TESINA DE LICENCIATURA

TITULO: Manejo Visual de Transformaciones entre modelos MOF

AUTORES: Rivera José Conrado, Rodríguez Nicolás

DIRECTOR: Giandini Roxana

CODIRECTOR: Pons Claudia

CARRERA: Licenciatura en Informática

Resumen

El paradigma de desarrollo de software conocido como MDE (acrónimo de "Model Driven software Engineering" hace énfasis en la separación entre la especificación de la funcionalidad esencial del sistema y la implementación de dicha funcionalidad usando plataformas tecnológicas específicas. Para ello, el MDE identifica dos tipos principales de modelos: modelos con alto nivel de abstracción e independientes de cualquier tecnología de implementación, llamados PIM (Platform Independent Model) y modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica, conocidos como PSM (Platform Specific Model). La transformación entre modelos constituye el motor del MDE.
Una transformación puede verse como un "programa" que toma un modelo como entrada y produce otro modelo como salida. Sería deseable que los modeladores no tengan que aprender un nuevo lenguaje para crear transformaciones. Lo ideal sería que pudiesen utilizar los mismos lenguajes de modelado estándar, por ejemplo MOF y/o UML.
Para solucionar este problema hemos definido una extensión de MOF que permite definir transformaciones. El objetivo de esta tesis es implementar una herramienta que permita crear transformaciones entre modelos MOF.

Líneas de Investigación

Lenguaje de modelado de software
Transformación de modelos
Implementación de plugins de Eclipse
Creación de Perfiles

Conclusiones

Sobre la investigación que se realizó sobre el trabajo efectuado por Giandini referente a la formalización de un Lenguaje de Transformación de Modelos se desarrolló un plugin para la plataforma Eclipse, en el cual se puede diseñar gráficamente una transformación con sus correspondientes metamodelos de entrada y de salida.
Creemos enfáticamente que el desarrollo de herramientas visuales, como lo es nuestra implementación, donde los conceptos están representados mediante iconos facilita ampliamente la labor de los programadores. MDA tiende a que la futura tarea del programador sea la de diseñar y el código sea derivable automáticamente. Para que esto sea posible es necesaria la existencia de herramientas como las que nosotros hemos desarrollado.

Trabajos Realizados

Se investigó la filosofía MDE (Model Driven Engeneering) y las transformaciones entre modelos. Se estudió el lenguaje propuesto por OMG (Object Managmment Group), QVT (Query/View/Transformation) y se comparó con SQVT (Simple QVT) que es otro lenguaje transformaciones. Se analizó y comprendió la plataforma de desarrollo Eclipse y sus plugins. Se implementó un plugin Eclipse que sirve herramienta para crear transformaciones visualmente, basado en SQVT, utilizando el lenguaje de programación Java.

Trabajos Futuros

El plugin solo soporta el chequeo de los dominios. Queda como trabajo futuro que se pueda generar un modelo de salida.
El plugin no hace distinción entre las categorías de las relaciones. Queda para trabajo futuro, realizar el chequeo y que la ejecución de la transformación tenga arranque en las relaciones que tengan en valor verdadero el atributo isTopLevel.
Integración con E-Plateu

Fecha de la presentación: [illegible] 2008

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Agradecimientos

*Quiero agradecer a los que me ayudaron para que esta tesis salga
a la luz. Para esto es necesario nombrar a mucha gente
que fue la que me rodeó en algún momento de este periodo de mi vida.
Creo que todos me hicieron llegar algo de aliento para poder terminar.
Mi madre Herminia, mi padre Alfredo, mi abuela Bibi,
Joaco, Anyu, tíos y primos, amigos de acá, del sur, jefes del trabajo,
compañeros, el Nico con quien espero seguir compartiendo grandes
proyectos, a mi hijo Thiago y a mi compañera Mariel.
A todos muchas gracias!
Conrado*

*Primero le quiero agradecer a mis padres por todo lo que me
han ayudado en este tiempo y por su apoyo incondicional.
Le quiero agradecer a mi hermana y mi cuñado por hacerme tío de
dos sobrinas hermosas y por la tercera que viene en camino.
A mis amigos de toda la vida que me han aguantado.
A mi amigo Conrado que tengo el honor de compartir esta tesis
y agradecerle por haberme dado un ahijado hermoso.
Por ultimo quiero agradecerle a mi novia Belén por comprenderme
y aguantarme día a día.
Nicolás*

*Le queremos agradecer a Roxana y Claudia por el apoyo
absoluto que nos han brindado para poder realizar esta tesis*

