name.firstToUpperCase () and
jm.ownedParameter-> first().type = p.type}
}

Query firstToUpperCase(string: String) : String {
 self.substring(1,1).toUpperCase()+ self.substring(2,self.size())
}

}
```

La relación UMLClass2JavaClass indica que por cada clase UML existe una clase Java con el mismo nombre, y por cada atributo de la clase UML existe un atributo con el mismo nombre, un método getter y un método setter dentro del modelo Java resultante.

Como conclusión, puede verse que las transformaciones son posibles solo mediante metamodelos compatibles con MOF, ya que esto permite que se pueda predicar de la misma forma sobre los metamodelos.

Por otro lado, los lenguajes gráficos de modelado son ampliamente aceptados en la industria, sin embargo su falta de precisión ha originado la necesidad de utilizar otros lenguajes de especificación, como OCL (Object

Constraint Language), para definir restricciones adicionales. Con el uso de OCL se ofrece al diseñador la posibilidad de crear modelos precisos y completos del sistema en etapas tempranas del desarrollo. Sin embargo para estimular su uso en la industria es necesario contar con herramientas que permitan la edición y evaluación de las especificaciones expresadas en OCL. En el siguiente capítulo se explica brevemente este lenguaje y las restricciones o propiedades que permite expresar.

4 OCL

Los lenguajes gráficos de modelado son ampliamente aceptados en la industria. Sin embargo su falta de precisión ha originado la necesidad de utilizar otros lenguajes de especificación, como OCL, para definir restricciones adicionales. Con el uso de OCL se ofrece al diseñador la posibilidad de crear modelos precisos y completos del sistema en etapas tempranas del desarrollo. Sin embargo para estimular su uso en la industria es necesario contar con herramientas que permitan la edición y evaluación de las especificaciones expresadas en OCL.

Cualquier modelo puede ser enriquecido mediante información adicional o restricciones sobre los elementos del modelo. Estas restricciones pueden ser escritas en lenguaje natural, pero en ese caso no es posible chequear su validez en el modelo y pueden ser mal interpretadas.

Un diagrama UML, como por ejemplo un diagrama de clases, en general no está lo suficientemente refinado para proveer todos los aspectos relevantes de una especificación. Existe, entre otras cosas, una necesidad de describir restricciones adicionales sobre objetos del modelo. Esas restricciones son a veces descriptas en lenguaje natural, pero en la práctica esto resulta siempre en ambigüedades. Como consecuencia, para escribir restricciones que no resulten ambiguas se han desarrollado los lenguajes formales, los cuales tienen la desventaja de tener que ser utilizados por personas con un gran conocimiento matemático, lo cual los hace difíciles de usar por los modeladores de sistemas.

OCL fue desarrollado para solucionar este problema. Es un lenguaje formal y al mismo tiempo es fácil de leer y escribir. Es un lenguaje puro de especificación, el cual garantiza estar libre de efectos laterales. Cuando se evalúa una expresión OCL, simplemente retorna un valor, sin hacer modificaciones en el modelo.

OCL no es un lenguaje de programación, es decir, no es posible escribir la lógica de un programa o flujos de control en OCL. No se pueden invocar procesos o activar operaciones que no sean de consulta con OCL.

OCL es un lenguaje tipado, por lo tanto, cada expresión OCL tiene un tipo. Para que una expresión esté bien formada, debe ajustarse a las reglas de operaciones entre tipos del lenguaje; por ejemplo, no puede compararse un Integer con un String.

Con el uso de OCL se ofrece al diseñador la posibilidad de crear modelos precisos y completos del sistema en etapas tempranas del desarrollo.

4.1 Especificación de una restricción en OCL

OCL permite especificar diferentes tipos de restricciones:

- Invariantes
- Pre y Postcondiciones

Con estas restricciones se pueden definir reglas de buena formación (Well – Formedness Rules) para el modelo.

4.1.1 Self

Cada expresión OCL esta escrita en el contexto de una instancia de un tipo especifico. En una expresión OCL la palabra reservada self es usada para referirse a esa instancia, a la cual se le especifica la restricción.

4.1.2 Invariantes

Una expresión OCL es una invariante si se estereotipa la expresión como <<invariant>>. Si una expresión OCL es una invariante para un elemento, debe evaluar verdadera para todas las instancias del elemento en todo momento. Por lo tanto, todas las expresiones OCL que denotan invariantes son del tipo Boolean.

El elemento del cual predica la expresión OCL, forma parte de la invariante y se escribe bajo la palabra clave *context* seguido del nombre del elemento. La etiqueta inv: declara que la restricción es una invariante.

Por ejemplo, para la definición de un metamodelo se podría pedir que no pueda especificarse una jerarquía múltiple. En este caso la invariante debe definirse sobre la meta-metaclase EClass de la siguiente manera:

```
context EClass inv:
self.eSuperTypes <= 1
```

En general, la sintaxis de una invariante es:

```
context [VariableName:] TypeName inv:
< OclExpression >
```

4.1.3 Pre y Postcondiciones

Una expresión OCL es una precondición o postcondición si se estereotipa la expresión como << precondition >> o << prostcondition >> respectivamente. Una pre o postcondición debe estar ligada a una operación o método. Se muestra agregando "pre" o "post" en la declaración según corresponda.

En este caso, la instancia contextual self se refiere al elemento al que pertenece la operación.

En el caso de ser una precondición, la restricción establece una condición que debe cumplirse antes de ejecutar la operación. En el caso de ser una postcondición, la restricción establece una condición que debe cumplirse después de ejecutar la operación.

En general, la sintaxis para definir pre y postcondiciones es la siguiente:

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
pre : param1 > ...
post: result = ...
```

La variable param1 se refiere a los parámetros de la operación para la cual se define la restricción. La variable result se refiere al valor de retorno de la operación.

Si una expresión OCL es definida como invariante, la restricción debe cumplirse en todo momento, en cambio en las precondiciones y postcondiciones debe cumplirse antes y después de ejecutar la operación, según sea el caso. En una expresión OCL utilizada como postcondición los elementos se pueden decorar con el postfijo "@pre" para hacer referencia al valor del elemento al comienzo de la operación.

Opcionalmente, se puede escribir el nombre de la pre o postcondición después de las palabras claves "pre" o "post", lo que permite que la restricción sea referenciada por nombre. En el siguiente ejemplo, el nombre de la precondición es parameterOK, mientras que el nombre de la postcondición es resultOK.

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
pre parameterOk: param1 > ...
post resultOk : result = ...
```

4.2 Package Context

Para especificar explícitamente en que paquete de invariantes pertenece una invariante, una pre o postcondicion, estas restricciones

deben ser encerradas entre las palabras claves 'package' y 'endpackage', con la siguiente sintaxis:

```
package Package::SubPackage

    context X inv:
    ... some invariant ...

    context X::operationName(..)
    pre: ... some precondition ...

endpackage
```

Un archivo OCL puede contener cualquier número de invariantes, pre y postcondiciones.

4.3 Reglas de buena formación a nivel modelo

Una invariante es una expresión OCL que debe ser verdadera para todas las instancias del elemento del cual predica, en cualquier momento.
Es posible especificar invariantes a nivel modelo. Estas restricciones aseguran que los objetos a ser instanciados cumplen determinadas propiedades. Estas restricciones predican entonces sobre los elementos del modelo. Por ejemplo, en el contexto del videoclub, una regla de buena formación seria la siguiente.

```
context Cliente
  inv: self.email.contains('@')
```

Self en este caso se refiere a una instancia de Cliente. Esta invariante debe cumplirse para cada uno de los clientes concretos, es decir, para cada una de las instancias de la clase Cliente. Esta invariante chequea que los emails de los clientes contengan el carácter '@'.
Puede observarse que para definir una propiedad sobre los elementos del modelo es necesario predicar sobre la clase de la cual esos elementos son instancias.

4.4 Reglas de buena formación a nivel metamodelo

De manera análoga, si quisiera definir restricciones para los modelos hay que predicar sobre las clases de las cuales los modelos son instancias. Las reglas de buena formación a nivel metamodelo predican sobre los elementos del metamodelo. Estas restricciones predican sobre los meta-elementos definidos en el metamodelo, por ejemplo, dado el metamodelo de UML, se predica sobre Class, Association, etc. Un ejemplo para esta clase de reglas es que en una

Clase no existan atributos con el mismo nombre o que una operación no tenga parámetros repetidos.

```
context Association
  inv WFR_1_Association:
    --[1] Los roles de una asociación deben tener distinto nombre
    self.memberEnd -> forAll(p,q| p.name = q.name implies p = q)
```

Self en este caso se refiere a una instancia de Association. Esta invariante debe cumplirse para cada una de las instancias de Association, es decir, para cada una de las asociaciones definidas en el modelo.
La regla anterior valida que las asociaciones tengan nombres de rol distinto en cada uno de sus extremos.

4.5 Reglas de buena formación a nivel meta-metamodelo

Si ahora quisiéramos definir un nuevo metamodelo, por ejemplo, el metamodelo relacional, y quisiéramos asegurarnos que este metamodelo este bien formado, es necesario definir reglas de buena formación sobre los meta-meta-elementos o elementos de MOF. Las reglas de buena formación a nivel meta-metamodelo son restricciones que aseguran que un metamodelo está bien formado. Un ejemplo para esta clase de reglas es que un paquete no contenga metaclases con el mismo nombre o que una metaclase no tenga meta-atributos repetidos.

```
context EPackage
  inv WFR_1_EPackage:
    --[2]The EClass must have different names.
   self.eClassifiers -> select(c | c.oclIsKindOf(EClass) )
      -> forAll (c, c2 | c.name = c2.name implies c = c2)
```

La regla anterior valida que todas las metaclases pertenecientes a un paquete tengan distinto nombre.

En el siguiente capítulo se estudian algunas de las herramientas disponibles que soportan MDA. El estudio se centra en la manera en que estas herramientas usan los estándares presentados (MOF, OCL) para la definición de los elementos necesarios para la transformación.

5 Lenguajes y herramientas para transformación de modelos

Actualmente hay una gran variedad de herramientas que soportan MDA. Las principales son:

- **Herramientas disponibles en la web-**

VIATRA [5] (VIsual Automated model TRAnsformations) framework
Epsilon
Kent Model Transformation language
Tefkat [18]
ATL (Atlas Transformation Language)
UMLX
AToM3 (A Tool for Multi-formalism and Meta-Modeling)
MOLA (MOdel transformation LAnguage)
Kermeta
MofScript

- **Herramientas comerciales—**

OptimalJ
MetaEdit+
ArcStyler,
Codagen Architect

En la presente tesis se presentan sólo algunas de estas herramientas, aquellas que están implementadas para el entorno Eclipse, ya que son las candidatas a ser enriquecidas con el editor gráfico desarrollado en este trabajo. Hay una breve explicación del resto de las herramientas en el glosario de términos.

5.1 VIATRA (VIsual Automated model TRAnsformations) framework

Viatra es una herramienta para la transformación de modelos que actualmente forma parte del framework VIATRA2, implementado en lenguaje Java y se encuentra integrado en Eclipse.
Provee un lenguaje textual para describir modelos y metamodelos, y transformaciones llamados VTML y VTCL respectivamente. La naturaleza del lenguaje es declarativa y está basada en técnicas de descripción de patrones, sin embargo es posible utilizar secciones de código imperativo. Se apoya en métodos formales como la transformación de grafos (GT) y la máquina de estados abstractos (ASM) para ser capaz de manipular modelos y realizar tareas de verificación, validación y seguridad, así como una temprana evaluación de características no funcionales como fiabilidad, disponibilidad y productividad del sistema bajo diseño.
Como puntos débiles podemos resaltar que Viatra no se basa en los estándares MOF ni QVT. No obstante, pretende soportarlos en un futuro mediante mecanismos de importación y exportación integrados en el framework.

5.2 Tefkat

Tefkat es un lenguaje de transformación de modelos y un motor para la transformación de modelos. El lenguaje esta basado en F-logic y la teoría de programas estratificados de la lógica. El motor es un plugin para Eclipse.
Tefkat fue uno de los subproyectos del proyecto Pegamento, desarrollado en Distributed Systems Technology Centre (DSTC) de Australia.
Esta implementado como un plugin de Eclipse y usa EMF para manejar los modelos basados en MOF, UML2 y esquemas XML.
Tefkat define un mapeo entre un conjunto de metamodelos origen en un conjunto de metamodelos destino. Una transformación tefkat consiste de reglas, patterns y templates. Las reglas contienen un término origen y un término destino. Los patterns son términos origen compuestos agrupados bajo un nombre, y los templates son términos destino compuestos agrupados bajo un nombre. Estos elementos están basados en la F-logic y la programación pura de la lógica, sin embargo, la ausencia de símbolos de función significa una reducción importante en la complejidad.

Tefkat define un lenguaje propio con una sintaxis concreta parecida a SQL, especialmente diseñado para escribir transformaciones reusables y escalables, usando un conceptos del dominio de alto nivel más que operar directamente con una sintaxis XML.

```
RULE ClassToTable
FORALL  Class c { name: n; }
MAKE    Table t { name: n; }
;
```

El lenguaje Tefkat esta definido en términos de EMOF (v2.0), y esta implementado en términos de Ecore. El lenguaje es muy parecido al paquete Relations de QVT.

5.3 ATL (Atlas Transformation Language)

ATL es un proyecto creado por gente de la Université de Nantes, Faculté des Sciences et Techniques, en Francia. ATL forma parte del framework de gestión de modelos AMMA que se encuentra integrado en Eclipse y EMF. Utiliza un lenguaje propietario llamado ATLAS para definir transformaciones que se ejecuta sobre JAVA para el cual proporciona un editor.
La naturaleza de ATLAS es declarativa, aunque lleva mucho tiempo utilizando también construcciones imperativas, y las transformaciones son expresadas como reglas de transformación ya que ATL cumple el estándar MOF.
ATL posee un algoritmo de ejecución preciso y determinista, sin embargo no se apoya sobre ningún método formal. No proporciona mecanismos para la validación de las transformaciones. Para llevar a cabo transformaciones compuestas se necesita ejecutar cada una de las transformaciones participantes una a una.
Aunque la sintaxis de ATL es muy similar a la de QVT, no es interoperable con este último.

5.4 EPSILON

Epsilon es una plataforma desarrollada como un conjunto de plug-ins (editores, asistentes, pantallas de configuración, etc.) sobre Eclipse. Presenta el lenguaje metamodelo independiente Epsilon Object Language que se basa en OCL. Puede ser utilizado como lenguaje de gestión de modelos o como infraestructura a extender con nuevos lenguajes específicos de dominio. Tres son los lenguajes definidos en la actualidad: Epsilon Comparison Language (ECL), Epsilon Merging Language (EML), Epsilon Transformation Language (ETL), para comparación, composición y transformación de modelos respectivamente. Se da soporte completo al estándar MOF mediante modelos EMF y documentos XML a través de JDOM.
La finalidad pretendida con la extensión del lenguaje OCL es dar soporte al acceso de múltiple modelos, ofrecer constructores de programación convencional adicionales (agrupación y secuencia de sentencias), permitir modificación de modelos, proveer depuración e informe de errores, así como conseguir una mayor uniformidad en la invocación.
Soporta mecanismos de herencia, *traceability* e introduce mecanismos de comprobación automática del resultado en composición y transformación de modelos.

5.5 AToM^3 (A Tool for Multi-formalism and Meta-Modeling)

AToM^3 es una herramienta para modelado en muchos paradigmas bajo el desarrollo de MSDL (Modelling Simulation and Design Lab) en la escuela de ciencias de la computación de la universidad de McGill.

Las dos tareas principales de AToM^3 son metamodelado y transformación de modelos. El metamodelado se refiere a la descripción o modelado de diferentes clases de formalismos usados para modelar sistemas (aunque se enfoca en formalismos de simulación y sistemas dinámicos, las capacidades de AToM^3 no se restringen a solo esos). Transformación de modelos se refiere al proceso automático de convertir, traducir o modificar un modelo en un formalismo dado en otro modelo que puede o no estar descrito en el mismo formalismo.

En AToM^3 los formalismos y modelos están descriptos como grafos, para realizar una descripción precisa y operativa de las transformaciones entre modelos.

Desde una meta especificación (en un formalismo de entidad relación), AToM^3 genera una herramienta para manipular visualmente (crear y editar) modelos descriptos en el formalismo especificado. Las transformaciones de modelos están ejecutadas por la reescritura de grafos. Las transformaciones pueden entonces ser expresadas declarativamente como modelos basados en grafos.

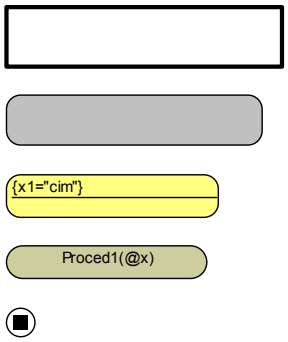
Algunos de los metamodelos actualmente disponibles son: Entidad/Relación, GPSS, autómatas finitos determinísticos, autómatas finitos no determinísticos, redes de Petri y diagramas de flujo de datos. Las transformaciones de modelos típicas incluyen la simplificación del modelo, generación de código, generación de simuladores ejecutables basados en la semántica operacional de los formalismos así como transformaciones que preservan el comportamiento entre modelos en diferentes formalismos.

5.6 MOLA (MOdel transformation LAnguage)

El proyecto Mola (http://mola.mii.lu.lv/) consiste de un lenguaje de transformación de modelos y de una herramienta para la definición y ejecución de transformaciones.
El objetivo del proyecto Mola es proveer un lenguaje gráfico para definir las transformaciones entre modelos que sea simple y fácilmente entendible.

El lenguaje para la definición de la transformación Mola es un lenguaje gráfico, basado en conceptos tradicionales como pattern matching y reglas que definen como los elementos deben ser transformados. El orden en el cual se aplican las reglas es el orden tradicional de los constructores de programación (secuencia, loop y branching).
Los procedimientos Mola definen la parte ejecutable de la transformación. La unidad principal ejecutable es la regla que contiene un pattern y acciones. Un procedimiento esta construido con reglas usando constructores

de la programación estructural tradicional, es decir, loops, branchings y llamadas a procedimientos, todos definidos de una manera grafica.
La parte ejecutable es similar a los diagramas de actividad de UML, que comienzan en un nodo inicio y contiene



loop symbols (Foreach and While)

rules

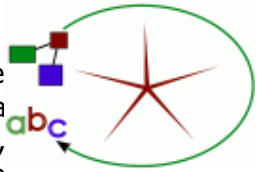
text statements

procedure calls (including external ones)

end nodes

Mola usa una manera simple para definir metamodelos: Diagramas de clases UML, los cuales consisten solo en un subconjunto de los elementos de UML, es decir, clases, asociaciones, generalizaciones y enumerativos. Solamente soporta herencia simple. Actualmente el metamodelo completo (el metamodelo fuente y el metamodelo destino) deben estar en el mismo diagrama de clases. Adicionalmente se le agregan asociaciones para mapear los elementos del metamodelo fuente en el destino.

5.7 MOFScript



La herramienta MOFScript permite la transformación de cualquier modelo MOF a texto. Es decir, permite la generación de código Java, EJB, JSP, C#, SQL Scripts, HTML o documentación a partir de los modelos. La herramienta está desarrollada como un plugin de Eclipse, el cual soporta el parseo, chequeo y ejecución de scripts escritos en MOFScript (es el lenguaje de transformación basado en QVT)

Algunas características de MOFScript son:

- Generación de texto a partir de modelos basados en MOF. La habilidad de generar texto a partir de cualquier modelo basado en un metamodelo definido en MOF, como por ejemplo, modelos UML o cualquier otro metamodelo.
- Mecanismos de control: la habilidad de especificar mecanismos de control básicos como loops y sentencias condicionales.
- Manipulación de strings con operaciones básicas.
- Permite producir recursos (archivos) como salida. Permite especificar el archivo salida para la generación de texto.
- Permite hacer *traceability* entre los modelos y el texto generado. Esta propiedad permite hacer una regeneración.
- El editor de scripts posee ayuda para completar el código.
- La ingeniería inversa todavía no es parte de la herramienta.

Los archivos que definen la transformación tienen como extensión **'.m2t'.**

El lenguaje de transformación MOFScript esta basado en el lenguaje QVT. MOFScript es un refinamiento del lenguaje operacional QVT. Es un lenguaje textual, basado en objetos y usa OCL para la navegación de los atributos. Además, presenta algunas características avanzadas, como la jerarquía de transformaciones, traceabilidad, etc.

5.8 Kermeta

El lenguaje Kermeta (http://www.kermeta.org/) fue desarrollado por un equipo de investigación de IRISA (investigadores de INRIA, CNRS, INSA y la Universidad Rennes).
El nombre Kermeta es una abreviación para "Kernel Metamodeling" y refleja el hecho que el lenguaje fue pensado como una parte fundamental para el metamodelado. La herramienta que ejecuta las transformaciones Kermeta esta desarrollada en Eclipse, bajo licencia EPL.
Kermeta es un lenguaje de modelado y de programación. Su metamodelo conforma el estándar EMOF. Fue diseñado para escribir modelos, para escribir transformaciones entre modelos y para escribir restricciones sobre estos modelos y ejecutarlos.
El objetivo de esta propuesta es brindar un nivel de abstracción adicional sobre el nivel de objetos, y de esta manera ver un sistema dado como un conjunto de conceptos (e instancias de conceptos) que forman un todo coherente que se puede llamar modelo.
Kermeta ofrece todos los conceptos de EMOF usados para la especificación de un modelo. Un concepto real de modelo, más precisamente de tipo de modelo, y una sintaxis concreta que encaja bien con la notación de modelo y metamodelo.

Kermeta ofrece dos posibilidades para escribir un metamodelo

- Escribir el metamodelo con Omondo[2], e importarlo
- Escribir el metamodelo en Kermeta y traducirlo a un archivo ecore usando la función "kermeta2ecore"

5.9 Kent Model Transformation language

El lenguaje de transformación de modelos Kent es la tercera generación de propuestas de transformación de modelos desarrolladas en la Universidad de Kent. El lenguaje intenta ser un lenguaje de especificación declarativa, con la opción de proveer partes constructivas si es requerido. La semántica del lenguaje va desde una base formal de relaciones
La semántica del lenguaje de transformación esta basada en el lenguaje Relations de QVT.
Los metamodelos son importados desde Poseidon[3], un editor de UML, donde fueron creados como diagramas de clases.

[2] Ver definición en Glosario de Términos
[3] Ver definición en Glosario de Términos

5.10 Conclusiones

Como vimos, un elemento fundamental en la transformación de modelos es la definición de los metamodelos. Muchas de las herramientas de transformación presentadas en este capítulo usan el estándar MOF para la definición de los metamodelos. Otras definen su propio lenguaje. La tabla 1 muestra, considerando los lenguajes presentados anteriormente, cuales de ellos usan lenguajes estándares para la definición de los metamodelos (es decir son compatibles con MOF), y para la definición de las transformaciones (es decir, se basan en QVT).

Tabla 1: Definición de Herramientas de Transformación

Nombre de la herramienta	Definición de los metamodelos	Lenguaje de transformación
VIATRA		
TefKat	Compatible con MOF	
ATL	Compatible con MOF	Basado en QVT
Epsilon		
AToM3		
MOLA	Compatible con MOF	
MofScript	Compatible con MOF	Basado en QVT
Kermeta	Compatible con MOF	
Kent	Compatible con MOF	Basado en QVT

Las herramientas que utilizan modelos compatibles con MOF presentan diferentes maneras para definir estos metamodelos, desde la escritura manual de un archivo en formato XMI, pasando por la definición de un nuevo lenguaje (como es el caso de ATL y el lenguaje km3) o por la importación de modelos UML desde otras herramientas (como es el caso de Kent usando Poseidon, o MOLA usando UML).
Analizando estos casos, se ve la necesidad de contar con un editor gráfico para la creación de metamodelos compatibles con MOF. Estos metamodelos podrán ser luego usados para ejecutar transformaciones en estas herramientas, facilitando el uso de las mismas.
En el próximo capítulo se muestra con más detalle, a través de casos de estudio realizados sobre dos de los lenguajes más utilizados, la dificultad en la definición de los metamodelos y la necesidad del aporte gráfico que se presenta en el capítulo 7.

6 Caso de estudio – ATL y MOFScript

En este capítulo se muestran dos casos de estudio. Uno desarrollado con ATL donde se muestra paso a paso un ejemplo sencillo de cómo ejecutar una transformación entre un modelo UML y uno relacional. El otro, desarrollado con MofScript, donde se muestra una transformación de un modelo UML a texto. En ambos casos, una de las principales dificultades en su uso es el ingreso de los metamodelos y los modelos necesarios para ejecutar la transformación querida.

6.1 Caso de estudio ATL – Transformación de un modelo UML al modelo relacional

En este caso de estudio se quiere transformar un modelo UML al modelo relacional. Para simplificar el ejemplo, solo se muestra como se transforman un subconjunto de elementos de UML a sus respectivos elementos del modelo relacional.

La transformación producirá una tabla por cada una de las clases persistentes pertenecientes al modelo de entrada. A grandes rasgos, la transformación debería generar lo siguiente:

- Por cada clase se creará una tabla con el mismo nombre
- Por cada atributo simple (es decir, no multivaluado) de la clase, se creará una columna con el nombre y el tipo del atributo
- Por cada atributo multivaluado de tipo simple (instancia del tipo Datatype) se creará una tabla con nombre nombreClase_nombreAtributo, con dos columnas: la primera, con igual nombre del la clase concatenado con el string 'ID' y la segunda con el nombre del atributo multivaluado.
- A cada tabla se le agrega una clave, llamada objectID la cual se agrega como una columna dentro de la tabla
- Por cada tipo de datos definido en el paquete PrimitiveTypes se creará un Type en el modelo relacional.
- Por cada atributo multivaluado que no sea de tipo simple, es decir, por cada asociación, se creará una tabla con el nombre de la clase origen concatenado con el nombre del atributo multivaluado. Dentro de esta tabla, se agregan dos columnas: la primera con la clave de la tabla correspondiente a la clase origen (con nombre de la clase origen concatenado con ID), y la segunda, con la clave correspondiente a la clase destino del modelo original (con nombre de la clase destino concatenado con ID).

Una vista general de la transformación está graficada en la figura 6-1, donde pueden verse los dos metamodelos (UML y Relacional) y los dos modelos (Modelo de Entrada, instancia de UML y el Modelo de Salida, instancia del metamodelo Relacional). También se muestra la transformación, instancia del lenguaje de transformación de ATL.

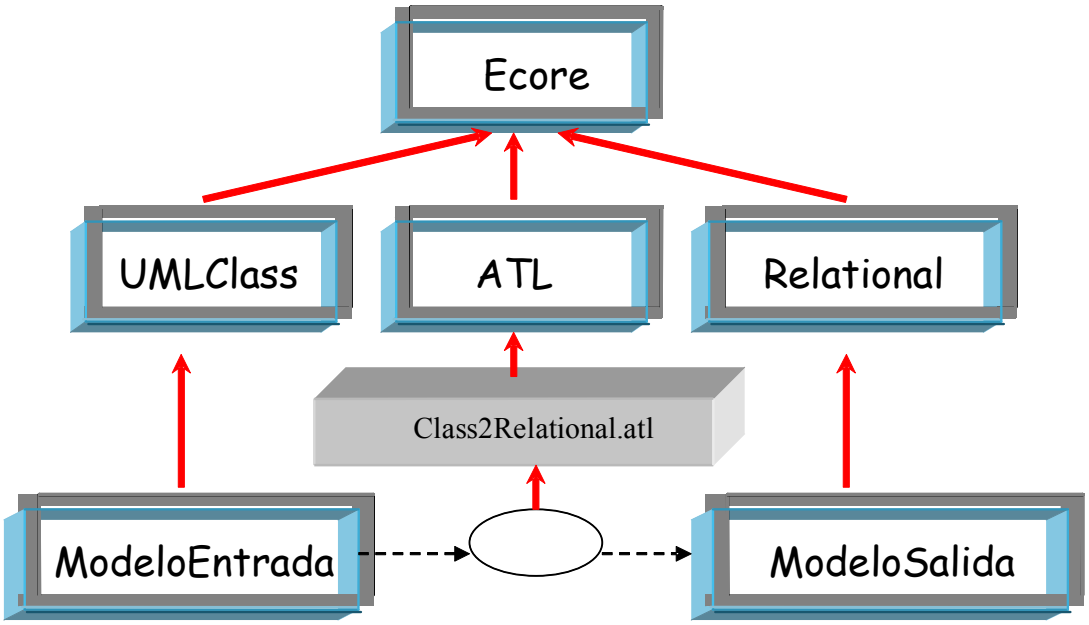


Figura 6-1 – Esquema para transformación

En la figura 6-2 se muestra un modelo UML en el cual se modela una Empresa que tiene varios empleados. Esta figura representa el modelo de entrada. De la empresa se conoce solo su nombre mientras que de los empleados se conoce el nombre de cada uno, y una lista de sus emails.

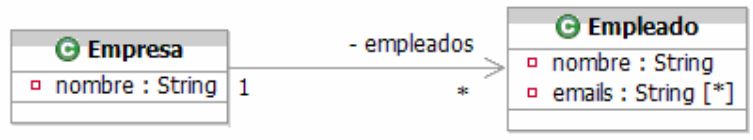


Figura 6-2 – Diagrama de clases UML para el modelo Empresa

A partir de este ejemplo se pretende obtener el esquema en el modelo relacional mostrado en la figura 6-3, donde se ejecutó la transformación mencionada anteriormente.

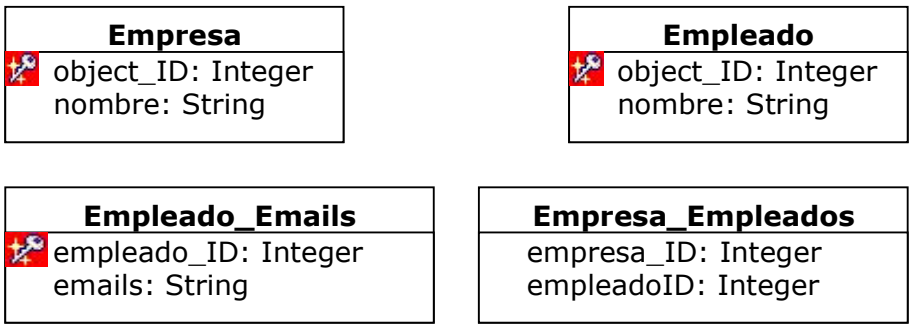


Figura 6-3 – Modelo relacional para el modelo Empresa

Observemos ahora los metamodelos de los cuales los modelos anteriores son instancia. El modelo UML es instancia del metamodelo que se muestra en la figura 6-4. Este metamodelo consiste de clases con nombre, el cual heredan de la clase abstracta NamedElement. La metaclase Class contiene un conjunto de atributos y tiene una referencia a las clases padres. La metaclase Datatype modela los tipos de datos primitivos, y al igual que Class, ambas heredan de Classifier. Los atributos pueden ser multivaluados, punto que tiene un impacto importante en la transformación.

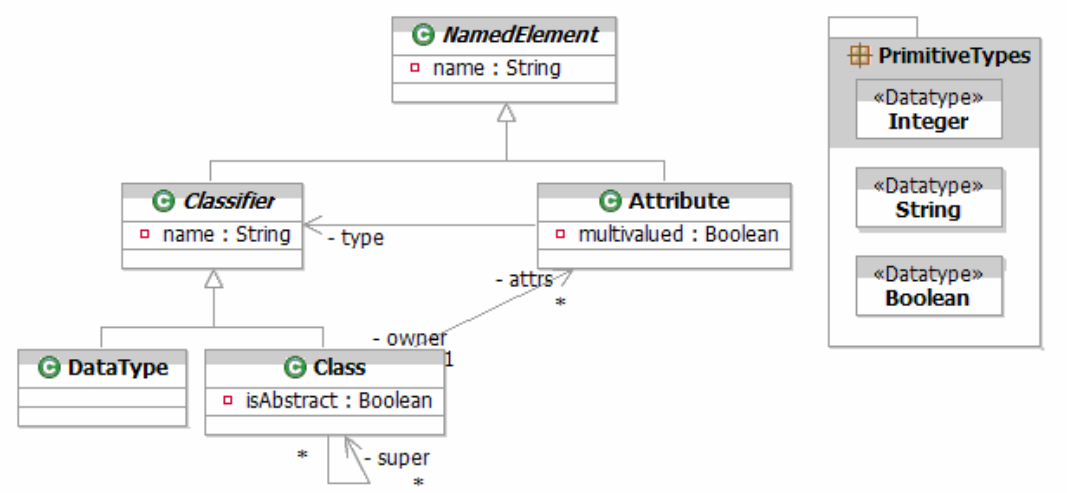


Figura 6-4 – Metamodelo UML

En la figura 6-5 se muestra el metamodelo Relacional. La clase principal en este modelo es Tabla, la cual contiene un conjunto de Columnas. Cada columna tiene un nombre y un tipo.

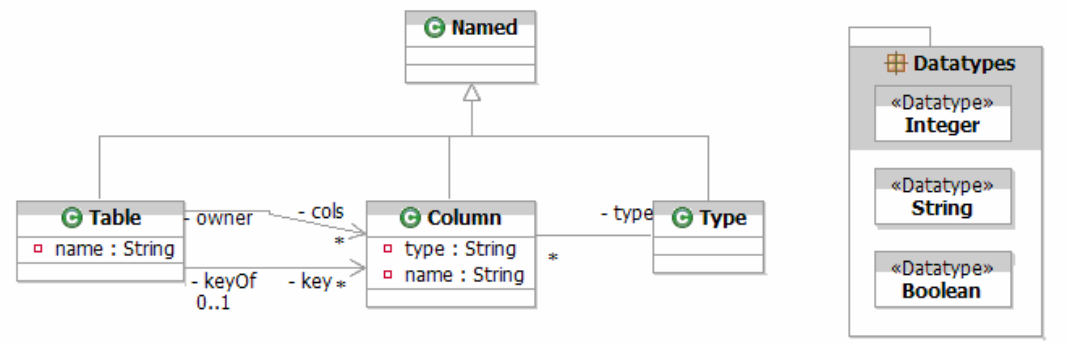


Figura 6-5 – Metamodelo Relacional

A continuación se muestran los pasos necesarios para ejecutar el ejemplo planteado en ATL.

1. Crear un proyecto ATL
2. Especificar los metamodelos de entrada y salida
3. Escribir la transformación ATL
4. Especificar el modelo de entrada
5. Ejecutar la transformación

1- Crear un proyecto ATL

Un nuevo proyecto ATL se crea desde el menú contextual New -> ATL Project, como muestra la figura 6-6. Una vez ingresado el nombre en el wizard, se creará el proyecto ATL que podrá contener los archivos necesarios para la transformación.

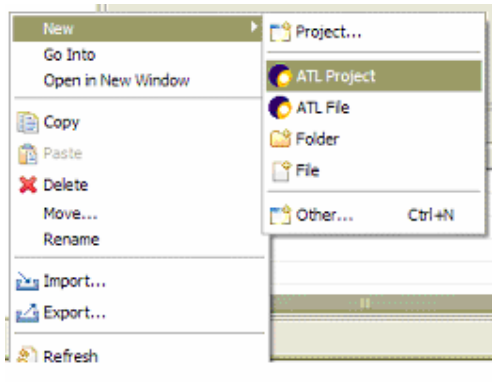


Figura 6-6 – Wizard para la creación de proyectos ATL

2- Especificar los metamodelos de entrada y salida

Como se dijo anteriormente, para poder ejecutar la transformación se requieren los metamodelos de entrada y salida. Las versiones gráficas de estos modelos están mostradas en las figuras 6-4 y 6-5. Pero esta versión gráfica no es computable. Por lo tanto, estos metamodelos deben ser especificados en otro formato.

Ya se indicó que ATL utiliza como lenguaje más abstracto la implementación de MOF: Ecore. Un metamodelo Ecore está especificado en un formato propio y los archivos se identifican por poseer extención .ecore. Este formato esta basado en la semántica del metamodelo Ecore y los archivos correspondientes están escritos en XMI 2.0. Es posible instanciar el meta-metamodelo Ecore manualmente para crear los metamodelos de entrada y salida, pero es una tarea particularmente difícil, en la que hay que tener un conocimiento extra: la sintaxis del lenguaje ecore. Por esta razón, la herramienta ATL incluye un lenguaje nuevo dedicado a la edición de metamodelos: Kernel MetaMetaModel (KM3). Se pueden especificar de esta manera los metamodelos en archivos con formato .km3 y luego inyectarlos en el formato Ecore generando los archivos .ecore. La sintaxis de KM3 es muy parecida a la sintaxis de Java. En la figura 6-7 puede verse el archivo .km3 para el desarrollo del ejemplo.

```
package Class {

	abstract class NamedElement {
		attribute name : String;
	}

	abstract class Classifier extends NamedElement {}

	class DataType extends Classifier {}

	class Class extends Classifier {
		reference super[*] : Class;
		reference attr[*] ordered container : Attribute
				oppositeOf owner;
		attribute isAbstract : Boolean;
	}

	class Attribute extends NamedElement {
		attribute multiValued : Boolean;
		reference type : Classifier;
		reference owner : Class oppositeOf attr;
	}
}

package PrimitiveTypes {
	datatype Boolean;
	datatype Integer;
	datatype String;
}
```

Archivo KM3 para especificar el metamodelo de entrada (Clases UML)

Figura 6-7 – Especificación en KM3 para el metamodelo UML

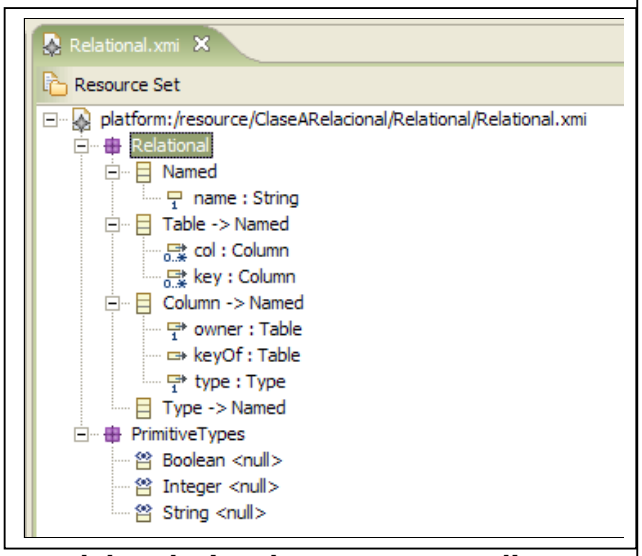
Una vez creados los archivos .km3, estos deben ser inyectados al formato ecore. Esta opción se encuentra en el menú contextual del navegador. Al ejecutarla, se genera un archivo con el mismo nombre, pero de extensión .ecore.
En la figura 6-8 se muestra como el archivo SimpleClass.ecore (resultante de inyectar en ecore el archivo SimpleClass.km3) puede ser editado con el editor provisto por EMF.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMI:XMI XMI:version="2.0" xmlns:XMI="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore">

  <ecore:EPackage name="PrimitiveTypes">
    <eClassifiers xsi:type="ecore:EDataType" name="Boolean"/>
    <eClassifiers xsi:type="ecore:EDataType" name="Integer"/>
    <eClassifiers xsi:type="ecore:EDataType" name="String"/>
  </ecore:EPackage>

  <ecore:EPackage name="Class">

    <eClassifiers xsi:type="ecore:EClass" name="NamedElement"
abstract="true">
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
ordered="false" unique="false" lowerBound="1" eType="/0/String"/>
    </eClassifiers>

    <eClassifiers xsi:type="ecore:EClass" name="Classifier" abstract="true"
eSuperTypes="/1/NamedElement"/>

    <eClassifiers xsi:type="ecore:EClass" name="DataType"
eSuperTypes="/1/Classifier"/>

    <eClassifiers xsi:type="ecore:EClass" name="Class"
eSuperTypes="/1/Classifier">
      <eStructuralFeatures xsi:type="ecore:EReference" name="super"
ordered="false" upperBound="-1" eType="/1/Class"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="attr"
upperBound="-1" eType="/1/Attribute" containment="true"
eOpposite="/1/Attribute/owner"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="isAbstract"
ordered="false" unique="false" lowerBound="1" eType="/0/Boolean"/>
    </eClassifiers>

    <eClassifiers xsi:type="ecore:EClass" name="Attribute"
eSuperTypes="/1/NamedElement">
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="multiValued"
ordered="false" unique="false" lowerBound="1" eType="/0/Boolean"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="type"
ordered="false" lowerBound="1" eType="/1/Classifier"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="owner"
ordered="false" lowerBound="1" eType="/1/Class" eOpposite="/1/Class/attr"/>
    </eClassifiers>
  </ecore:EPackage>
</XMI:XMI>
```

Figura 6-8 – Especificación del metamodelo UML en Ecore (archivo SimpleClass.ecore)

En la otra figura 6-9 se muestra también las metaclases Ecore necesarias para representar el metamodelo de salida.

```
package Relational {

abstract class Named {
              attribute name : String;
       }

class Table extends Named {
       reference col[*] ordered container : Column oppositeOf owner;
       reference key[*] : Column oppositeOf keyOf;
       }

class Column extends Named {
       reference owner
: Table oppositeOf col;
      reference keyOf[0-1]
: Table oppositeOf key;
       reference type
: Type;
       }

class Type extends Named {}
}

package PrimitiveTypes {
       datatype Boolean;
       datatype Integer;
       datatype String;
}
```



Archivo .km3 para el metamodelo relacional y su correspondiente archivo .ecore representando la misma información

Figura 6-9 – Especificación del metamodelo Relacional en KM3

3- Escribir la transformación ATL

Para escribir la transformación ATL primero hay que crear un archivo nuevo con extensión .atl con el wizard para crear archivos de Eclipse.
Al clickear sobre este archivo, se abrirá un editor de texto en el cual se debe escribir la transformación requerida.
A continuación se detalla la transformación requerida para este ejemplo.

```
module Class2Relational;
create OUT : Relational from IN : Class;

uses strings;

helper def: objectIdType : Relational!Type =
        Class!DataType.allInstances()->
                select(e | e.name = 'Integer')->first();

rule Class2Table {
        from
                c : Class!Class
        to
                out : Relational!Table (
```

```
                name <- c.name,
                col <-
                        Sequence {key} ->
                        union(c.attr->select(e | not e.multiValued)),
                key <- Set {key}
        ),

        key : Relational!Column (
                name <- 'objectId',
                type <- thisModule.objectIdType
        )
}

rule DataType2Type {
        from
                dt : Class!DataType
        to
                out : Relational!Type (
                        name <- dt.name
                )
}

rule DataTypeAttribute2Column {
        from
                a : Class!Attribute (
                        a.type.oclIsKindOf(Class!DataType)
                        and not a.multiValued
                )
        to
                out : Relational!Column (
                        name <- a.name,
                        type <- a.type
                )
}
rule MultiValuedDataTypeAttribute2Column {
        from
                a : Class!Attribute (
                        a.type.oclIsKindOf(Class!DataType) and a.multiValued
                )
        to
                out : Relational!Table (
                        name <- a.owner.name + '_' + a.name,
                        col <- Sequence {id, value}
                ),
                id : Relational!Column (
                        name <- a.owner.name.firstToLower() + 'Id',
                        type <- thisModule.objectIdType
                ),
                value : Relational!Column (
                        name <- a.name,
                        type <- a.type
                )
```

```
}
rule ClassAttribute2Column {
	from
		a : Class!Attribute (
			a.type.oclIsKindOf(Class!Class) and not a.multiValued
		)
	to
		foreignKey : Relational!Column (
			name <- a.name + 'Id',
			type <- thisModule.objectIdType
		)
}
rule MultiValuedClassAttribute2Column {
	from
		a : Class!Attribute (
			a.type.oclIsKindOf(Class!Class) and a.multiValued
		)
	to
		t : Relational!Table (
			name <- a.owner.name + '_' + a.name,
			col <- Sequence {id, foreignKey}
		),
		id : Relational!Column (
			name <- a.owner.name.firstToLower() + 'Id',
			type <- thisModule.objectIdType
		),
		foreignKey : Relational!Column (
			name <- a.name + 'Id',
			type <- thisModule.objectIdType
		)
}
```

4- Especificar el modelo de entrada

El modelo de entrada debe ser una instancia del metamodelo de entrada ya especificado. Este modelo se especifica en un archivo de texto, con formato XMI, de la siguiente manera:

```
<?xml version="1.0" encoding="ASCII"?>
<XMI:XMI XMI:version="2.0" xmlns:XMI="http://www.omg.org/XMI"
xmlns="Class">
  <Class name="Empresa">
    <attr name="nombre" multiValued="false" type="/2"/>
    <attr name="empleados" multiValued="true" type="/1"/>
  </Class>
  <Class name="Empleado">
    <attr name="nombre" multiValued="false" type="/2"/>
    <attr name="emailAddresses" multiValued="true" type="/2"/>
  </Class>
  <DataType name="String"/>
</XMI:XMI>
```

5- Ejecutar la transformación

Se ejecuta un proyecto ATL mediante el comando Run -> Run…

En la figura 6-10 se muestra la ventana que permite especificar el archivo que contiene el modelo y metamodelo de entrada. En la tabla, IN contiene la ubicación del modelo de entrada (Sample-SimpleClass.ecore) y SampleUML contiene la ubicación del archivo con extensión .ecore que contiene el metamodelo SampleUML.