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Índice General

CAPÍTULO 1 8
Introducción 8
1.1- Motivación 8
1.2- Objetivos 8
1.3 - Organización de la tesis 9
CAPÍTULO 2 11
Conceptos Básicos sobre Metamodelos y Lenguajes de Transformación de Modelos 11
2.1 - MDE (Ingeniería dirigida por modelos) 11
2.1.1 - Modelos 11
2.1.2 - MDA (Arquitectura Dirigida por Modelos) 13
2.2 - Metamodelos 14
2.2.1 - MOF (MetaObject Facility) 15
2.2.2 - Arquitectura de metamodelo de cuatro capas 15
2.2.3 - La Infraestructura 2.0 17
2.3 - Lenguaje de transformación 18
2.3.1 - ¿Qué es una Transformación? 18
2.3.2 - ¿Qué es un Lenguaje de Transformación? 18
2.3.3 - Lenguaje de transformación de modelos 18
2.4 - Definición de Lenguajes a través de Perfiles 19
2.4.1 - Estereotipos 19
2.4.2 - Perfiles 19
CAPÍTULO 3 21
El lenguaje OCL 21
3.1 - Definición 21
3.2 - Descripción de OCL 21
3.3 - Expresiones OCL 22
3.3.1 - Invariantes 23
3.3.2- Definición 24
3.3.3 - Pre y Post Condiciones 24
3.3.4– Expresión de Valor Inicial 26
3.3.5 - Expresión de Valor derivado 26
3.3.6– Expresión de Consulta 27
3.3.7 – Guarda 28
3.4 - El paquete Expressions: 28
3.5 - ExpressionInOcl 32
CAPÍTULO 4 34
Ambiente de Desarrollo y Herramientas Utilizadas 34
4.1 - Eclipse 34
4.1.1 - La Plataforma 34
4.1.2 - La Plataforma Runtime y la Arquitectura de Plugin 35
4.1.3 - El Proyecto Eclipse 36
4.1.4 - La Arquitectura Eclipse 36
4.1.5 - RCP (Plataforma de Cliente Rico) 37
4.2 - Eclipse Modeling Framework Project (EMF) 38
4.3 - Graphical Editing Framework (GEF) 39
4.3.1 - Resolver un problema con GEF 40
4.3.2 - La arquitectura Model - View - Controller: 42
4.3.3 - El funcionamiento en conjunto del model - view - controler 44

4.4 - Graphical Modeling Framework (GMF) 44
4.5 - El Plugin UML2 45
4.5.1 - Introducción 45
4.5.2 - El UML y la industria de software 45
4.5.3 - El nuevo enfoque de UML 2.0 46
4.5.4 - Estándares que conforman UML 46
4.5.5 - Organización de la superestructura 46
4.5.6 - Diagramas de estructura y de comportamiento 47
4.5.7 - El Plugin 49
CAPÍTULO 5 50
Nuestra propuesta: un Plugin Eclipse para especificar transformaciones visualmente 50
5.1 - Lenguajes de transformaciones 50
5.1.1 – Un ejemplo de transformación usando QVT 50
5.1.2 - Una alternativa simple a QVT: SQVT 51
5.1.3 - El modelo SQVT 52
5.1.3.1 - Dominios chequeados y forzados 52
5.1.3.2 - Definición en forma completa del modelo SQVT 53
5.1.4 - Un ejemplo de transformación usando SQVT 56
5.1.5 - Las desventajas de QVT 56
5.1.6 – Ubicación del modelo SQVT dentro de la Arquitectura de 4 capas de Modelado 57
5.2 - La herramienta e-Platero 58
5.2.1 - Objetivos de e-Platero 58
5.2.2 - Arquitectura de e-Platero 58
5.3 - Nuestra propuesta: un Plugin para la creación de transformaciones visualmente 59
5.3.1 - Ejemplo de una transformación generada por el plugin visual de transformaciones 59
5.3.1.1 - La vista 60
5.3.1.2 - La lógica 61
CAPÍTULO 6 63
Implementación del Editor 63
6.1 - Creación del Ecore 63
6.2 - Creación del genmodel 64
6.2.1 - Model Code 65
6.2.2 - Edit Code 65
6.2.3 - Editor Code 66
6.3 - Creación del gmftool 67
6.4 – Creación del gmfgraph 69
6.5 – Creación del gmfmap 70
6.6 - Creación del gmfgen 70
6.7 – Utilización del Plugin GMF 72
6.8 - Implementación del cargador de metamodelos 73
CAPÍTULO 7 74
Implementación del Evaluador de transformaciones 74
7.1 - Diseño del Evaluador 74
7.2 - Construcción de una transformación 74
7.3 - Construcción del contexto de evaluación 76
7.4 - Creación de un ambiente de evaluación de OCL 77
7.5 - El proceso de evaluación 78
7.6 - Interpretación de los resultados 80

7.7 - Referencias a otras relaciones. Las relaciones como funciones 80
7.8 – Asunciones 81
CAPÍTULO 8 82
Conclusiones 82
8.1 – Conclusiones 82
8.2 – Trabajos Futuros 83
Referencias Bibliográficas 84
10 - ANEXO I 87
Utilización del Editor Gráfico 87
10.1 - Creación de los archivos transformation y transformation_diagram 87
10.2 - Edición de la transformación 88
10.2.1 – Creación de una figura Relation 88
10.2.2 – Creación de una figura Domain 89
10.2.3 – Creación de una figura VariableDeclaration 89
10.2.4 – Creación de una relación de Dominio 90
10.2.5 – Creación de una relación de Codominio 91
10.2.6 – Creación de una relación de Declaración de Variable 92
10.2.7 – Creación de una figura Invariant 93
10.2.8 – Creación de una relación de When 94
10.2.9 – Creación de una relación de Where 94
10.2.10 – Creación de una figura Query 95
10.2.11 – Finalizando la edición de la Transformación 96
10.3 - Carga de los metamodelos 96
10.4 - Validación de la transformación 97
GLOSARIO 99

Índice de figuras

Figura 2.1 - Capas de los tipos de modelos 12
Figura 2.2 - Arquitectura de metamodelo de cuatro capas 17
Figura 2.3 - Relaciones entre metamodelos QVT 19
Figura 2.4 - Ejemplo de un Perfil 20
Figura 3.1 - Expresión OCL utilizada como Invariante o Definición 23
Figura 3.2 - Expresión OCL utilizada como precondición o postcondición 25
Figura 3.3 - Expresión OCL utilizada como Valor Inicial de una Propiedad 26
Figura 3.4 - Expresión OCL utilizada en una Operación de Consulta 27
Figura 3.5 - Expresión OCL utilizada como una Guarda 28
Figura 3.6 - El Paquete Expressions 29
Figura 3.7 - ExpressionInOCL 33
Figura 4.1 - Plataforma Eclipse 35
Figura 4.2 - Conexión entre el modelo y la vista 40
Figura 4.3 - Modificaciones del modelo y la vista 40
Figura 4.4 - Asociación entre el modelo y la vista 41
Figura 4.5 - Actualización de la vista al modificarse el modelo 41
Figura 4.6 - Actualización de los listener al producirse un evento 42
Figura 4.7 - Seteo de propiedades a la vista 43
Figura 4.8 - Funcionamiento del model – view - controler 44
Figura 4.9 - Vista del plugin GMF 44
Figura 4.10 - Organización de la superestructura 47
Figura 5.1 -Una Instanciación gráfica de la Transformación 51
Figura 5.2 - Modelo SQVT 52
Figura 5.3 - La Transformación de Modelos en la Arquitectura de 4 capas 58
Figura 5.4 - Arquitectura de e-Platero 59
Figura 5.5 - Vista de una transformación visual 60
Figura 5.6 - Secuencia de la visualización 61
Figura 5.7 - Nuestro modelo de transformación 61
Figura 6.1 - Vista del archivo ecore 64
Figura 6.2 - Vista del archivo genmodel 66
Figura 6.3 - Vista del archivo gmftool 68
Figura 6.4 - Vista de la galería de figuras del archivo gmfgraph 69
Figura 6.5 - Vista de los nodos del archivo gmfgraph 69
Figura 6.6 - Vista del archivo gmfmap 70
Figura 6.7 - Vista del archivo gmfgen con sus propiedades 71
Figura 6.8 - Vista del GMF Dashboard, del plugin GMF 72
Figura 7.1 - Flujo en la construcción de una transformación 74
Figura 7.2 - TransformationExecutionContext como contenedor de objetos Ecore 76
Figura 7.3 - Diagrama de Secuencia de la evaluación de OCL 77
Figura 7.4 - Diagrama de Secuencia de la evaluación del predicado "when" de una relación 78
Figura 7.5 - Diagrama de Secuencia de la evaluación del predicado "where" de una relación 79
Figura 7.6 - Resultados modelados: la jerarquía EvaluationResult 80

Figura 10.1 - Creación de la transformación 87
Figura 10.2 - Creación de una figura Relation 88
Figura 10.3 - Creación de una figura Domain 89
Figura 10.4 - Creación de una figura VariableDeclaration 90
Figura 10.5 - Creación de una relación de dominio 91
Figura 10.6 - Creación de una relación de codominio 92
Figura 10.7 - Creación de una relación de Declaración de Variable 92
Figura 10.8 - Creación de una figura Invariant 93
Figura 10.9 - Creación de una relación when 94
Figura 10.10 - Creación de una relación where 95
Figura 10.11 - Creación de una figura Query 95
Figura 10.12 - Vista de la transformación finalizada 96
Figura 10.13 - Vista de las ventanas para cargar los metamodelos 97
Figura 10.14 - Vista del evaluador de transformación 98

CAPÍTULO 1
Introducción

En este capítulo se presenta una descripción general del trabajo, junto con la motivación, los objetivos planteados y una explicación general de la distribución de los temas estudiados en cada capítulo.