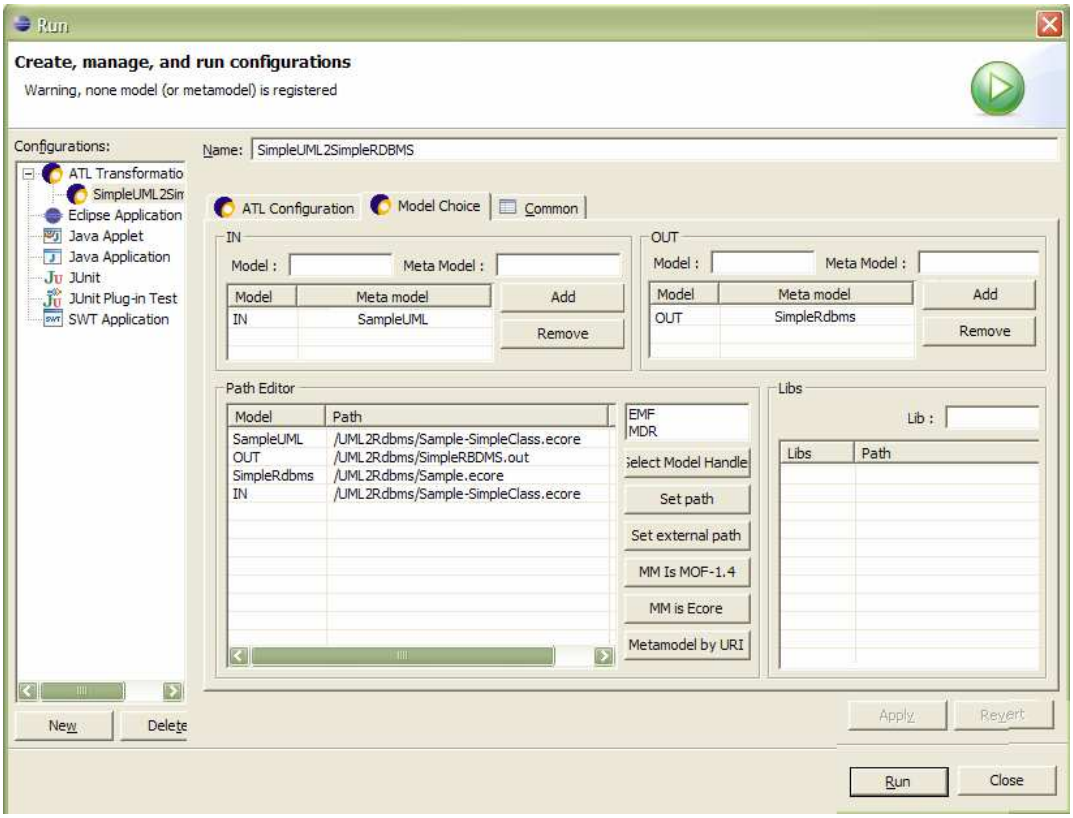


Figura 6-10 – Ventana de configuración de ATL

Por otro lado, se especifica el metamodelo de salida (OUT) y se especifica como modelo, un nombre de archivo que será generado al ejecutar la transformación (en este caso SimpleRdbms.out)

Al ejecutar la transformación, se genera el siguiente archivo de texto

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMI:XMI XMI:version="2.0" xmlns:XMI="http://www.omg.org/XMI"
xmlns="Relational">
  <Table name="Family" key="/0/@col.0">
    <col name="objectId" keyOf="/0" type="/3"/>
    <col name="name" type="/2"/>
  </Table>
  <Table name="Person" key="/1/@col.0">
    <col name="objectId" keyOf="/1" type="/3"/>
    <col name="firstName" type="/2"/>
```

```
    <col name="closestFriendId" type="/3"/>
  </Table>
  <Type name="String"/>
  <Type name="Integer"/>
  <Table name="Person_emailAddresses">
    <col name="PersonId" type="/3"/>
    <col name="emailAddresses" type="/2"/>
  </Table>
  <Table name="Family_members">
    <col name="FamilyId" type="/3"/>
    <col name="membersId" type="/3"/>
  </Table>
</XMI:XMI>
```

6.2 Caso de estudio MOFScript

Como se comentó en el capítulo anterior, MOFScript permite la transformación de cualquier modelo compatible con MOF a texto.

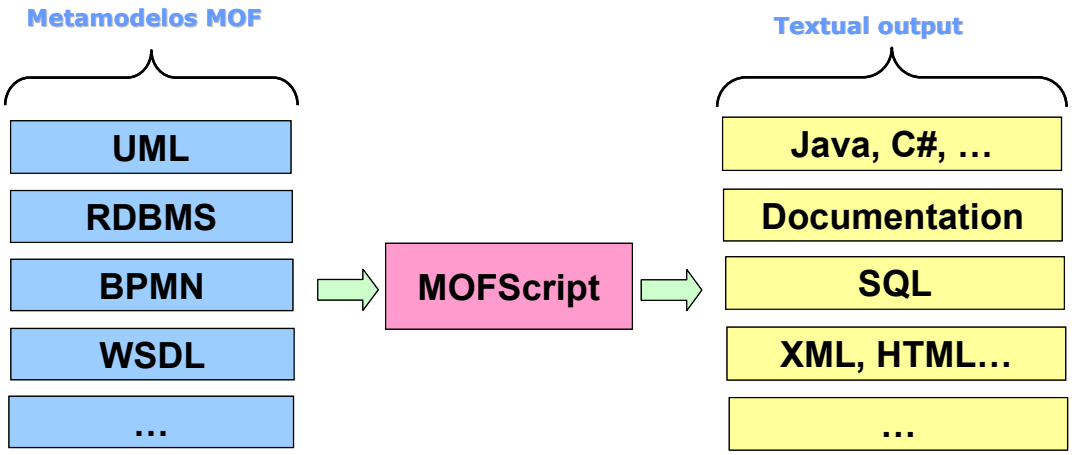


Figura 6-11 – Esquema de transformación Modelo a Texto

Como puede verse en la figura 6-11, la entrada a la herramienta MOFScript puede ser cualquier modelo cuyo metamodelo sea compatible con MOF, como lo son UML, RDBMS, etc. La salida será cualquier documento de texto que se especifique, los cuales pueden estar o no en el mismo archivo. Por ejemplo, en el caso de Java, puede especificarse la transformación para que se genere un archivo por clase, o en el caso de HTML, que se genere todo dentro del mismo archivo.

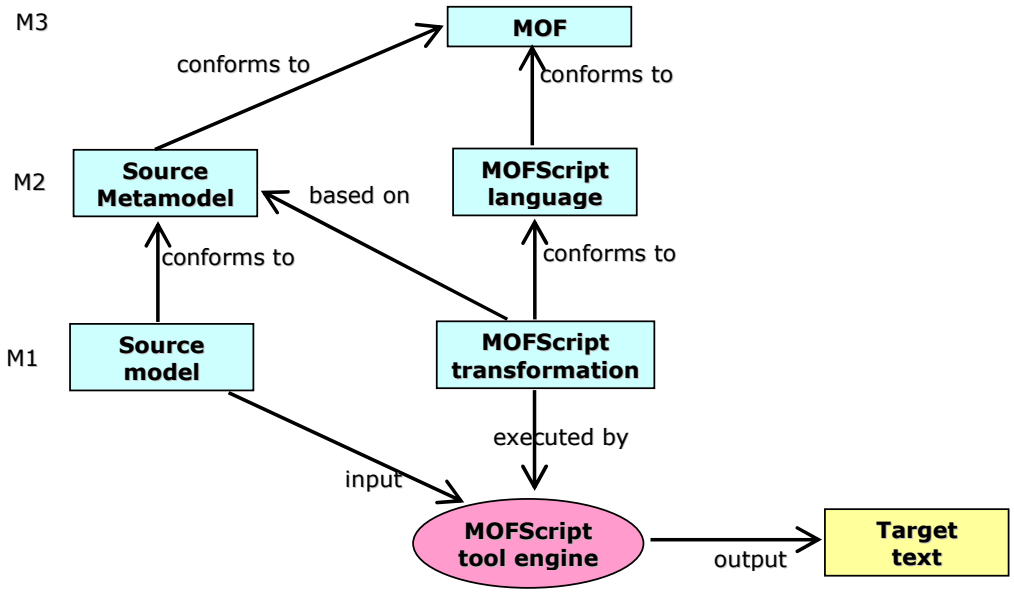


Figura 6-12 – Transformación Modelo a Texto en las 4 capas de modelado

La figura 6-12 muestra la relación entre los elementos necesarios para ejecutar una transformación. Puede verse que sólo el modelo fuente tiene metamodelo definido. La salida no tiene un metamodelo explícitamente definido, es un texto cuyo formato se especifica en la transformación.

A continuación se muestran los pasos necesarios para ejecutar el ejemplo planteado en MOFScript.

1. Crear un proyecto
2. Especificar el metamodelo de entrada
3. Escribir la transformación MOFScript
4. Especificar el modelo de entrada
5. Compilar y ejecutar la transformación

1- Crear un proyecto

Una transformación en MOFScript puede ser ejecutada desde cualquier proyecto, es decir, no se necesita crear un proyecto MOFScript, sino que se puede crear la transformación dentro de un proyecto JAVA o cualquier otro.

2- Especificar el metamodelo de entrada

Como metamodelo de entrada pueden usarse algunos definidos ya por los proveedores de la herramienta, o definir metamodelos nuevos. Los metamodelos provistos se encuentran disponibles on line y algunos son UML, OCL, Ecore y Relational.

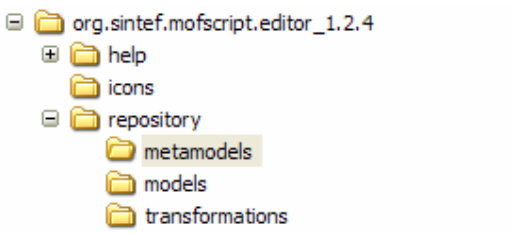


Figura 6-13 – Repositorio de metamodelos

Definir un nuevo metamodelo no es una tarea fácil, y utilizando esta herramienta, usarlo tampoco. Para esto, se los debe guardar dentro de la carpeta de metamodelos correspondiente dentro del repositorio de uno de los plugins de MOFScript. La figura 6-13 muestra donde deben almacenarse. Luego de almacenarlos, se debe reiniciar eclipse para que la herramienta los reconozca.

3- Escribir la transformación MOFScript

```
texttransformation ExampleTransformation (in
ec:"http://www.eclipse.org/emf/2002/Ecore") {

  /**
   * Main (entry point)
   */
  ec.EClass::main () {
     file (self.name + ".java")
     'public class ' self.name ' { '
        self.eAllAttributes->forEach(p:ec.EAttribute) {
        p.privateProperty()
       }
    '
} // end of class ' self.name '
'
  }

  ec.EAttribute::privateProperty () {
    '   private ' self.eType.name ' ' self.name.toLower() ';
    '
  }
}
```

4- Compilar

Una vez escrita la transformación, se debe compilar para saber si existen errores y poder corregirlos.
La figura 6-14 muestra como compilar la transformación (con una opción del menú contextual). Si hubiera errores se mostrarán en la consola.

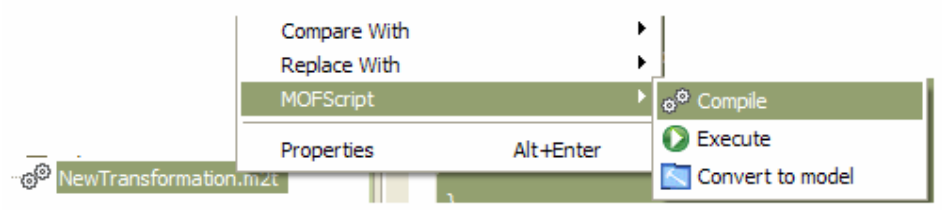


Figura 6-14 – Menú contextual para compilar un archivo MOFScript

5- Especificar el modelo de entrada

Como ayuda para definir el modelo de entrada, MOFScript permite definirlo con MOF como un diagrama de clases. Luego se tiene el archivo con extensión .ecore, al cual se le agrega información por medio de otro archivo con extensión .genmodel. Este archivo a otro formato, como UML, XSD.
Al momento de ejecutar la transformación, debe especificarse este modelo de entrada.

6- Ejecutar la transformación

La transformación se ejecuta mediante una opción en el menú contextual sobre el archivo de transformación, como se muestra en la figura 6-15. Al momento de ejecutarla, se debe ingresar el archivo que representa al modelo.
La salida de la transformación puede imprimirse en la consola, o se puede especificar dentro de archivos, como es este caso, donde se genera un archivo por cada una de las clases JAVA generadas.

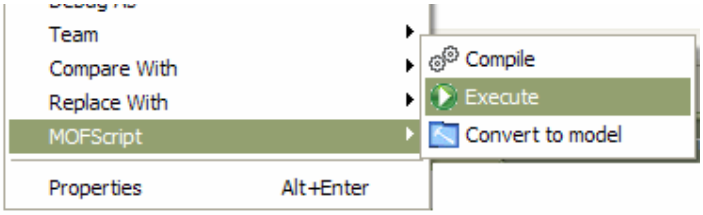


Figura 6-15 – Menú contextual para ejecutar un archivo MOFScript

6.3 Conclusiones

En este capítulo se ilustró el uso de las herramientas ATL y MOFScript -mediante la ejecución de una transformación de UML a Relacional en el caso de ATL y de UML a código JAVA en el otro-. Ambas herramientas tienen en cuenta el tema de interoperabilidad, ya que usan XMI. Ambas soportan el estándar MOF para las definiciones.
Se puede observar que la principal dificultad en ambas herramientas aparece a la hora de especificar los metamodelos y los modelos necesarios.
Al respecto, en el caso de ATL, éste define un lenguaje adicional, KM3 que asiste a la especificación de los metamodelos. Por otro lado, MOFScript define una librería de metamodelos on-line, pero no brinda ninguna ayuda adicional en el caso que el usuario quiera especificar su propio metamodelo.
Concluyendo, se hace evidente entonces la necesidad de contar con un editor gráfico para la creación de metamodelos, que luego puedan ser

usados como entrada en éstas y otras herramientas de transformación. Además, se hace necesario la posibilidad de instanciar ese metamodelo fácilmente, para posibilitar la creación de un modelo libre de errores.

El próximo capítulo se presenta la herramienta ePlatero, y las extensiones que implementan los objetivos planteados en esta tesis.

7 Aporte a las herramientas para transformaciones entre modelos - Extensión de la herramienta ePlatero

e-Platero es el acrónimo para "Eclipse PLugin Aiding Traceability in an Environment with Refinement-Orientation". Es una herramienta implementada como plugins para la plataforma de Eclipse, que soporta el proceso de desarrollo conducido por modelos usando una notación gráfica con base formal. Es la evolución de PAMPERO [4, 26, 27], un plugin implementado para la versión de Eclipse 2.1.

7.1 Objetivos

Los principales objetivos de ePlatero son:

- Creación y edición gráfica de modelos UML
- Edición y validación de restricciones OCL en el nivel del metamodelo, incluyendo reglas de buena formación para modelos
- Edición y validación de restricciones OCL en el nivel de modelos (reglas de negocio)
- Edición y validación de relaciones de refinamiento entre modelos: se puede editar un mapping para especificar la relación entre elementos abstractos y concretos

Los objetivos agregados en la presente tesis son:

- Creación y edición gráfica de modelos Ecore (para la definición de metamodelos)
- Instanciación del metamodelo Ecore (para la definición de modelos)
- Edición y validación de restricciones OCL en el nivel del meta-metamodelo, para verificar si un metamodelo esta bien definido.
- La posibilidad de interoperar con otras herramientas, que soporten MDA, mediante el uso del estándar xmi.

7.2 Arquitectura de ePlatero

La arquitectura de ePlatero respeta el estándar de Eclipse para el desarrollo de plugins. Está formado de 9 plugins relacionados entre sí, como se muestra en la figura 7-1:

- Editor UML
- Editor de metamodelos
- Editor de fórmulas OCL
- Evaluador de fórmulas OCL
- Analizador léxico / Parseador
- Repositorio del proyecto
- Coordinador
- Evaluador de refinamiento

- Generador de Micro-Mundos

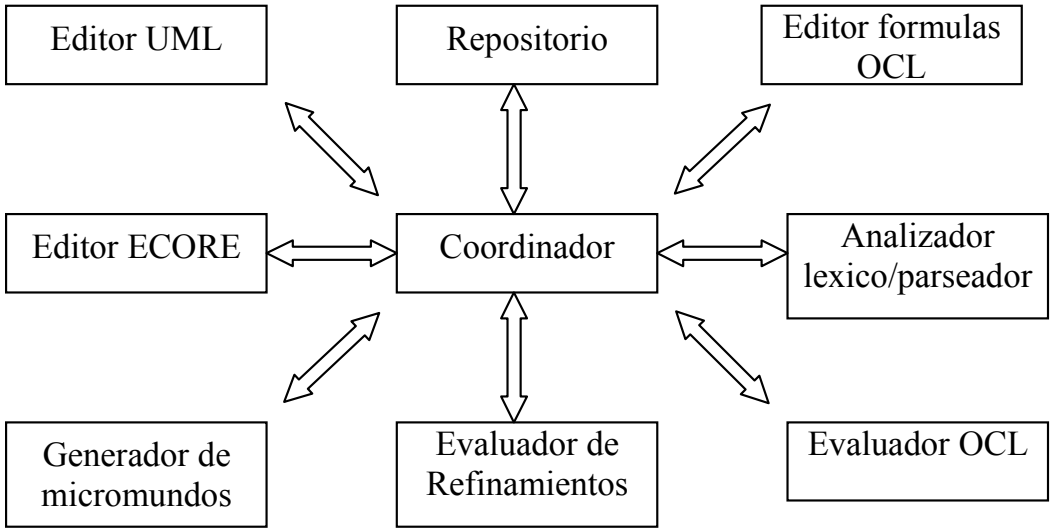


Figura 7-1 – Arquitectura de ePlatero

¿Qué plugins se usaron para la construcción de la presente extensión de ePlatero?
La presente extensión de ePlatero se construyó en base a tres plugins desarrollados en los laboratorios de IBM: EMF [7,9], GMF [12] y Ecore. EMF permite mantener los repositorios de diagramas, modelos, y metamodelos. GMF permite la construcción del código base para el editor gráfico de los metamodelos y Ecore representa el meta-metamodelo, que se instancia a través del editor gráfico. Además, para la evaluación de las reglas OCL sobre los metamodelos definidos se modificó un plugin ya definido en ePlatero (el evaluador de fórmulas OCL).

7.3 Descripción de los módulos existentes de ePlatero

Para la implementación de la herramienta que se describe a continuación, se utilizaron algunos componentes de ePlatero y se agregaron otros plugins. En las siguientes secciones se explica brevemente algunos de los módulos que integran la versión original de ePlatero. Luego se explica las extensiones planteadas en este trabajo.

7.3.1 Editor gráfico de modelos UML

El editor gráfico de UML esta desarrollado en base a los plugins GMF, EMF y UML2. Este editor permite crear modelos ePlatero. Un modelo ePlatero está formado por un diagrama de clases UML donde se instancian elementos de UML, como por ejemplo, una clase. Se hace una clara separación entre el diagrama, y los atributos propios del diagrama y el modelo en sí mismo. Por lo tanto, un modelo ePlatero se compone de dos archivos, un archivo con extensión .ePlatero y otro con extensión .uml. El primero contiene los

elementos del diagrama mientras que el segundo mantiene las instancias de los metaelementos de UML2.

La figura 7-2 muestra como crear un modelo con ePlatero. Para esto, dentro de cualquier proyecto (puede ser un proyecto Java Java o de cualquier otro tipo) en la opción File>>New>> Other>>Model Wizard, hay que buscar la categoría ePlatero, donde se encuentran los wizards definidos en la herramienta.

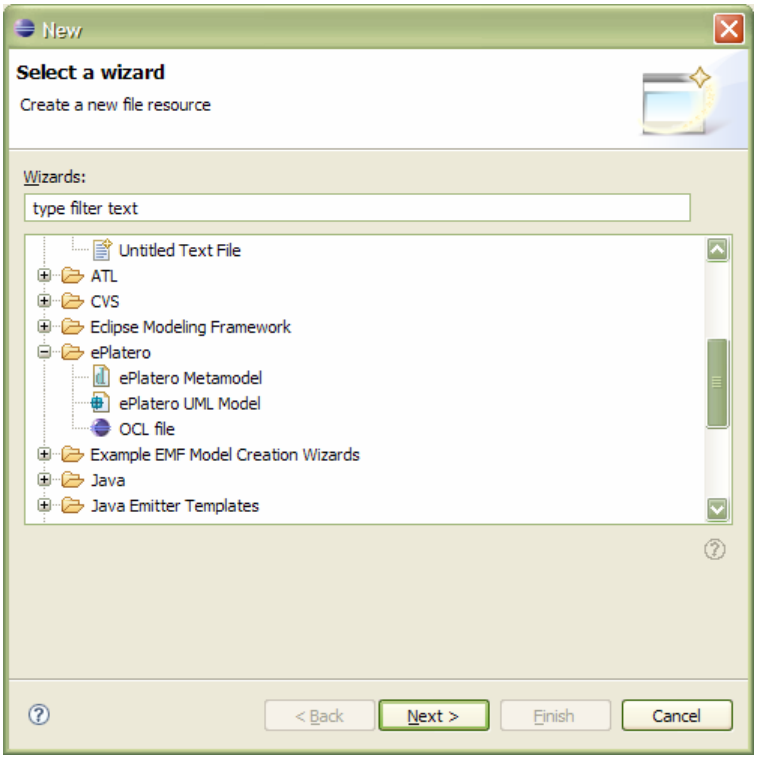


Figura 7-2 – Wizard para la creación de modelos UML, metamodelos y archivos OCL

El primer *wizard* (o asistente), "ePlatero Metamodel" permite crear los metamodelos eCore que posteriormente se usarán en las herramientas de transformación. El segundo *wizard*, "ePlatero Model" permite crear modelos UML. Por último, el tercer *wizard* permite definir archivos OCL para escribir las reglas sobre los modelos, y metamodelos.

En la figura 7-3 se muestra el editor de diagramas de clases editando un nuevo modelo UML llamado "Flights".

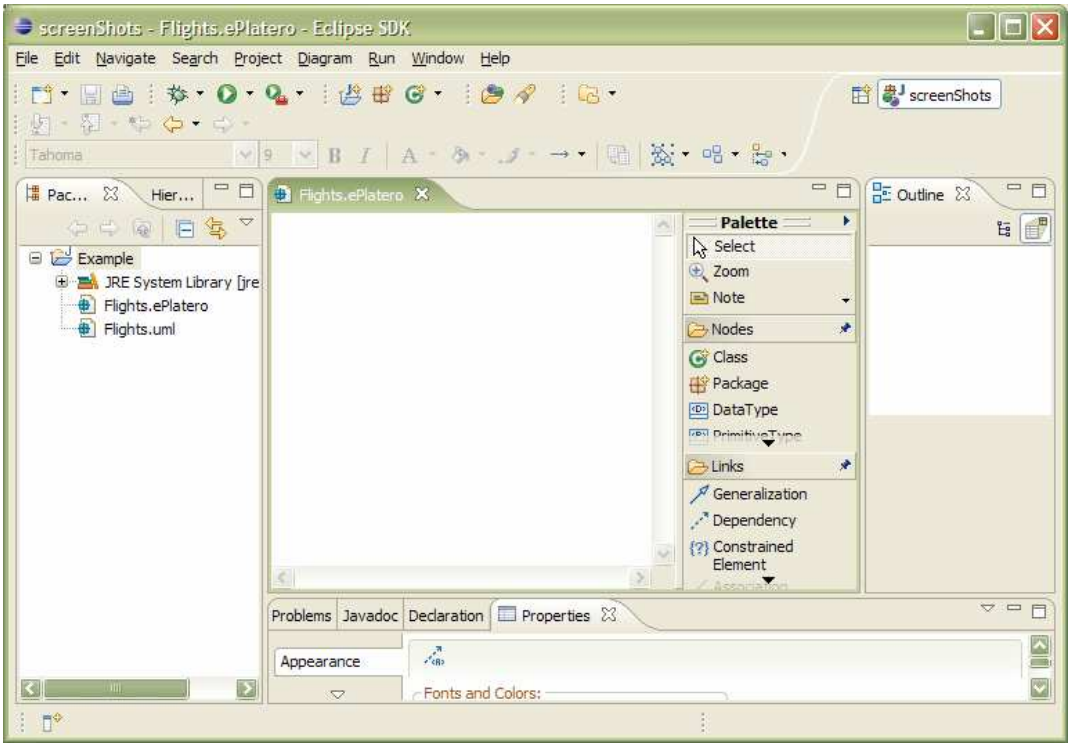


Figura 7-3 – Editor ePlatero para modelos UML

En la vista de la izquierda se ven los recursos creados por el editor. Como se mencionó anteriormente, por cada modelo se crean dos archivos, uno con extensión .ePlatero en el cual se encuentra la información gráfica, es decir, las posiciones donde se dibujan los elementos, los colores utilizados, etc., mientras que en el segundo archivo, con extensión .uml, contiene la información propia del modelo, como son las clases que contiene, las relaciones entre ellas, etc. Estos elementos son una instanciación de los meta-elementos definidos en el metamodelo de UML.

Estos dos archivos necesitan estar sincronizados para que el modelo se mantenga válido. Todo cambio en el editor gráfico se verá reflejado en el archivo .uml. Para ver los cambios en el archivo uml, clickeando sobre él, se abre un editor EMF, el cual permite navegar a través de los elementos instanciados, como se muestra en la figura 7-4.

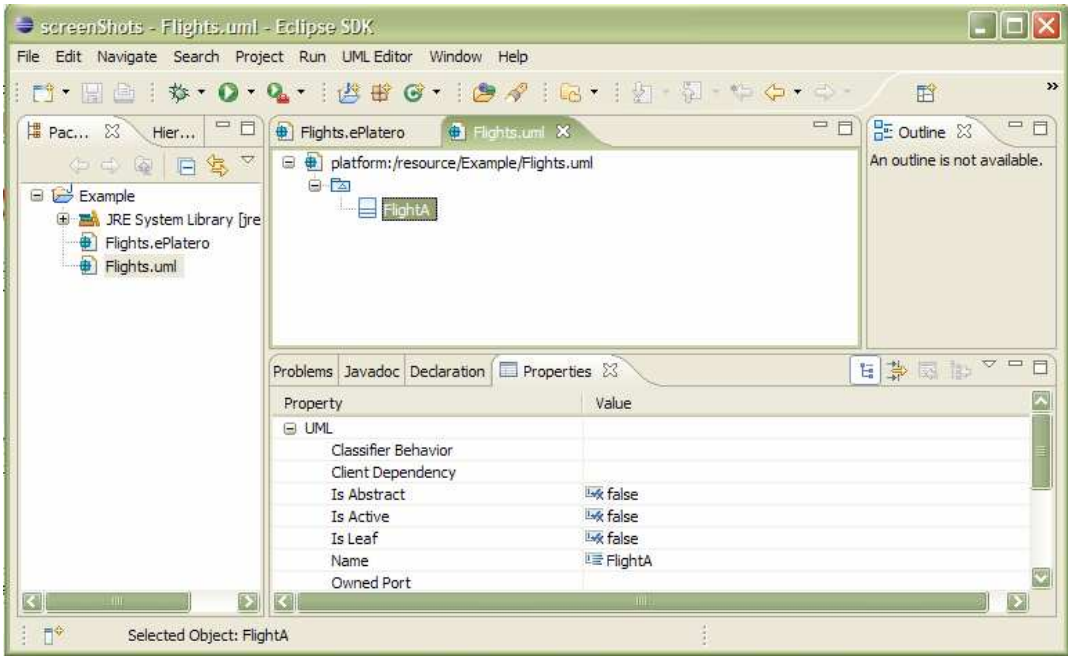


Figura 7-4 – Editor EMF para archivo uml

7.3.2 Editor y evaluador de fórmulas OCL

El editor de fórmulas OCL se usa para editar las reglas de buena formación que se quieren evaluar. Este editor permite la edición de propiedades, sintaxis *highlighting* (las palabras clave están coloreadas para una mejor visualización de la fórmula), asistencia en la escritura del código y corrección de errores. Asimismo muestra un *outline* en el cual puede verse la estructura del archivo OCL. En la figura 7-5 puede verse este editor.

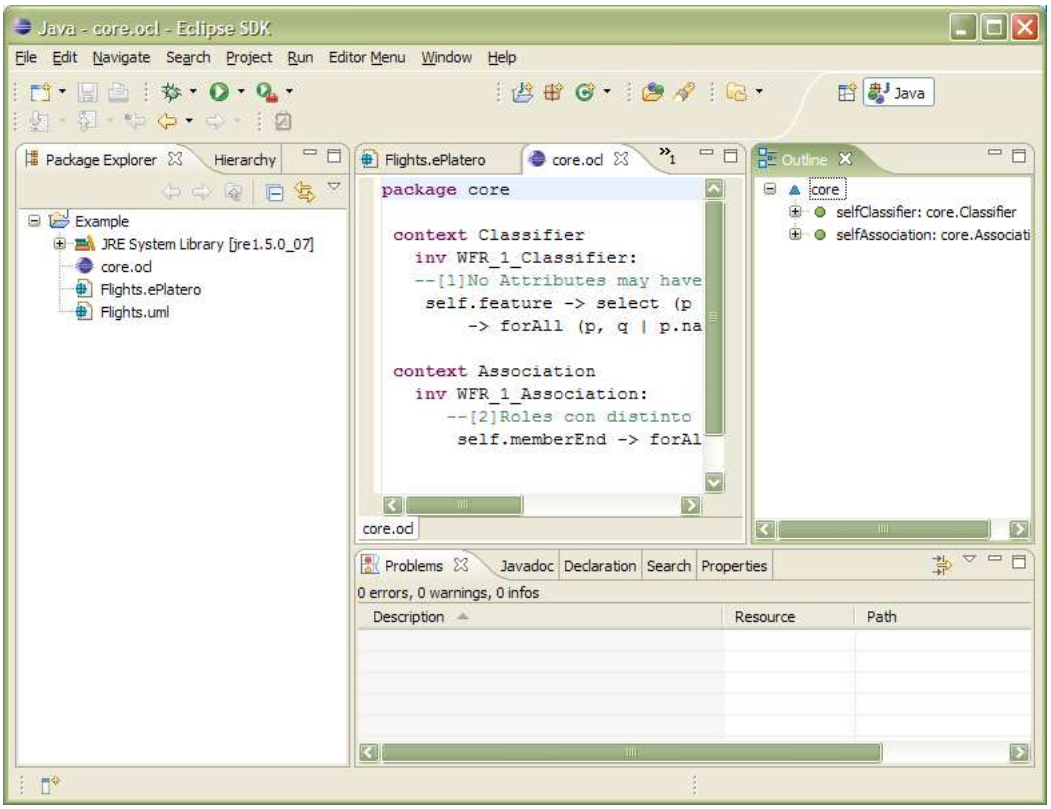


Figura 7-5 – Editor de reglas OCL

El evaluador OCL es el responsable de ejecutar el análisis semántico de las fórmulas OCL. La versión original de ePlatero permite la evaluación de invariantes en dos niveles: invariantes de clase y de metaclase, es decir, evalúa a nivel modelo y metamodelo pero solo sobre UML.

Un ejemplo de una invariante a nivel metamodelo UML seria la invariante que se muestra en la figura 7-6, donde puede verse en el editor de fórmulas OCL algunas reglas de buena formación sobre dos de los elementos de UML: Classifier y Association. El evaluador esta ligado al metamodelo UML.

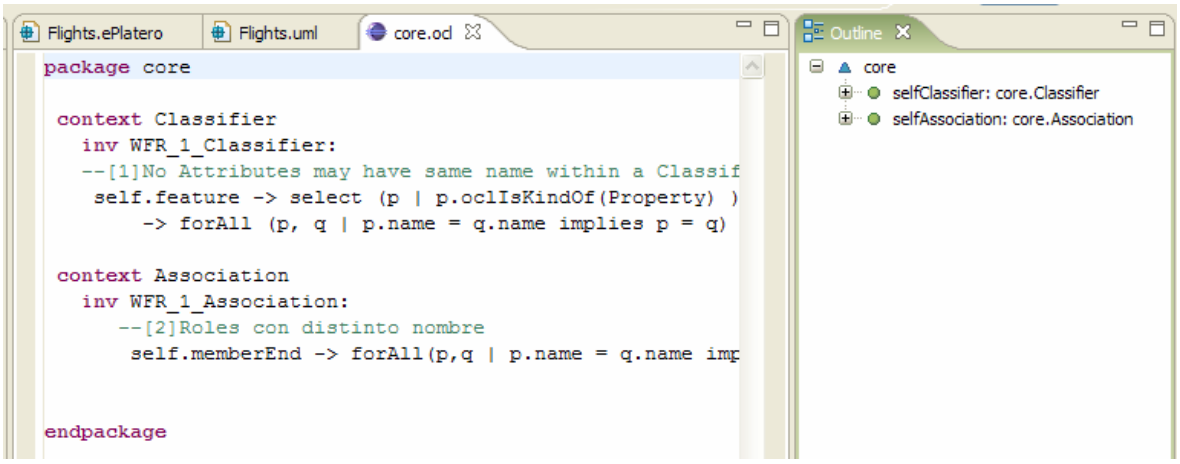


Figura 7-6 – Editor de fórmulas OCL para invariantes de metamodelo

Las reglas escritas en el archivo con extensión .ocl serán evaluadas mediante la opción Evaluate (menú contextual iniciado sobre el archivo), como se muestra en la figura 7-7. Esta opción solicitará el ingreso (o la selección) del modelo (archivo uml) que se desea validar.

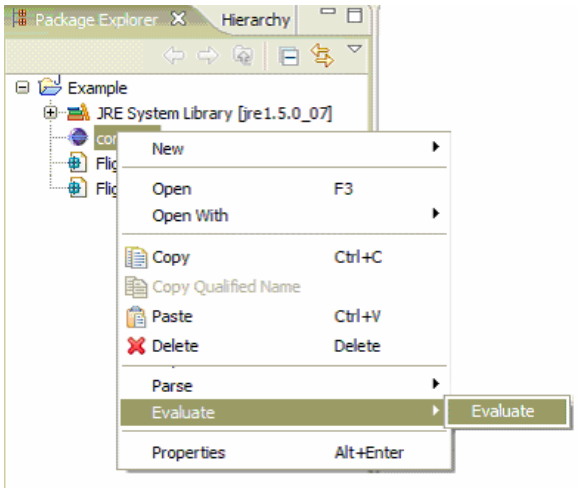


Figura 7-7 – Menú para evaluar las fórmulas OCL sobre un modelo

Si el modelo no cumple las reglas definidas, se reportarán los errores como se muestra en la figura 7-8. Esto permite ayudar al usuario a identificar el error y facilita la corrección del mismo. En la descripción del error, se muestra el comentario escrito en la regla de buena formación. En la columna "*Path*" se indica el proyecto en el cual se encuentra el archivo con error y en la columna "*Location*" se muestra el archivo del modelo y el elemento no cumple con la invariante.

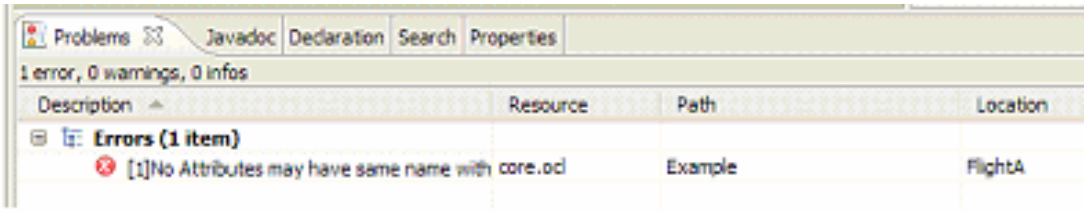


Figura 7-8 – Vista problemas indicando los errores

Un ejemplo de una invariante a nivel modelo seria la siguiente.

```
context Classifier inv:
        self.feature -> select ( p| p.oclIsKindOf(Operation))
        -> size() < 2
```

Esta invariante es una regla de diseño, que especifica que en un buen diseño las instancias de un Classifier deben tener siempre menos de dos operaciones.

De la misma manera que en las reglas de metamodelo, los errores son informados en la vista problemas, con pistas para poder corregirlos.

7.3.3 Generador de micromundos

Refinamiento es una relación entre dos elementos o modelos, donde uno de ellos especifica de forma completa al otro que ya ha sido definido con cierto detalle. La especificación del refinamiento nos indica como se relaciona el elemento abstracto con el concreto. Puede evaluarse, a través de una sucesión de pasos matemáticos, la correctitud con respecto a la especificación original. Los evaluadores de OCL tradicionales son incapaces de determinar si una condición de refinamiento escrita en OCL se cumple en un modelo UML, porque se evalúan fórmulas OCL en una instancia particular del modelo, mientras que las condiciones de refinamiento necesitan ser validadas en todas las posibles instanciaciones. Por lo tanto para evaluar las condiciones de refinamiento, se extrae del modelo UML un número relativamente pequeño de instanciaciones, y se verifica si satisfacen las condiciones mencionadas. Esta estrategia, llamada micromodelos de software, fue propuesta por Daniel Jackson [13] para evaluar fórmulas escritas en Alloy.

Si la respuesta es positiva, hay una posibilidad de que la propiedad se cumple. En este caso, la respuesta no es definitiva porque puede haber un mundo más grande que no cumple con la propiedad, sin embargo la respuesta positiva es alentadora. Si la respuesta es negativa, entonces existe al menos un mundo que viola la propiedad. En ese caso, la respuesta es definitiva ya que la propiedad no se cumple en el modelo.

ePlatero entonces, permite definir relaciones de refinamiento entre modelos, y especificar y evaluar restricciones sobre los mismos. Para esto, se genera un micromundo siguiendo ciertos criterios para su creación, donde poder evaluarlas.

7.4 Descripción de los módulos nuevos de ePlatero

En las siguientes secciones se explica las nuevas funcionalidades de ePlatero: el editor gráfico de metamodelos, que permite definir nuevos metamodelos, el instanciador de modelos,para poder instanciar un modelo a partir de un metamodelo dado y el evaluador de formulas OCL a nivel meta-metamodelo.

7.4.1 Editor gráfico de metamodelos

El editor gráfico de metamodelos esta desarrollado en base a los plugins GMF, EMF y Ecore. Este editor permite especificar metamodelos mediante diagramas de clases. Cada metamodelo esta especificado en dos archivos, el primero con extensión .m2epl y un segundo archivo con extensión .ecore. Análogamente a lo que pasa con los modelos ePlatero, el primer archivo guarda información perteneciente al diagrama y el segundo las instancias de los elementos del metamodelo Ecore.

Para poder crear un metamodelo mediante el editor de metamodelos de ePlatero, dentro de cualquier proyecto (puede ser un proyecto Java, o de otro tipo) en la opción File>>New>> Other, hay que buscar la categoría ePlatero, y elegir la opción "ePlatero Metamodel Diagram".

Una vez elegida la opción, se abre el editor de metamodelos que permite diagramar mediante los elementos de la paleta lateral. Estos elementos son los elementos que define el metamodelo Ecore, como EClass, EAttribute, etc. La figura 7-9 muestra el editor de metamodelos.

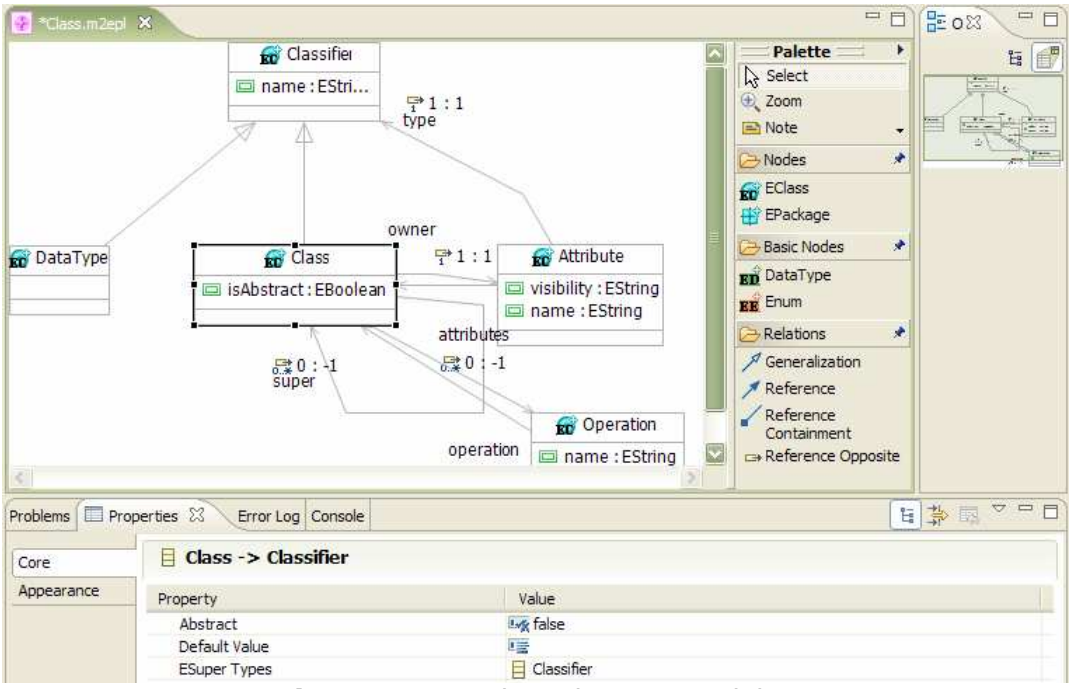


Figura 7-9 – Editor de metamodelos

En la vista de la izquierda se ven los recursos creados. Por cada metamodelo definido se crean dos archivos: uno con extensión .m2epl, en el cual se encuentra la información grafica, es decir, las posiciones donde se dibujan los elementos, los colores utilizados, etc., mientras que en el segundo archivo, el archivo con extensión .ecore, se guarda la información del metamodelo mismo, como las clases que contiene, las relaciones entre ellas, los tipos de datos definidos, etc. Estos elementos son una instanciación de los meta-metaelementos definidos en el metamodelo de Ecore.

Estos dos archivos necesitan estar sincronizados para tener un modelo válido. Todo cambio en el editor gráfico se verá reflejado en el archivo con extensión .ecore. Para ver los cambios en el archivo ecore, clickeando sobre el archivo se abre un editor EMF, el cual permite navegar a través de los elementos instanciados. Como se ve en la figura 7-10, la selección de un elemento permite la edición de sus propiedades.

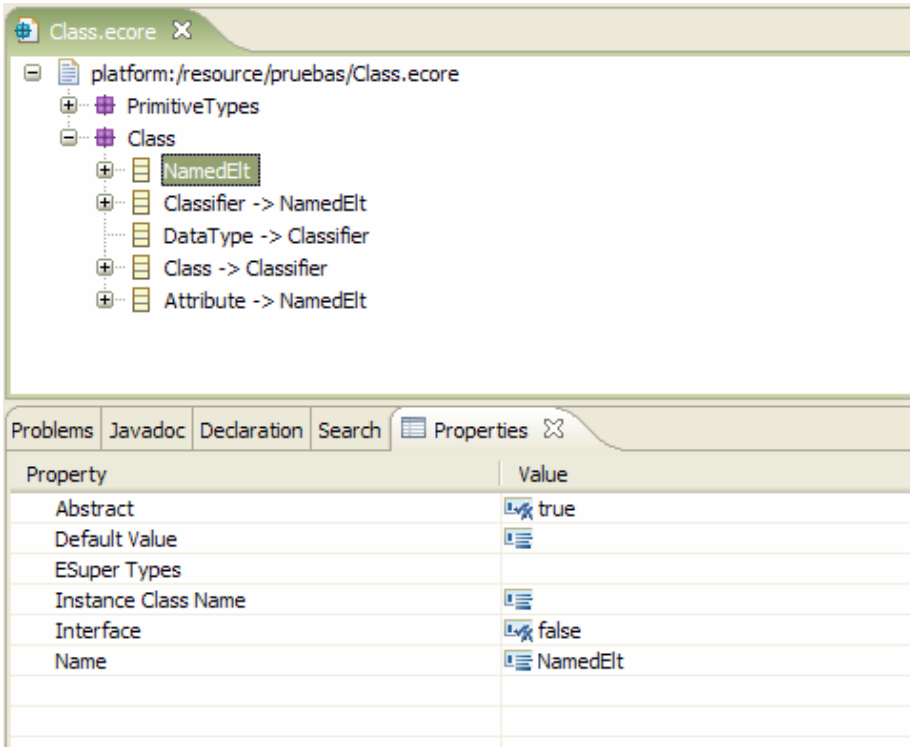


Figura 7-10 – Editor EMF

7.4.2 Instanciar un modelo a partir del metamodelo

El mismo plugin que permite definir metamodelos también permite instanciarlos. Para lograr esto, en la vista del archivo con extensión .ecore, donde se encuentra la información de los elementos del metamodelo definido, se elige el elemento que se desea instanciar y se selecciona la opción "**Create Instance**" como lo muestra la figura 7-11.

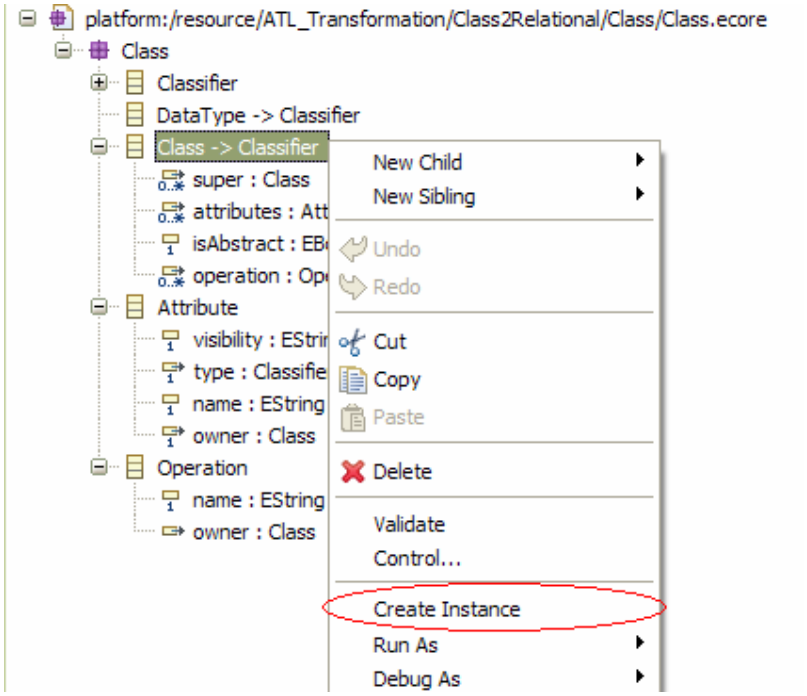


Figura 7-11 – Crear instancias de los elementos del metamodelo

La primer instancia generará un archivo con el mismo nombre que el archivo ecore, pero con extensión xmi. Las instancias siguientes agregarán información a este archivo. En la figura 7-12 se muestra el árbol de estas instancias.

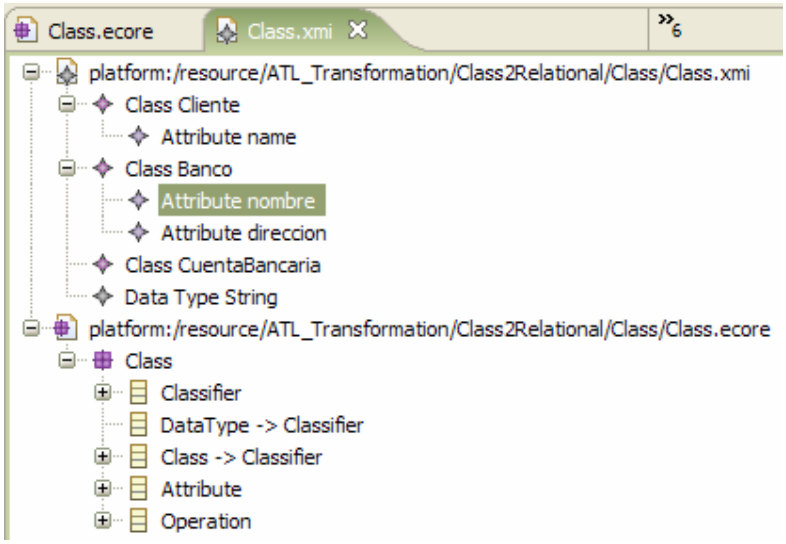


Figura 7-12 – Modelo Banco, instancia del metamodelo Class

7.4.3 Evaluador de fórmulas OCL a nivel meta-metamodelo

EPlatero define un evaluador de fórmulas OCL a nivel modelo y metamodelo. Con la extensión presentada en este trabajo, se hizo necesaria

la evaluación de fórmulas para poder evaluar buena formación sobre los metamodelos definidos. Los archivos para definir estas fórmulas son los mismos que para definir reglas sobre el metamodelo, salvo que al evaluar las fórmulas se realiza sobre un modelo ecore. La evaluación se realiza de la misma manera, mediante la opción "Evaluate" del menú contextual. El archivo sobre el cual se evaluará, será uno con extensión .ecore.

Ejemplos de estas reglas de buena formación seria:

```
context EClass
  inv WFR_1_Classifier:
  --[1] No Attributes may have same name within an EClass
   self.eStructuralFeatures -> select (p | p.oclIsKindOf(EAttribute) )
     ➔  forAll (p, q | p.name = q.name implies p = q)
```

Esta regla indica que dentro de una EClass no puede haber atributos con el mismo nombre. Esta regla se debe cumplir siempre, es decir, debe ser verdadera para todo metamodelo definido en todo momento.

Lo que se desprende como línea en los trabajos futuros es la posibilidad de definir restricciones sobre los modelos instancia de algún metamodelo definido. Actualmente ePlatero soporta la evaluación de reglas OCL a nivel modelo y metamodelo pero resta realizar la integración de dicho evaluador con el evaluador presentado en esta tesis.

8 Conclusiones y trabajos futuros

La Ingeniería de Software Conducida por Modelos (MDE) es un nuevo paradigma de software que propone mejorar la construcción de software a través de un proceso guiado por modelos. MDE promete una mejora de la productividad y de la calidad del software generado ya que reduce el salto semántico entre el dominio del problema y de la solución, reduciendo también los tiempos de desarrollo. La transformación entre modelos constituye el motor de MDE y de esta manera los modelos pasan de ser entidades meramente contemplativas a ser entidades productivas.

Cada vez son más las herramientas MDA que se están desarrollando pero cada una de ellas define su propia arquitectura de trabajo a la hora de especificar nuevas transformaciones. Esto constituye una fuerte desventaja, debido a que no se usan los estándares existentes; además, cada herramienta demanda un tiempo de estudio para que sea posible su utilización. Como consecuencia, tampoco es posible la reutilización de los modelos necesarios para la transformación: metamodelos, modelos y la definición de los pasos de transformación. Es decir, por cada herramienta, se deberá reingresar la información para especificar la transformación, los metamodelos y el modelo. Esto hace que la utilización de una nueva herramienta de transformación sea un paso difícil de tomar.

En esta tesis se han analizado las principales herramientas MDA disponibles actualmente. Como se mostró, algunas utilizan estándares, como MOF, XMI, etc., mientras que otras definen sus propios lenguajes. Todas diseñan e implementan su propio lenguaje de transformación, algunos de ellos basados en QVT. La principal dificultad que se ve en el uso de estas herramientas es, además de la necesidad de definir la transformación, es también la cuestión de definir los metamodelos de entrada y de salida, y del modelo de entrada. La falta de un editor gráfico que facilite estas tareas obstaculiza gravemente el uso de las herramientas.

Se mostró particular interés en ATL, ya que esta herramienta utiliza los estándares para la especificación de los metamodelos y modelos. También tiene la ventaja de permitir la definición y fácil uso de nuevos metamodelos.

Por otro lado, los lenguajes gráficos de modelado son ampliamente aceptados en la industria, sin embargo su falta de precisión ha originado la necesidad de utilizar otros lenguajes de especificación para definir restricciones adicionales, como es el caso de OCL. Con su uso se consiguen modelos precisos y completos del sistema en etapas tempranas del desarrollo. Sin embargo para estimular su uso en la industria es necesario contar con herramientas que permitan la edición y evaluación de las especificaciones expresadas en OCL.

En este trabajo hemos desarrollado un plugin para el entorno Eclipse que cumple los siguientes puntos:

- Permite especificar gráficamente metamodelos, brindando una forma más intuitiva para estas construcciones.

- Posee una base formal, ya que permite la edición y evaluación de restricciones definidas en OCL. Esto resulta útil para la evaluación de propiedades sobre los metamodelos definidos, tales como reglas de buena formación, métricas, etc.

- Además de la evaluación de propiedades a nivel meta-metamodelo, la herramienta permite la instanciación de un modelo a partir del metamodelo definido. Esto es de gran ayuda, ya que da lugar a un marco de trabajo mucho más amigable para los desarrolladores.

En base a lo planteado y a los resultados obtenidos en el presente trabajo, se proponen las siguientes líneas para trabajos futuros:

- Completar el prototipo presentado para permitir la transformación de modelos con el lenguaje estándar para transformaciones de modelos: QVT. Para esto será necesario implementar un nuevo plugin que permita definir la transformación en QVT. Al utilizar el lenguaje estándar, será posible la reutilización de la transformación y de los modelos.

- Permitir la evaluación de propiedades a nivel modelo. Para ello es necesario enriquecer al evaluador para que tenga la capacidad de evaluar propiedades en el nivel de metamodelo o M1, definidas junto con los metamodelos.

Para concluir, se considera que el aporte de este trabajo es valioso en el área de MDD, ya que, sobre la base de un análisis cuidadoso y extenso de las necesidades existentes y de la tecnología que le da soporte, hemos generado una propuesta que facilita el proceso de desarrollo de modelos y transformaciones (a través del uso de definiciones gráficas) y que incrementa su confiabilidad (a través de la aplicación OCL).

Por otro lado la propuesta se basa completamente en la aplicación de los estándares aceptados en MDD.

Si bien resta realizar experimentos de aplicación en casos industriales reales, a partir de los casos de estudio desarrollados se vislumbran su aplicabilidad y sus ventajas.

Referencias bibliográficas

[1] Akehurst D., Howells W., McDonald-Maier K. Kent Model Transformation Language. Model Transformation in Practice Workshop at MODELS 2005 Conference. Montego Bay, Jamaica. (2005).

[2] ATL Project - www.eclipse.org/m2m/atl/

[3] Booch,G., Rumbaugh,J. y Jacobson,I., The UML User Guide Addison Wesley Longman, Inc. (1998).

[4] Claudia Pons, Roxana Giandini, Gabriela Pérez, P. Pesce, V.Becker, J. Longinotti, J.Cengia. PAMPERO: Precise Assistant for the Modeling Process in an Environment with Refinement Orientation. Publication: Lecture Notes in Computer Science ISSN 0302-9743. volume 3297. Editors: N. Nunes, B. Selic, A. Silva and A. Toval.. . pp. 246 – 249, © Springer-Verlag Berlin Heidelberg 2005

[5] Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA: visual automated transformations for formal verification and validation of UML models. In: Proceedings 17th IEEE international conference on automated software engineering (ASE 2002), pp. 267–270, Edinburgh, UK. (2002).

[6] Cockburn, Alistair. Agile Software Development. Boston: Addison-Wesley, 2002.

[7] Eclipse Modeling Framework EMF. http://www.eclipse.org/modeling/emf/

[8] ePlatero. http://sol.info.unlp.edu.ar/eclipse. (2006).

[9] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, Timothy Grose. - Eclipse Modeling Framework (EMF) . Addison Wesley Professional.

[10] Freddy Allilaire, Jean Bézivin, Frédéric Jouault, Ivan Kurtev - ATL: Eclipse Support for Model Transformation - Eclipse Technology Exchange Workshop (ETX2006). (2006)

[11] Gerber, Anna - Raymond, Kerry, "MOF to EMF: There And Back Again", Proc. Eclipse Technology Exchange Workshop, OOPSLA 2003, Anaheim, USA, Oct 2003, pp 66-70

[12] Graphical Modeling Framework - www.eclipse.org/gmf/

[13] Jackson, Daniel, Shlyakhter, I. and Sridharan. A micromodularity Mechanism. In proceedings of the ACM Sigsoft Conference on the Foundation of Software Engineering FSE'01. (2001).

[14] Jouault F., Kurtev I. Transforming Models with ATL. Model Transformation in Practice Workshop at MODELS 2005 Conference. Montego Bay, Jamaica (2005).

[15] Kent Beck. Extreme Programming Explained: Embrace Change. Boston: Addison Wesley, 2000

[16] Kleppe, Anneke G. and Warmer Jos, and Bast, Wim. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[17] Lano K., Catalogue of Model Transformation. 2006, http://www.dcs.kcl.ac.uk/staff/kcl/tcat.pdf

[18] Lawley M., Steel J. Practical Declarative Model Transformation with TefKat. Model Transformation in Practice Workshop at MODELS 2005 Conference. Montego Bay, Jamaica. (2005).

[19] MDA Guide, v1.0.1, omg/03-06-01, June 2003. http://www.omg.org.

[20] Mellor, Stephen J. and Scott, Kendall and Uhl, Axel and Weise Dirk. MDA Distilled, Principles of Model_Driven Architecture. Addison-Wesley, 2004.

[21] Meta Object Facility (MOF) 2.0 Core Specification. OMG - (2005).

[22] MOF QVT final adopted specification. Technical Report ptc/05-11-01, OMG, 2005.

[23] MOFScript Home page - www.eclipse.org/gmt/mofscript/

[24] Object Constraint Language OCL 2.0. OMG Final Adopted Specification. Document ptc/03-10-14. (2003).

[25] OMG (Object Management Group) http://www.omg.org

[26] PAMPERO –

[27] Pons, Claudia. Formal Tools Supporting the Evolutionary Software Development Process. Eclipse Technology eXchange Meeting. http://www.cas.ibm.com/conferences/etx/. At International Conference on Software Engineering ICSE 2003, Portland Oregon. May 3-10, 2003.

[28] R.Giandini, G.Perez and C. Pons. A minimal OCL-based Profile for Model Transformation, VI Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento (JIISIC'07).ISBN: 978-9972-2885-2-4. Lima, Perú, febrero de 2007.

[29] Standard Profiles UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification (chapter 18). http://www.omg.org.

[30] The Eclipse Project. Home Page. Copyright IBM Corp, 2000-2005. http://www.Eclipse.org/.

[31] UML 1.4, Object Management Group, The Unified Modeling Language (UML) Specification – Version 1.4, en http://www.omg.or (1999).

[32] UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification. August 2003. http://www.omg.org.

Glosario de siglas y términos

En este Glosario se describen siglas y términos utilizados en el desarrollo de este trabajo. El contenido que sigue es simplemente, una ayuda rápida para ilustrar algunos conceptos.

AndroMDA
AndroMDA (pronunciado "Andrómeda") es un framework de código extensible asociado al paradigma MDA (Model Driven Architecture). Los modelos UML son transformados en componentes desplegables para la plataforma elegida (J2EE, Spring, .NET). Al contrario de otros entornos de desarrollo MDA, AndroMDA incluye un conjunto de cartuchos enfocados a los Kits de desarrollo actuales. AndroMDA también incluye un Kit para desarrollar sus propios cartuchos generadores de código o personalizar los existentes: el cartucho Meta. Utilizándolo, se puede construir un generador propio de código empleando una herramienta de UML. Debido a que su generador de código soporta plataformas actuales, se ha convertido en la principal herramienta de código abierto de MDA para el desarrollo de aplicaciones empresariales.

ArcStyler 5.5
ArcStyler es una de las herramientas MDA comerciales más extendida. Puede generar código a partir de modelos para cualquier plataforma como .NET o J2EE, siendo una herramienta genérica que nos permite transformaciones de modelo a código sin restricciones. También permite transformaciones entre modelos con la nueva herramienta, AIM, que incorpora esta versión. Soporta el lenguaje de modelado UML 1.4 para el diseño, aunque es independiente de cualquier versión UML ya que se apoya en otra herramienta que le proporciona dicha funcionalidad, MagicDraw. Se apoya en MOF para definir sus propios modelos y en XMI para almacenarlos, lo que le permite exportar e importar los distintos modelos usados. Además, contiene un repositorio de modelos al cual accede a través de JMI, pero esto no implica que no puedan añadírsele otros repositorios de modelos e incluso otras convenciones de interfaces de acceso. En general, la arquitectura de la herramienta es bastante flexible.

ASP
Active Server Pages (ASP) es una tecnología del lado servidor de Microsoft para páginas web generadas dinámicamente, que ha sido comercializada como un anexo a Internet Information Server (IIS). La tecnología ASP está estrechamente relacionada con el modelo tecnológico de su fabricante. Intenta ser solución para un modelo de programación rápida ya que programar en ASP es como programar en VisualBasic, pero con muchas limitaciones. Lo interesante de este modelo tecnológico es poder utilizar diversos componentes ya desarrollados como algunos controles ActiveX.

ATL
ATL (ATLAS Transformation Language) es un lenguaje de transformación de modelos y conjunto de herramientas desarrolladas por el Grupo ATLAS (INRIA & LINA). En el campo de Model-Driven Engineering (MDE), ATL provee formas de producir un conjunto de modelos destino, desde un conjunto de modelos fuente. Desarrollado sobre la plataforma Eclipse, el ATL Integrated Environnement (IDE) provee un número de herramientas estándar de desarrollo (syntax highlighting, debugger, etc.) que facilita el desarrollo de transformaciones en ATL. El proyecto ATL incluye también una librería de transformaciones ATL.

Eclipse
Eclipse es una plataforma de software de código abierto independiente de otras plataformas. Esta plataforma, típicamente ha sido usada para desarrollar entornos integrados de desarrollo (del inglés IDE), como el IDE de Java llamado Java Development Toolkit (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse (y que son usados también para desarrollar el mismo Eclipse). Sin embargo, también se puede usar para otros tipos de aplicaciones cliente. Eclipse es también una comunidad de usuarios, extendiendo constantemente las áreas de aplicación cubiertas. Un ejemplo es el recientemente creado Eclipse Modeling Project, cubriendo casi todas las áreas de Model Driven Engineering. Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente no lucrativa, que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

EMF - Eclipse Modeling Framework Project (EMF).
El proyecto EMF es un framework de modelado y facilidades de generación de código para la creación de herramientas de instalación y otras aplicaciones basadas en un modelo de datos estructurados. Desde una especificación de modelo descrito en XMI, EMF ofrece herramientas y soporte en tiempo de ejecución para producir un conjunto de clases Java para el modelo, junto con un conjunto de clases adaptadoras que permiten la visualización y la edición basada en comandos, del modelo, y un editor básico.

ePlatero
ePlatero, es un plug-in de código abierto para la plataforma Eclipse, desarrollado por nuestro grupo de investigación, que corre sobre el framework para metamodelado EMF.
Es una herramienta CASE educativa que soporta el proceso de desarrollo de software conducido por modelo utilizando notación gráfica y con fundamento formal. Puede interoperar con herramientas que soporten MDA.

FLASH
Adobe FLASH® (hasta 2005 Macromedia FLASH®) o FLASH® se refiere tanto al programa de edición multimedia como al reproductor de SWF (Shockwave FLASH) Adobe Flash Player, escrito y distribuido por Adobe, que utiliza gráficos vectoriales e imágenes ráster, sonido, código de programa, flujo de vídeo y audio bidireccional (el flujo de subida sólo está disponible si se usa conjuntamente con Macromedia Flash Communication Server). En sentido estricto, Flash es el entorno y Flash Player es el programa de máquina virtual utilizado para ejecutar los archivos generados con Flash.

GEF
GEF es el acrónimo de Graphical Editing Framework. Es un framework implementado para la plataforma Eclipse que ayuda en el desarrollo de componentes gráficos. Consiste de tres componentes principales, *draw2d* usado para los componentes visuales, Request/Commands usado durante la edición para crear pedidos y comandos capaces de rehacerse y deshacerse, y por último una paleta de herramientas para mostrar las opciones al usuario.

GMF
GMF es el acrónimo de Graphical Modeling Framework, que se traduce como Framework para modelados gráficos. Es un framework implementado para la plataforma Eclipse. Permite el desarrollo de editores gráficos y es un plugin que depende principalmente de EMF y GEF.

HTML
Es el acrónimo inglés de HyperText Markup Language, que se traduce al español como Lenguaje de Marcas Hipertextuales. Es un lenguaje de marcación diseñado para estructurar textos y presentarlos en forma de hipertexto, que es el formato estándar de las páginas web. Gracias a Internet y sus navegadores, el HTML se ha convertido en uno de los formatos más populares y fáciles de aprender que existen para la elaboración de documentos para web.

JMI
El Java Metadata Interface (JMI) en un estándar para la gestión de los metadatos. La especificación JMI permite la aplicación de una dinámica, independiente de la plataforma para la gestión de infraestructuras de la creación, el almacenamiento, el acceso, el descubrimiento, y el intercambio de metadatos. JMI se basa en la especificación MOF de OMG, un estándar de la industria que apoya la gestión de los metadatos. JMI define el estándar de interfaces Java para modelar estos componentes, y, por tanto, es independiente de la plataforma. JMI permite el descubrimiento, la búsqueda, el acceso y la manipulación de los metadatos, ya sea en tiempo de diseño o en tiempo de ejecución. La semántica de cualquier modelo de sistema puede ser totalmente descubierta y manipulada. JMI prevé

también intercambio de meta-metadatos a través de XML utilizando la especificación estándar XML Metadata Interchange (XMI).

JSP

JavaServer Pages (JSP) es una tecnología Java que permite generar contenido dinámico para web, en forma de documentos HTML, XML o de otro tipo. Esta tecnología es un desarrollo de la compañía Sun Microsystems. La Especificación JSP 1.2 fue la primera que ser liberó y en la actualidad está disponible la Especificación JSP 2.1. Las JSPs permiten la utilización de código Java mediante scripts. Además es posible utilizar algunas acciones JSP predefinidas mediante etiquetas. Estas etiquetas pueden ser enriquecidas mediante la utilización de Librerías de Etiquetas (Tag Libraries) externas e incluso personalizadas.

MDE

Acrónimo inglés de Model Driven Engeenering, en español se traduce como Ingeniería de Software Conducida por Modelos.
El paradigma MDE tiene dos ejes principales: - por un lado hace énfasis en la separación entre la especificación de la funcionalidad esencial del sistema y la implementación de dicha funcionalidad usando plataformas tecnológicas específicas. Por otro lado, en MDE los modelos son considerados los conductores primarios en todos los aspectos del desarrollo de software. MDE identifica dos tipos principales de modelos: modelos con alto nivel de abstracción e independientes de cualquier tecnología de implementación, llamados PIM y modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica, conocidos como PSM. Un PIM es transformado en uno o más PSMs, es decir que para cada plataforma tecnológica específica se genera un PSM específico.

MDD

Otro acrónimo relacionado a MDE es Model-Driven Development (MDD), que en español se traduce como Desarrollo de Software Conducida por Modelos. Es visto como un sinónimo de MDE, ambos describen la misma metodología de desarrollo de Software.

MDA

En español, Arquitectura conducida por modelos. Han surgido varios enfoques dentro del ámbito de MDE, pero sin duda la iniciativa más conocida y extendida es la MDA, acrónimo de Model Driven Architecture, presentada por el consorcio OMG (Object Management Group) en noviembre de 2000 con el objetivo de abordar los desafíos de integración de aplicaciones y los continuos cambios tecnológicos. MDA propone el uso de un conjunto de estándares (descriptos en este Glosario) como MOF, UML, JMI o XMI. Su objetivo es separar la especificación de la funcionalidad del sistema de su implementación sobre una plataforma concreta, por lo que se hace una distinción entre modelos PIM y modelos PSM.

MOF
El Meta Object Facility (MOF), es un estándar de OMG para MDD. La página oficial de referencia se puede encontrar en OMG's Meta Object Facility. MOF se originó en el Lenguaje Unificado de Modelado (UML); OMG tenía la necesidad de contar con una arquitectura de Metamodelado para definir el UML. MOF está diseñado como el nivel más abstracto de una arquitectura de cuatro capas o niveles. Proporciona un meta-metamodelo en la capa superior, denominado nivel M3. Este modelo M3 es el lenguaje utilizado por MOF para construir metamodelos, denominados modelos M2. El ejemplo más destacado de un modelo MOF de nivel M2, es el metamodelo UML, es decir el modelo que describe a UML. Estos modelos M2 describen los elementos del nivel M1, y por lo tanto describen modelos M1. Estas serían, por ejemplo, modelos escritos en UML. La última capa es el nivel M0 o capa de datos. Se utiliza para describir el mundo real (instancias de elementos M1).

NET
.NET es un proyecto de Microsoft para crear una nueva plataforma de desarrollo de software con énfasis en transparencia de redes, con independencia de plataforma y que permita un rápido desarrollo de aplicaciones. Basado en esta plataforma, Microsoft intenta desarrollar una estrategia horizontal que integre todos sus productos, desde el Sistema Operativo hasta las herramientas de mercado. .NET podría considerarse una respuesta de Microsoft al creciente mercado de los negocios en entornos Web, como competencia a la plataforma Java de Sun Microsystems.

OCL
Lenguaje de Restricciones para Objetos (OCL, por su sigla en inglés, Object Constraint Language) es un lenguaje declarativo para describir reglas que se aplican a metamodelos MOF, y a los modelos UML, desarrollado en IBM y en la actualidad parte del estándar UML. OCL inicialmente era sólo un lenguaje de especificación formal integrado a UML. Sin embargo, OCL puede ser usado con cualquier metamodelo MOF de OMG, incluyendo UML. El Object Constraint Language es un lenguaje de texto preciso que permite definir restricciones y consultas sobre expresiones de objetos de cualquier modelo o metamodelo MOF que de otra manera no pueden ser expresadas mediante la notación gráfica. OCL es un componente clave de la nueva propuesta estándar OMG para la transformación de los modelos, la especificación QVT. Muchos otros lenguajes de transformación de modelos como ATL, también están construidos utilizando OCL.

OMG
El Object Management Group u OMG (de su sigla en inglés Grupo de Gestión de Objetos) es un consorcio dedicado a la gestión y el establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como UML, XMI, CORBA. Es una organización no lucrativa que promueve el uso de tecnología orientada a objetos mediante guías y documentos de especificación de estándares independientes de las implementaciones de las diferentes empresas. El grupo está formado por

compañías y organizaciones de software como lo son: Hewlett-Packard (HP), IBM, Sun Microsystems, Apple Computer.

OMONDO
Omondo es una herramienta de modelado UML que se integra con Eclipse. Depende de otros plugins de Eclipse, como GEF, EMF y UML2. Permite relacionar el modelo UML con código Java, manteniendo la consistencia aun si se hacen cambios en el código. Es una herramienta comercial, cuyo código no esta disponible.

OptimalJ
OptimalJ es la herramienta que se puede considerar que mejor adapta la visión MDA, es decir en ella podemos encontrar los niveles bien diferenciados de PIM, PSM y código. Se trata básicamente de un entorno de desarrollo para aplicaciones empresariales que permite generar con rapidez aplicaciones J2EE completas a partir del modelo de alto nivel (PIM). Y precisamente es ahí donde se encuentra su mayor inconveniente, ya que al ser mono-plataforma obliga a sus clientes a tener un cierto dominio en dicha tecnología. Pero también precisamente por dedicarse exclusivamente a ese entorno, consigue adaptarse a los procesos de desarrollo de J2EE con modelos de forma sorprendente, generándonos a partir de un PIM una estructura de modelos PSM con sus puentes de comunicación que permiten construir aplicaciones web en muy poco tiempo. Además y en la misma línea implementa todo tipo de patrones para dicha plataforma y consigue un PSM y modelo de código de buena calidad.

PAMPERO
Pampero es un plugin desarrollado para la plataforma Eclipse. PAMPERO es una herramineta educacional para el proceso de desarrollo de software conducido por modelos usando notación gráfica. Este plugin es la versión anterior a ePlatero, estaba desarrollado para Eclipse 2.1. En el año 2004 fue el ganador del concurso internacional "ECI: internacional Challenge for Eclipse" (http://www.scs.carleton.ca/~deugo/ice/) organizado por IBM, en la categoría estudiantil.

PIM
Es el acrónimo inglés de Platform Independent Model, que se traduce al español como Modelo Independiente de la Plataforma. MDE identifica dos tipos principales de modelos: modelos con alto nivel de abstracción e independientes de cualquier tecnología de implementación, llamados PIM.

POSEIDON
Poseidon es una herramienta para modelos UML. Es una herramienta comercial, contiene los 9 diagramas de UML. Permite generar código Java, y exportar los diagramas a varios formatos.

PSM
Es el acrónimo inglés de Platform Specific Model, que se traduce al español como Modelo específico de la Plataforma modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica, conocidos como PSM. En MDE un PIM es transformado en uno o más PSMs, es decir que para cada plataforma tecnológica específica se genera un PSM específico.

QVT
En MDD, QVT (Query/ View/ Transformation) es un estándar para transformación de modelos definido por el OMG (Object Management Group). La especificación del lenguaje QVT tiene una naturaleza híbrida, declarativa/imperativa, con la parte declarativa dividida en una arquitectura de dos niveles. Esta especificación define tres paquetes principales, uno por cada lenguaje definido: QVTCore, QVTRelation y QVTOperational. Estos paquetes principales se comunican entre sí y comparten otros paquetes intermedios.

RDBMS
Un RDBMS es un Sistema Administrador de Bases de Datos Relacionales. RDBMS viene del acrónimo en inglés Relational Data Base Management System. Los RDBMS proporcionan el ambiente adecuado para gestionar una base de datos.

UML
Lenguaje Unificado de Modelado (UML, por su sigla en inglés, Unified Modeling Language) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad. Es el lenguaje estándar oficial, respaldado por el OMG (Object Management Group). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software. UML ofrece un estándar para describir la estructura y el comportamiento del sistema, incluyendo aspectos conceptuales tales como procesos de negocios y funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de software reutilizables.

XML
Es el acrónimo inglés de eXtensible Markup Language («lenguaje de marcas extensible»), es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C). Permite definir la gramática de lenguajes específicos. Por lo tanto XML no es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades. Algunos de estos lenguajes que usan XML para su definición son XHTML, SVG, MathML.

XMI
XMI o XML Metadata Interchange (XML de Intercambio de Metadatos) es una especificación para el Intercambio de Diagramas. La especificación para

el intercambio de diagramas fue escrita para proveer una manera de compartir modelos UML entre diferentes herramientas de modelado. En versiones anteriores de UML se utilizaba un esquema XML para capturar los elementos utilizados en el diagrama; pero este esquema no decía nada acerca de cómo el modelo debía graficarse. Para solucionar este problema la nueva Especificación para el Intercambio de Diagramas fue desarrollada mediante un nuevo esquema XML que permite construir una representación SVG (Scalable Vector Graphics).

Anexo I

1. Gramática de OCL 2.0

La gramática que se muestra a continuación se puede emplear para generar un parser de OCL 2.0.

```
packageDeclaration      ::= 'package' pathname contextDeclList
'endpackage' | contextDeclList
contextDeclList         ::= contextDeclaration*
contextDeclaration ::= propertyContextDecl | classifierContextDecl |
operationContextDecl
propertyContextDecl     ::= 'context' pathname '::' simpleName ':' type
initOrDerValue+
initOrDerValue          ::= 'init' ':' oclExpression | 'derive' ':'
oclExpression
classifierContextDecl   ::= 'context' [simpleName':'] pathname
invOrDef+
invOrDef          ::= 'inv' [simpleName] ':' oclExpression   | 'def'
[simpleName] '=' defExpression
defExpression           ::= simpleName ':' type '=' oclExpression  |
operation '=' oclExpression
operationContextDecl    ::= 'context' operation prePostOrBodyDecl+
prePostOrBodyDecl       ::= 'pre' [simpleName] ':' oclExpression  |  'post'
[simpleName] ':' oclExpression |
                          'body' [simpleName] ':' oclExpression
operation         ::= pathName '(' [variableDeclarationList] ')' [':' type]
variableDeclarationList   ::= variableDeclaration ( ',' variableDeclaration )*
variableDeclaration       ::= simpleName [':' type] ['=' oclExpression]
type              ::= pathname | collectionType |  tupleType
collectionType          ::= collectionKind '(' type ')'
tupleType         ::= 'TupleType' '(' variableDeclarationList ')'
oclExpression           ::= logicalImpliesExpression | letExpression
letExpression           ::= 'let' variableDeclarationList 'in' oclExpression
logicalImpliesExpression  ::=logicalXorExpression |
            logicalImpliesExpression 'implies' logicalXorExpression
logicalXorExpression    ::= logicalOrExpression |  logicalXorExpression
'xor' loficalOrExpression
logicalOrExpression     ::= logicalAndExpression |   loficalOrExpression
'or' logicalAndExpression
logicalAndExpression    ::= relationalExpression |  logicalAndExpression
'and' relationalExpression
relationalExpression    ::= addExpression |  addExpression
relationalOperator addExpression
relationalOperator      ::=  < | <= | > | >= | <> | =
addExpression           ::= mulExpression |  addExpression  '+'
mulExpression
                  |  addExpression '-' mulExpression
```

```
mulExpression              ::= unaryExpression |  mulExpression '*'
unaryExpression
                       |  mulExpression '/' unaryExpression
unaryExpression            ::= primaryExpression |  '-' unaryExpression |
'not' unaryExpression
primaryExpression          ::=  literalExp | '(' oclExpression ')' |
postfixExpression | ifExpression
                           |  propertyCall
ifExpression          ::= 'if' oclExpression 'then' oclExpression 'else'
        oclExpression 'endif'
postfixExpression   ::= primaryExpression '.'propertyCall
                          |  primaryExpression '->' propertyCall
                          |  primaryExpression '^' messageCall
                          | primaryExpression '^^' messageCall
propertyCall                ::= simpleName [@pre] [qualifiers]
[propertyCallParams]
propertyCallParams         ::= '(' [declarator] oclExpression ['','
ocExpression]*   ')'
declarator              ::= simpleName ['','' simpleName]* [':'type ]
                  [';' simpleName ':' type '=' oclExpression] '|'
messageCall                 ::= pathName '('  [messageCallArgument] ['','
messageCallArgument]* ')'
              messageCallArgument ::= '?' [:type] oclExpression
qualifiers              ::= '[' oclExpression ['','' oclExpression]*']'
formalParameter             ::= simpleName ':' type
literalExp              ::= collectionLiteralExp | tupleLiteralExp |
primitiveLiteralExp
collectionLiteralExp        ::= collectionKind ''{'' collectionLiteralParts ''}'' |
collectionKind ''{'' ''}''
collectionKind              ::= ''Set'' | ''Bag'' | ''Sequence'' | ''Collection'' |
''OrderedSet''
collectionLiteralParts      ::= collectionLiteralPart ( '','' collectionLiteralPart )*
                              | oclExpression | collectionRange
collectionRange             ::= oclExpression ''..'' oclExpression
tupleLiteralExp             ::= ''Tuple'' '' variableDeclarationList ''
primitiveLiteralExp         ::= integer | real | string | ''true'' | ''false''
pathname              ::= simpleName | pathName ''::'' simpleName
integer                     ::= [0-9]+
real                  ::= integer[.]integer[eE][+-]?integer | integer[eE][+-
]?integer | integer[.]integer
string                      ::= ['']['^'']*['']
simpleName          ::= [a-zA-Z ][a-zA-Z0-9 ]*
```

2. Tipos básicos de OCL

En este anexo se incluye los tipos básicos de OCL con sus respectivas operaciones. Toda implementación de OCL debe incluir esta librería.

Real: Este tipo representa el concepto matemático de real.

Operación	Descripción
+ (r : Real) : Real	Retorna la suma del receptor y el argumento, r. La suma es un nuevo Real.
- (r : Real) : Real	Retorna la diferencia entre el receptor y el argumento, r. La diferencia es un nuevo Real.
* (r : Real) : Real	Retorna el resultado de multiplicar al receptor por el argumento, r. El resultado es un nuevo Real.
- : Real	Retorna un Real que es la negación del receptor.
/ (r : Real) : Real	Retorna el valor de dividir el receptor por el argumento, r.
abs() : Real	Retorna el valor absoluto del receptor.
floor() : Integer	El entero mayor que es menor o igual al receptor.
round() : Integer	El entero que es más acotado al receptor. Cuando hay dos enteros, el más grande.
max(r : Real) : Real	El máximo del receptor y el argumento.
min(r : Real) : Real	El mínimo del receptor y el argumento, r.
< (r : Real) : Boolean	Retorna true si el receptor es menor que el argumento y false en caso contrario.
> (r : Real) : Boolean	Retorna true si el receptor es mayor que el argumento y false en caso contrario.
<= (r : Real) : Boolean	Retorna true si el receptor es menor o igual que el argumento y false en caso contrario.
>= (r : Real) : Boolean	Retorna true si el receptor es mayor o igual que el argumento y false en caso contrario.

Tabla 1–Operaciones para los reales

Integer: Este tipo representa el concepto matemático de entero. La superclase de Integer es Real, para que cada uno de los parámetros del tipo Real, pueda usar un entero como dicho parámetro.

Operación	Descripción
+ (i : Integer) : Integer	Retorna la suma del receptor y el argumento, r. La suma es un nuevo Integer.
- (i : Integer) : Integer	Retorna la diferencia entre el receptor y el

	argumento, r. La diferencia es un nuevo Integer.
* (i : Integer) : Integer	Retorna el resultado de multiplicar al receptor por el argumento, r. El resultado es un nuevo Integer.
- : Integer	Retorna un Integer que es la negación del receptor.
/ (i : Integer) : Real	Retorna el valor de dividir el receptor por el argumento, r.
abs() : Integer	Retorna el valor absoluto del receptor.
div(i : Integer) : Integer	El número de veces que i se ajusta completamente dentro del receptor.
mod(i : Integer) : Integer	El resto de la división entera.
max(i : Integer) : Integer	El máximo del receptor y el argumento i.
min(i : Integer) : Integer	El mínimo del receptor y el argumento, r.

Tabla 2–Operaciones para los enteros

String: El tipo String representa una secuencia de caracteres que pueden ser ASCII o Unicode.

Operación	Descripción
size() : Integer	Retorna el número de caracteres del receptor.
concat(s : String) : String	Retorna un nuevo String formado por la concatenación del receptor y el argumento.
substring(lower : Integer, upper : Integer) : String	Retorna un nuevo String que consta del substring del receptor. Los argumentos indican los caracteres a tomar.
toInteger() : Integer	Convierte al receptor en un valor entero.
toReal() : Real	Convierte al receptor en un valor real.

Tabla 3–Operaciones para los Strings

Boolean: El tipo Boolean representa los valores true / false.

Operación	Descripción
or (b : Boolean) : Boolean	True si el receptor o el argumento son true.
xor (b : Boolean) : Boolean	True si el receptor o b son true, pero no ambos.
and (b : Boolean) : Boolean	True si el receptor y b son true.
not : Boolean	True si el receptor es false y false en caso contrario.
implies (b : Boolean) : Boolean	True si el receptor es false, o si el receptor es true y b es true.

Tabla 4–Operaciones para los boolean

Collection: A continuación se detallan las operaciones de colecciones predefinidas.

Operación	Descripción
col->size():Integer	El número de elementos en la colección.
col->includes(object: T): Boolean	True si el objeto está incluido en la Colección.
col->excludes(object: T): Boolean	True si el objeto no está incluido en la colección.
col->count(object: T): Boolean	El número de veces que object está presente en la colección.
col->includesAll(c2: Collection(T)):Boolean	Verifica si la colección col contiene todos los elementos de c2.
col->includesAll(c2: Collection(T)):Boolean	Verifica que la colección col no contenga ninguno de los elementos de c2.
col->isEmpty(): Boolean	True si la colección es vacía.
col->notEmpty():Boolean	True si la colección no es vacía.
col->sum():T	La suma de todos los elementos de la colección. Los elementos deben ser de un tipo que soporte la operación +.
col->product(c2: Collection(T2)):Set(Tuple(first:T, second:T2))	El producto cartesiano de la colección col y c2.
col->exists(iterators | body):Boolean	True si la expresión body se evalúa a true al menos para un elemento de la colección.
col->forAll(iterators | body):Boolean	True si la expresión body se evalúa a true para cada elemento de la colección, false en caso contrario
col->isUnique (iterators | body):Boolean	True si el valor resultante de evaluar la expresión body es único para cada elemento de la colección.
col->any(iterator | body) = T	Retorna cualquier elemento de la colección que al evaluar la expresión body sea verdadera. Si no hay ningún elemento que cumpla con dicha condición se retorna null.
col->one(iterator | body) = T	True si hay un solo elemento de la colección para el cual la expresión body es verdadera, false en caso contrario.

Tabla 5 –Operaciones para Collection

Set: A continuación se detallan las operaciones para sets predefinidas.

Operación	Descripción
col->union(s: Set(T)):Set(T)	La unión de col y s.
col->union(bag: Bag(T)):Bag(T)	La unión de col y bag.
col->=(s:Set(T)):Boolean	True si el receptor y s tienen los mismos elementos.
col->intersection(s: Set(T)):Set(T)	La intersección de col y s.

col->intersection(bag: Bag(T)):Set(T)	La intersección de col y bag.
col->-(s: Set(T)): Set(T)	Los elementos de col que no están en s.
col->including(object:T):Set(T)	Retorna un conjunto que contiene todos los elementos de col más object.
col->excluding(object:T):Set(T)	El conjunto que contiene todos los elementos de col menos object.
col-> symmetricDiference(s:Set(T)):Set(T)	El conjunto que contiene todos los elementos que están en col o en s, pero no en ambos.
col->count(object:T):Integer	El número de ocurrencias de object en col.
col->flatten():Set(T)	Si el tipo de elemento no es un tipo colección el resultado es col, de otro modo el resultado es una colección que contiene todos los elementos de todos los elementos de col.
col->asSet():Set()	Un conjunto idéntico a col. Esta operación existe por razones de conveniencia.
col->asOrderedSet():OrderedSet(T)	Un OrderedSet que contiene todos los elementos de col.
col->asSequence():Sequence(T)	Una secuencia que contiene todos los elementos de col.
col->asBag():Bag(T)	El Bag que contiene todos los elementos de col.
col->select(iterator | body) =Set(T)	Retorna un subconjunto de la colección para el que la expresión body es verdadera.
col->reject(iterator | body) =Set(T)	El subconjunto de la colección para el que la expresión body es falso.
col->collect(iterators | body) =Bag(T)	El Bag de elementos con los resultados de aplicar la expresión body a cada miembro de la colección.
col->sortedBy(iterator | body) =OrderedSet(T)	Retorna el OrderedSet que contiene todos los elementos de la colección. El elemento para el que la expresión body tiene el valor más bajo es el primer elemento de la colección resultante, y así sucesivamente. El tipo de la expresión body debe soportar la operación <. Esta operación debe retornar un valor del tipo Boolean y deben ser transitiva si a <b y b <c entonces a <c.

OrderedSet: A continuación se detallan las operaciones para OrderedSet predefinidas.

Operación	Descripción
col->append(object:T):OrderedSet(T)	El conjunto de elementos, que consiste de todos los elementos de col, seguidos de object.
col->prepend(object:T):OrderedSet(T)	La secuencia que consiste en object, y todos los elementos de col.
col->insertAt(index: Integer, object:T):OrderedSet(T)	El conjunto que consiste de col pero object se inserta en la posición index.
col->subOrderedSet(lower:Integer, upper: Integer):OrderedSet(T)	El subconjunto de col que comienza en la posición lower y finaliza en la posición upper.
col->at(i:Integer): T	El i-ésimo elemento de col.
col->indexOf(obj:T):Integer	El índice del objeto obj en la secuencia.
col->first():T	El primer elemento en col.
col->last():T	El último elemento en col.

Tabla 7–Operaciones para OrderedSet

Bag: A continuación se detallan las operaciones de bag predefinidas.

Operación	Descripción
col->=(bag:Bag(T)):Boolean	True si col y bag tienen los mismos elementos.
col->union(bag: Bag(T)):Bag(T)	La unión de col y bag.
col->union(set: Set(T)):Bag(T)	La unión de col y set.
col->intersection(bag: Bag(T)):Bag(T)	La intersección de col y bag.
col->intersection(set: Set(T)):Set(T)	La intersección de col y set.

col->including(object:T):Bag(T)	El bag que contiene todos los elementos de col más object.
col->excluding(object:T):Bag(T)	El bag que contiene todos los elementos de col menos todas las ocurrencias de object.
col->count(object:T):Integer	El número de ocurrencias de object en col.
col->flatten():Bag(T2)	Si el tipo de elemento no es un tipo colección el resultado es col, de otro modo el resultado es el bag que contiene todos los elementos de todos los elementos de col
col->asBag():Bag(T)	Un Bag idéntico a self. Esta operación existe por razones de conveniencia.
col->asSequence():Sequence(T)	Una secuencia que contiene todos los elementos de col.
col->asSet():Set(T)	El conjunto que contiene todos los elementos de col.
col->asOrderedSet():OrderedSet(T)	El conjunto ordenado que contiene todos los elementos de col.
col->select(iterator | body) =Bag(T)	Retorna un sub-bag de la colección para el que la expresión body es verdadera.
col->reject(iterator | body) =Bag(T)	El sub-bag de la colección para el que la expresión body es falso.
col->collect(iterators | body) =Bag(T)	El Bag de elementos con los resultados de aplicar la expresión body a cada miembro de la colección.
col->sortedBy(iterator | body) =Sequence(T)	Retorna la secuencia que contiene todos los elementos de la colección. El elemento para el que la expresión body tiene el valor más bajo es el primer elemento de la colección resultante, y así sucesivamente. El tipo de la expresión body debe soportar la operación <. Esta operación debe retornar un valor del tipo Boolean y deben ser transitiva

	si a <b y b <c entonces a <c.

Tabla 8–Operaciones para Bag

Sequence: A continuación se detallan las operaciones de sequence predefinidas.

Operación	Descripción
col->count(object:T):Integer	El número de ocurrencias de object en el receptor.
col->=(s:Sequence(T)):Boolean	True si col consiste de todos los elementos de s en el mismo orden.
col->union(s:Sequence(T)): Sequence(T)	La secuencia que consiste de todos los elementos de col, seguido por todos los elementos en s
col->flatten():Sequence(T2)	Si el tipo de elemento no es un tipo colección el resultado es la misma secuencia col, de otro modo el resultado es la secuencia que contiene todos los elementos de todos los elementos de col.
col->append(object:T):Sequence(T)	La secuencia de elementos que consiste de todos los elementos de col seguidos de object.
col->prepend(object:T):Sequence(T)	La secuencia de elementos que consiste de object, y todos los elementos de col
col->insertAt(index: Integer, object: T):Sequence(T)	La secuencia que consiste de col con object insertado en la posición index.
col->subSequence(lower:Integer, upper: Integer):Sequence(T)	La subsecuencia de col que comienza en la posición lower y finaliza en la posición upper.
col->at(i:Integer): T	El i-ésimo elemento de la secuencia col.
col->indexOf(obj:T):Integer	El índice del objeto obj en la secuencia.
col->first():T	El primer elemento en col.
col->last():T	El último elemento en col.
col->including(object:T):Sequence(T)	La secuencia que contiene todos los elementos de col más object agregado como último elemento.
col->excluding(object:T):Sequence(T)	La secuencia que contiene todos los elementos de col menos todas las

	ocurrencias de object.
col->asBag():Bag()	El bag que contiene todos los elementos de self, incluyendo duplicados.
col->asSequence():Sequence(T2)	La secuencia idéntica a col. Esta operación existe por razones de conveniencia.
col->asSet():Set(T)	El conjunto que contiene todos los elementos de col, con duplicados removidos.
col->asOrderedSet():OrderedSet(T)	El conjunto ordenado que contiene todos los elementos de col, en el mismo orden, y con duplicados removidos.
col->select(iterator | body) =Sequence(T)	Retorna un subsecuencia de la colección para el que la expresión body es verdadera.
col->reject(iterator | body) = Sequence (T)	La subsecuencia de la colección para el que la expresión body es falso.
col->collect(iterators | body) = Sequence (T)	La subsecuencia de elementos con los resultados de aplicar la expresión body a cada miembro de la colección.
col->sortedBy(iterator | body) =Sequence(T)	Retorna la secuencia que contiene todos los elementos de la colección. El elemento para el que la expresión body tiene el valor más bajo es el primer elemento de la colección resultante, y así sucesivamente. El tipo de la expresión body debe soportar la operación <. Esta operación debe retornar un valor del tipo Boolean y deben ser transitiva si a <b y b <c entonces a <c.

Tabla 9–Operaciones para Sequence

Anexo II

En el CD que se adjunta, puede verse los siguientes archivos:

- jre-1_5_0_12-windows-i586-p.exe
- eclipse.zip

1. Como instalar ePlatero

1- Instalar la máquina virtual de Java (JVM) versión 1.5

Lo primero que hay que instalar para poder ejecutar Eclipse es una maquina virtual de Java. Para esto, simplemente clickear el archivo jre-1_5_0_12-windows-i586-p.exe y seguir los pasos para su instalación.

2- Como instalar eclipse y los plugins utilizados

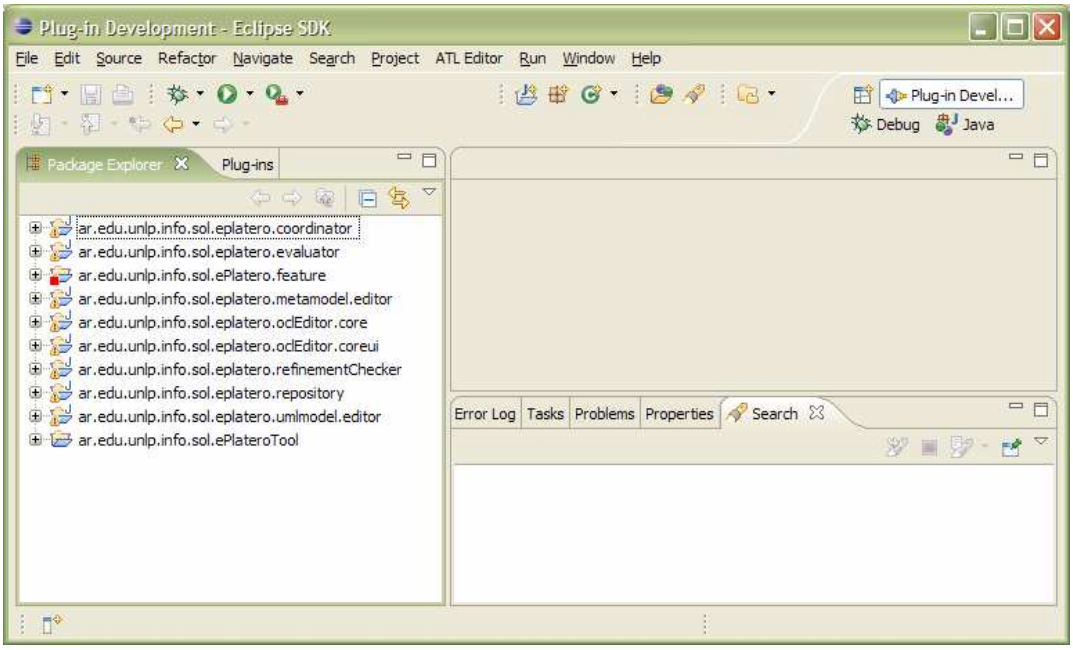
El CD incluye un archivo eclipse.zip que contiene los archivos necesarios para ejecutar Eclipse, y los plugins utilizados ya instalados en el ambiente. Simplemente hay que descomprimir este archivo en el disco rígido. En los pasos siguientes se supondrá que se hizo en el disco c: \carpeta-home.

Para asegurar que Eclipse compile con la versión 1.5 de la JVM, dentro de la carpeta eclipse, clickear con el botón derecho sobre el archivo c: \carpeta-home\ePlatero, y en la propiedad Target indicar donde esta instalada la JVM del paso 1.

Por ejemplo:
C: \carpeta-home \eclipse\eclipse.exe –clean –vm "C:\Program Files\Java\jre1.5.0_07\bin\javaw.exe"

3- Inicializar Eclipse

Simplemente hacer click sobre el archivo ePlatero.lnk lo cual inicializará el entorno Eclipse. Se abrirá una ventana popup para ingresar la ruta al workspace. Elegir la carpeta c:\carpeta-home\eclipse\workspace.

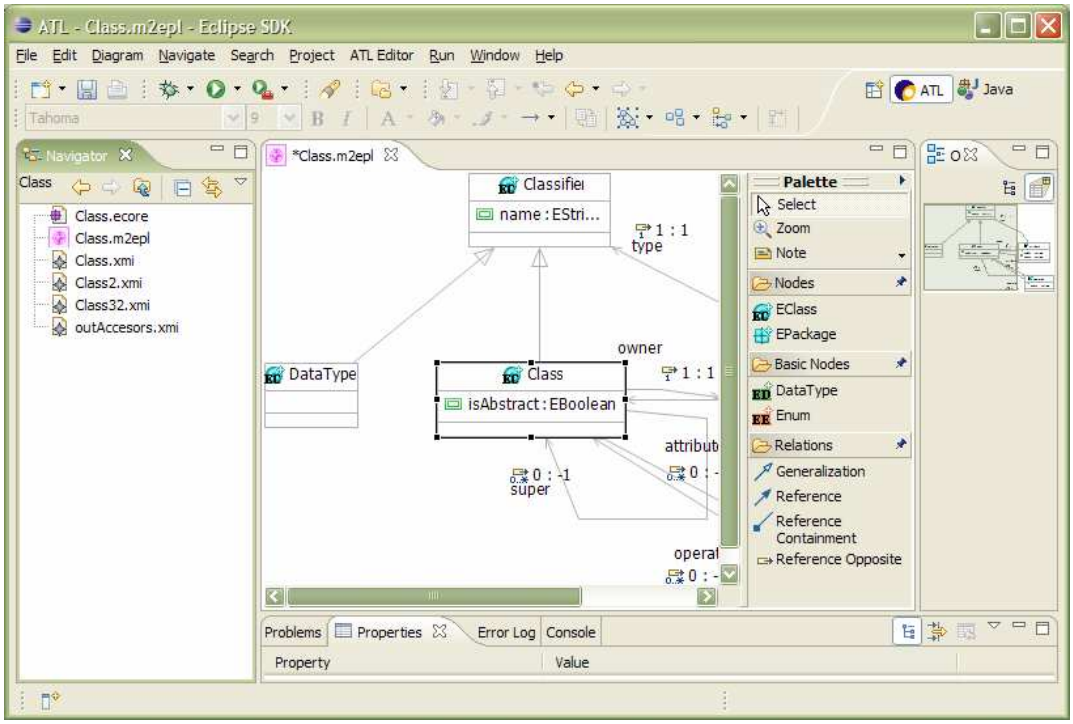


Arriba se ven los plugins desarrollados en ePlatero donde puede examinarse el código fuente de los mismos. La extensión del presente trabajo abarca los plugins
ar.edu.unlp.info.sol.eplatero.metamodel.editor
ar.edu.unlp.info.sol.eplatero.evaluator

El resto de los plugins corresponde a la versión actual de ePlatero.

4- Iniciar la versión runtime de Eclipse.
Para ver los plugins definidos en la figura anterior funcionando, es necesario iniciar la versión runtime. Para esto, simplemente clickear sobre el icono Run

Cuando se ejecuta el runtime de Eclipse, puede verse la siguiente pantalla:



En el árbol del lado izquierdo de la pantalla hay dos proyectos para navegar:

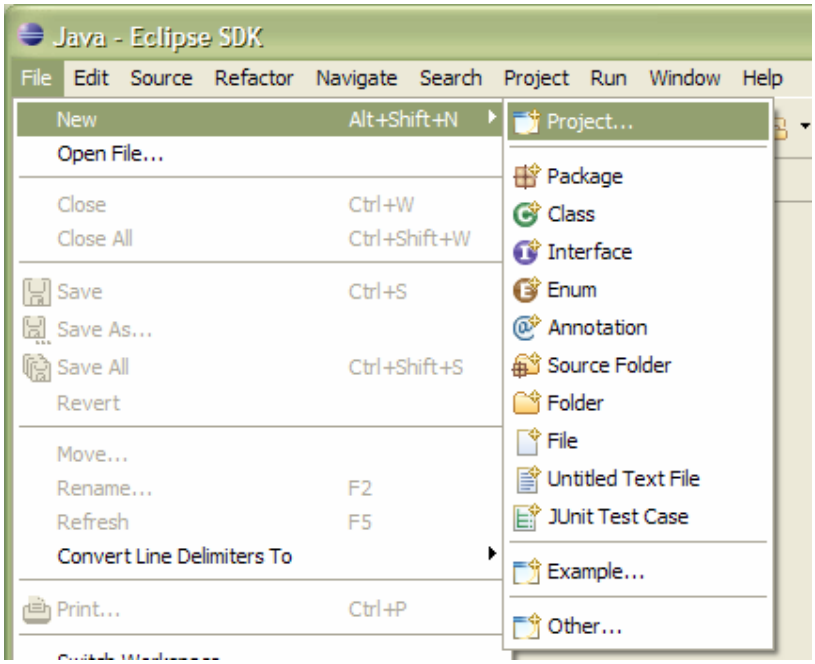
- **CLass2Relational:** este proyecto define la transformación para pasar un subconjunto de elementos UML al modelo relacional. Contiene los siguientes archivos y carpetas:
  - **ATLFile:** Dentro de la carpeta ATLFile se encuentra el archivo con extensión .atl donde se define la transformación de un modelo de clases al modelo relacional.
  - **Class**: Dentro de la carpeta Class se encuentra el metamodelo (archivo Class.m2epl y Class.ecore) y el modelo instanciado a partir de este metamodelo (archivo Class.xmi)
  - **LaunchConfigurationScreen**: contiene unas imágenes para configurar la ejecución de la transformación.
  - **Relational**: dentro de la carpeta Relational se encuentra el metamodelo relacional.
  - **salida**.txt: es la salida luego de ejecutar la transformación.

- **UML2GetterSetter:** este proyecto define la transformación para que a partir de un diagrama de clases, encapsule los atributos, es decir, cambie los atributos públicos a privados, y agregue un método get y un método set para modificar ese atributo.
  Contiene los siguientes archivos y carpetas:
  - **PublicToPrivate.atl:** define la transformación mencionada.
  - **MetamodeloUML.m2epl y MetamodeloUML.ecore**: el metamodelo para un subconjunto de elementos de UML
  - **Metamodelo**.xmi: el modelo instanciado a partir del metamodelo definido anteriormente.
  - **LaunchConfigurationScreen**: contiene unas imágenes para configurar la ejecución de la transformación.
  - **salida**.txt: es la salida luego de ejecutar la transformación.

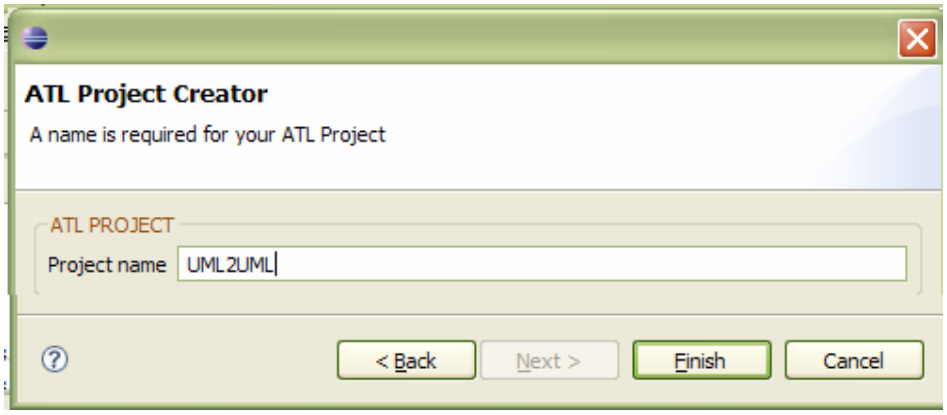
2. User Guide

En las siguientes secciones se explica brevemente como crear un proyecto ATL, como crear la transformación, los metamodelos, modelos y como ejecutar una transformación sencilla.
En este ejemplo se quiere, dado un modelo UML, convertirlo a otro modelo UML donde por cada atributo del primero se le agreguen dos operaciones: su getter y su setter.
Para ello, hay que crear primero un proyecto ATL donde definir el archivo de transformación. Luego hay que crear el metamodelo UML, el modelo y ejecutar la transformación. Los siguientes puntos ilustran paso a paso como llevar a cabo esta ejecución.

1- Crear un proyecto ATL que permita una transformación
Primero es necesario crear un proyecto ATL.



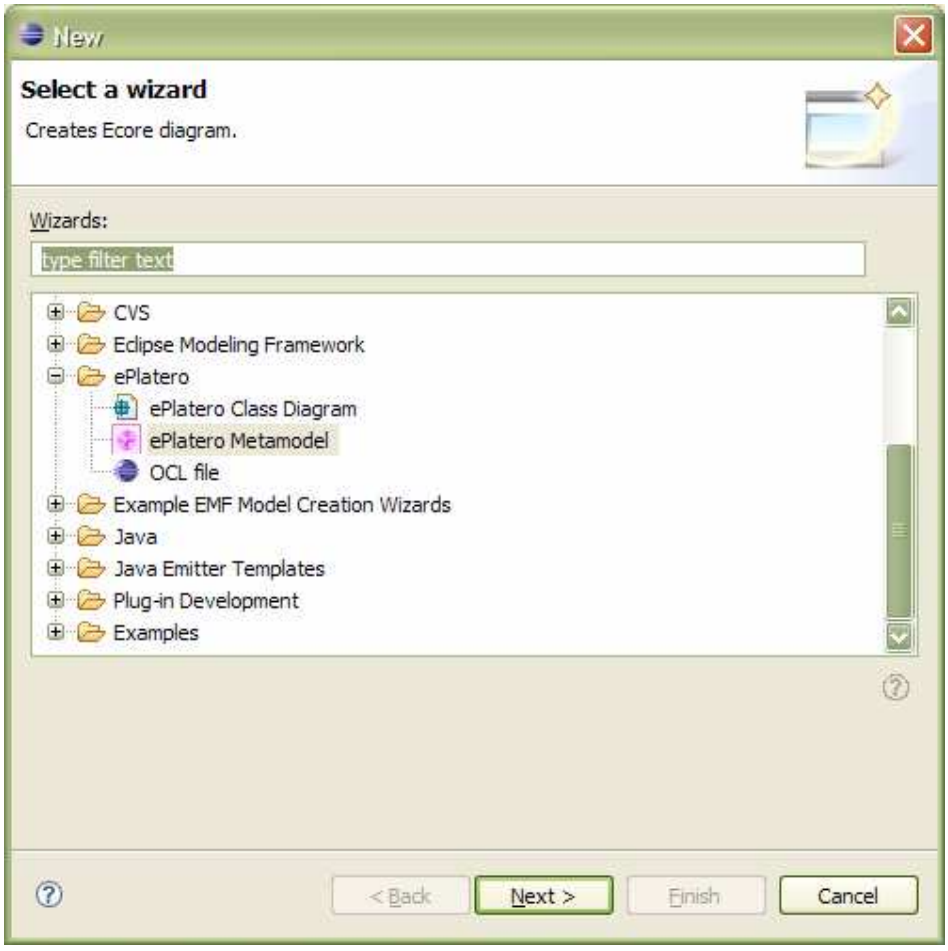
Y seleccionar dentro de la categoría "ATL", la opción ATL Project. Esto inicializa el wizard para la creación del proyecto ATL.



Luego como nombre "UML2UML" y elegir la opción "Finish". Con esto quedará creado el proyecto ATL.

2- Como crear los metamodelos de entrada y salida

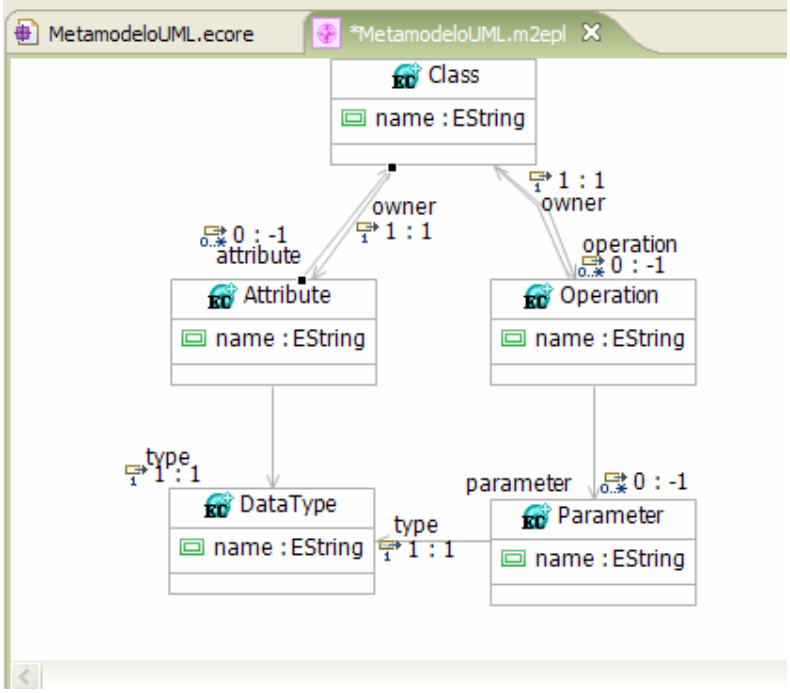
Se puede crear un metamodelo desde cero, a través del wizard en File>>New>>Other>>ePlatero>>ePlatero Metamodel



Al seleccionar la opción "Next" se abre una ventana para ingresar el nombre del metamodelo y el proyecto en el cual se quiere definir.

Como nombre del metamodelo escribir "MetamodeloUML" y como nombre del proyecto seleccionar el creado anteriormente "UML2UML".

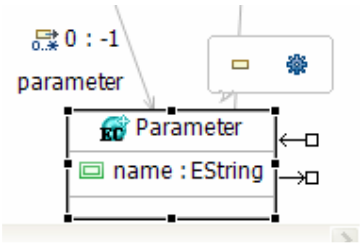
Se abrirá el editor gráfico de metamodelos. A modo de ejemplo ingresar un metamodelo simple, como el que se muestra en la siguiente figura:



Para crear elementos, simplemente arrastrarlos desde la paleta hasta el editor. Todos los cambios hechos con el editor gráfico se mantienen consistentes con la definición del metamodelo (archivo con extensión .ecore).

Para cambiar las propiedades de los elementos, hay que seleccionarlos y editar sus propiedades en la vista *Property*.

Sobre la clase se inicia un menú contextual que permite crear EAtributos y EOperaciones para una clase.



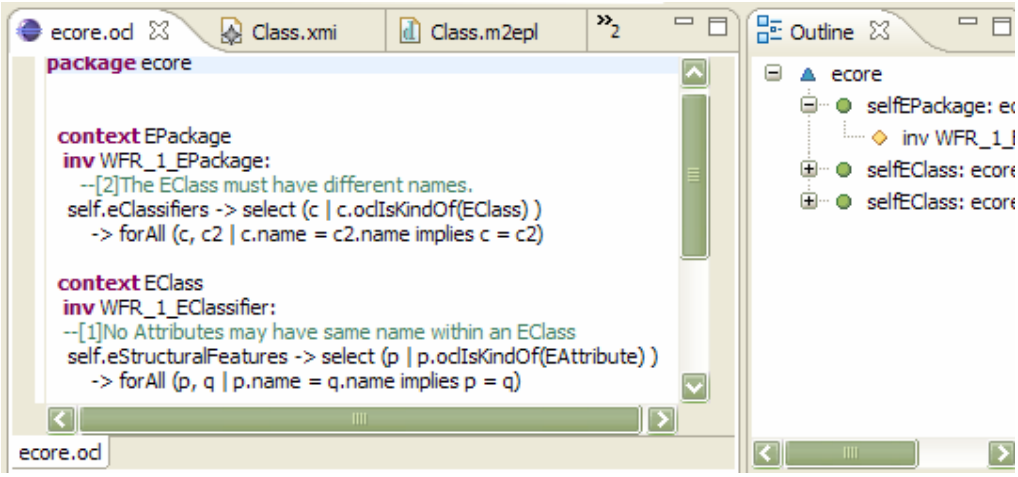
Crear cinco EClass, y nombrarlas como "Class", "Attribute". "Operation", "DataType" y "Parameter". A cada una de ellas agregarles un EAttributo llamado nombre, por defecto el tipo del atributo será EString.

Las relaciones entre clases son:

- Entre Class y Attribute: una relación contenida (Containment), de 0 a muchos (se define con el limite superior en -1 desde la vista de propiedades). Una relación similar entre Class y Operation.
- Entre Attribute y DataType, una relación no contenida, con multiplicidad 1. Una relación similar entre Parameter y DataType.
- Entre Operation y Parameter crear una referencia navegable, con multiplicidad 0 a muchos.
- Por último, entre "Operation" y "Class" y entre "Attribute" y "Class" definir dos operaciones no contenidas, con nombre "owner", de multiplicidad 1. Definirlas como opuestas (EOpposite) de las definidas anteriormente entre las mismas clases.

3- Escribir restricciones sobre el metamodelo

El editor OCL se usa para editar reglas a ser evaluadas sobre el metamodelo. Permite escribir restricciones, provee sintaxis coloreada (syntax highlighting), asistencia en la escritura de las reglas, y corrección de errores. Tiene un outline que muestra la estructura del archivo OCL.



Para crear un archivo OCL, hay que usar el wizard para OCL que se inicial en File>>New>> Other>>ePlatero>>OCL File

Nombrar el archivo como ecore.ocl y agregarle el siguiente texto

```
package ecore
  context EPackage
   inv WFR_1_EPackage:
     --[2]The EClass must have different names.
    self.eClassifiers -> select (c | c.oclIsKindOf(EClass) )
       -> forAll (c, c2 | c.name = c2.name implies c = c2)

  context EClass
   inv WFR_1_EClassifier:
   --[1]No Attributes may have same name within an EClass
    self.eStructuralFeatures -> select (p | p.oclIsKindOf(EAttribute) )
       -> forAll (p, q | p.name = q.name implies p = q)

  context EClass
   inv WFR_2_EClassifier:
     --[2]The eReferences must have different names as role names.
    self.eStructuralFeatures -> select (p | p.oclIsKindOf(EReference) )
       -> forAll (p, q | p.name = q.name implies p = q)
```

```
endpackage
```

Para evaluar las restricciones definidas en el archivo OCL sobre el metamodelo, desde la vista de recursos, a la izquierda, clickear con el botón derecho sobre el archivo ecore.ocl y seleccionar la opción "Evaluate". Esto abrirá una ventana para seleccionar sobre que modelo (o metamodelo) evaluar el archivo. Seleccionar el archivo MetamodeloUML.ecore.

Si existen errores en la definición del metamodelo, se mostrarán en la vista de problemas, donde además se especificará el error para ayudar a su corrección.

4- Escribir una transformación en ATL

Para crear una transformación hay que seleccionar el wizard en File>>New>>ATL File y definirla dentro de este archivo.

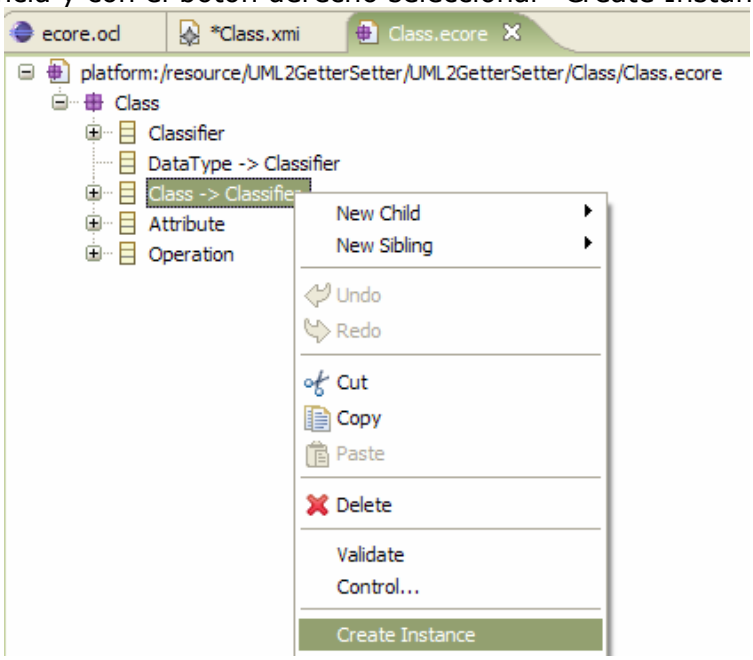
Crear un archivo nombrado "PublicToPrivate" y escribir el siguiente texto:

```
module PublicToPrivate; -- Module Template
create OUT : MetamodelUML from IN : MetamodelUML;

helper context String def: firstToUpper() : String =
      self.substring(1, 1).toUpper() + self.substring(2, self.size());

rule PublicToPrivateAttribute {
      from
            s : MetamodeloUML!Attribute (
                  (s.visibility = 'public')
            )
      to
            t : MetamodeloUML!Attribute (
                  name <- s.name,
                  type <- s.type,
                  visibility <- 'private',
                  owner <- s.owner
            ),

            -- getter
            get : MetamodeloUML!Operation (
                  name <- 'get' + s.name.firstToUpper(),
                  owner <- s.owner,
                  parameter <- parameterOUT
            ),
            parameterOUT : MetamodeloUML!Parameter (
                  name <- 'return',
                  type <- s.type

            ),

            -- setter
            set : MetamodeloUML!Operation (
                  name <- 'set' + s.name.firstToUpper(),
                  owner <- s.owner,
                  parameter <- parameterIN
            ),
            parameterIN : MetamodeloUML!Parameter (
```

```
                    name <- 'un' + s.name.firstToUpper(),
                    type <- s.type
                    )

}
rule Datatype {
      from
            d : MetamodeloUML!DataType
      to
            t : MetamodeloUML!DataType (
                  name <- d.name
            )
}
```

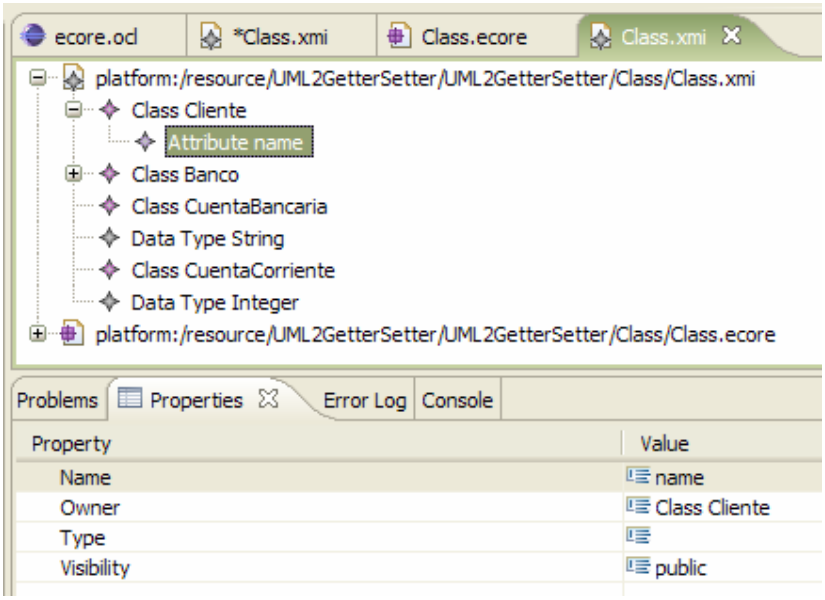
5- Instanciar el modelo de entrada

Primero hay que abrir el metamodelo del cual se quiere crear su instancia. Para esto, hay que hacer click sobre el archivo UML.ecore. Desde el editor EMF, clickear sobre la metaclase de la cual quiero crear una instancia y con el botón derecho seleccionar "Create Instance"

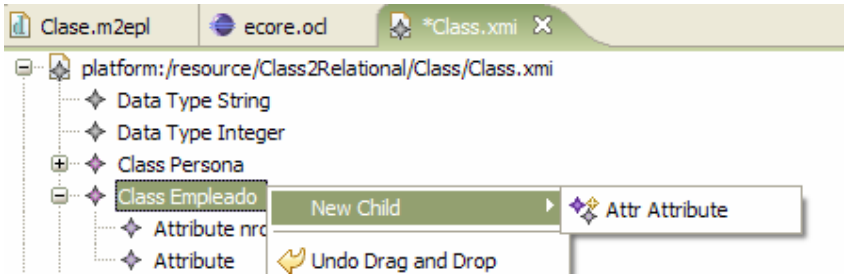


Esto creará un archivo .xmi donde se muestra en la parte superior las instancias que están siendo creadas, y en la parte inferior el metamodelo al que pertenecen.

Seleccionando cualquier instancia, se permite cambiar sus propiedades (los atributos que fueron definidos en la metaclase a la cual instancia)

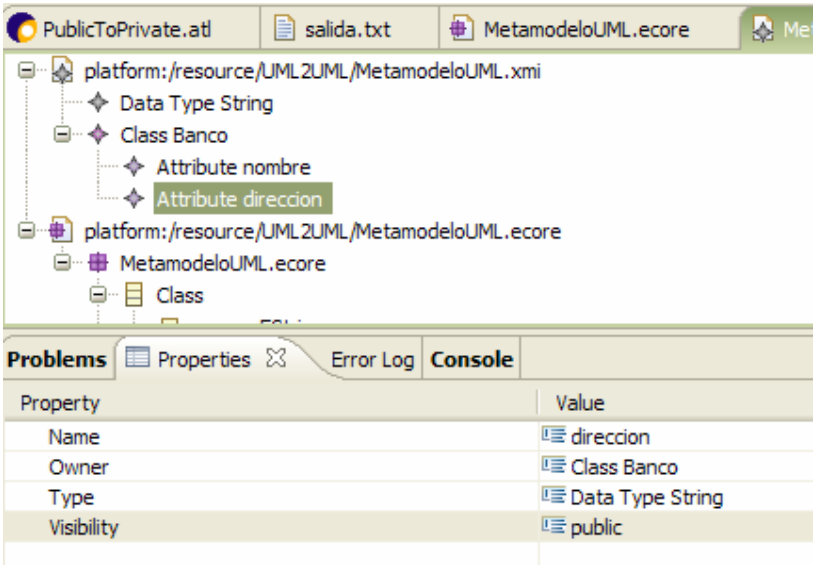


Para crear elementos anidados, como por ejemplo, para crear atributos dentro de una clase, en el árbol del modelo se clickea sobre el elemento padre (en este caso la clase) y con el botón derecho aparecen los elementos que se permiten crear. La figura siguiente muestra la creación de un elemento anidado dentro de otro.



Para este ejemplo, se instanció una sola clase, llamada "Banco" con dos atributos públicos, nombre y dirección.

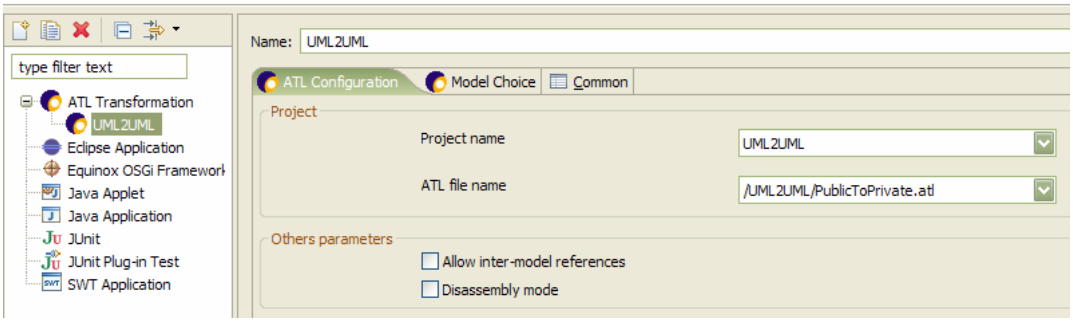
En la siguiente figura puede verse la instancia, junto con la vista Property que permite la edición de sus propiedades.



6- Ejecutar la transformación escrita

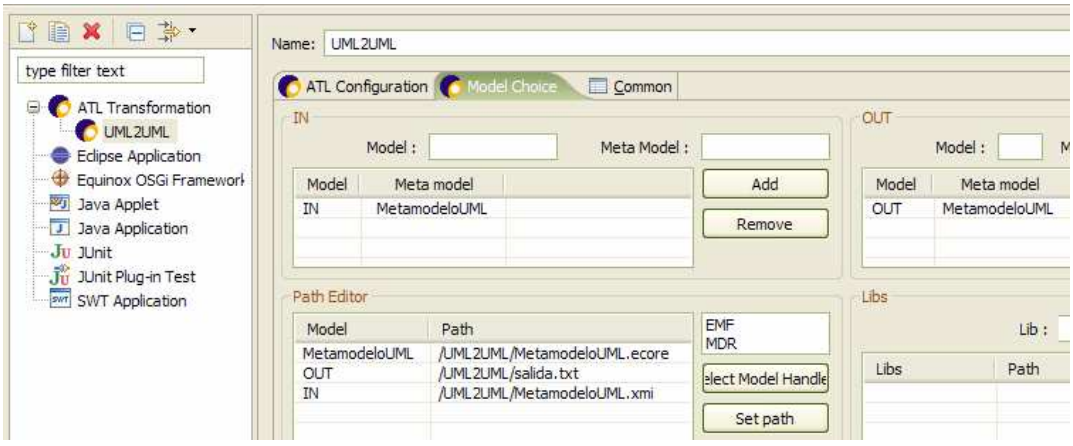
Para ejecutar una transformación en ATL primero hay que configurar esa ejecución. Para esto, hay que seleccionar en el menú Run>>Run e indicar los metamodelos de entrada y salida, el modelo de entrada y el archivo de salida.

Configurar la ejecución según las siguientes imágenes



En la imagen anterior, se especifica el proyecto y el archivo ATL que se quiere ejecutar.

En la siguiente imagen, se especifican los metamodelos, modelo y archivo de salida.



Para terminar, seleccionar la opción “RUN”, la que creará dentro del proyecto el archivo salida. Inspeccionarlo y comprobar que realizó la transformación querida, como puede verse a continuación.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:MetamodeloUML.ecore="http:/www.MetamodeloUML.ecore">

  <MetamodeloUML.ecore:Class name="Banco">
    <attribute name="nombre" type="/1" visibility="private"/>
    <attribute name="direccion" type="/1" visibility="private"/>

    <operation name="getNombre">
      <parameter name="return" type="/1"/>
    </operation>

    <operation name="setNombre">
      <parameter name="unNombre" type="/1"/>
    </operation>

    <operation name="getDireccion">
      <parameter name="return" type="/1"/>
    </operation>

    <operation name="setDireccion">
      <parameter name="unDireccion" type="/1"/>
    </operation>
  </MetamodeloUML.ecore:Class>

  <MetamodeloUML.ecore:DataType name="String"/>
</xmi:XMI>
```