1.1- Motivación

El paradigma de desarrollo de software conocido como MDE (acrónimo de "Model Driven software Engineering" [4], es decir "Ingeniería de Software dirigida por Modelos") hace énfasis en la separación entre la especificación de la funcionalidad esencial del sistema y la implementación de dicha funcionalidad usando plataformas tecnológicas específicas. Para ello, el MDE identifica dos tipos principales de modelos: modelos con alto nivel de abstracción e independientes de cualquier tecnología de implementación, llamados PIM (Platform Independent Model) [2] y modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica, conocidos como PSM (Platform Specific Model) [2]. La transformación entre modelos constituye el motor del MDE; dichos modelos son sucesivamente transformados, comenzando con modelos independientes de la plataforma, con el objetivo de obtener a cada paso modelos más específicos, hasta llegar al código ejecutable.
Una transformación puede verse como un "programa" que toma un modelo como entrada y produce otro modelo como salida. Y por lo tanto dicha transformación debe tener su especificación (escrita en algún lenguaje de especificación) y su implementación (escrita por ejemplo en Java).
En la actualidad existen diversos lenguajes para definir dichas transformaciones, algunos de ellos son declarativos y otros ejecutables.
Sería deseable que los modeladores no tengan que aprender un nuevo lenguaje para crear transformaciones. Lo ideal sería que pudiesen utilizar los mismos lenguajes de modelado estándar, por ejemplo MOF [5] (Meta-Object Facility) y/o UML [29] (Unified Modeling Language).
Para solucionar este problema se ha definido una extensión de MOF que permite definir transformaciones. En este contexto es deseable contar con una herramienta amigable, basada en código abierto que automatice dicha extensión, por ejemplo un plugin para la plataforma Eclipse [13].

1.2- Objetivos

El objetivo de esta tesis es implementar una herramienta que permita crear transformaciones entre modelos MOF.
En particular la herramienta permitirá la edición de los "íconos" de este lenguaje gráfico y su "implementación" mediante el correspondiente metamodelo.
Se utilizará un lenguaje visual y declarativo que es una extensión de UML (definida mediante el mecanismo de estereotipos).

Finalmente la herramienta permitirá validar automáticamente la consistencia de los modelos de entrada y salida de la transformación. Dos modelos son consistentes cuando el resultado de aplicar la transformación al modelo de entrada resultará en el modelo de salida.
La herramienta a construir es un plugin para el IDE Eclipse, y el lenguaje a utilizar es Java.

1.3 - Organización de la tesis

Este trabajo se propone dar solución al problema de la creación de transformaciones. El plugin visual para la creación de transformaciones está basado en la definición del lenguaje de transformaciones, inspirado en QVT (Query/View/Transformation), llamado SQVT (Simple QVT) propuesto por Giandini [6], y en la herramienta e-Platero [32].
Fue elegido el lenguaje SQVT [28] porque mejora la expresividad y minimiza al lenguaje QVT [31], que es la propuesta estándar de la OMG [4] (Object Management Group) para especificar transformaciones. En particular, SQVT aumenta la utilización de expresiones OCL (Object Constraint Language) [30] para definir los predicados de una transformación. Además está creado utilizando un perfil UML en vez de extender MOF como propone la OMG.
MDA [1] (Arquitectura Dirigida por Modelos) propone separar la especificación de un sistema de los detalles en que ese sistema utiliza su plataforma. Un lenguaje de transformación de modelos permite convertir un modelo en otro. Estos conceptos se desarrollarán con más detalle en el capítulo dos de este trabajo.
El capítulo tres analizará las expresiones OCL que son utilizadas en SQVT para mejorar la expresividad en la definición de transformaciones.
Dado que el plugin propuesto está pensado para ser integrado a la herramienta e-Platero y ésta está implementada sobre la plataforma Eclipse, se estudiará, en el capítulo cuatro, la plataforma Eclipse Rich Client Platform [15] (RCP) y cómo se conectan los plugins a esta arquitectura. Algunos de ellos, ya existentes, facilitarán el desarrollo de nuestra herramienta. Para la creación del editor gráfico se utilizará EMF [20] (Eclipse Modeling Framework) que sirve para trabajar con modelos; GEF [21] (Graphical Editing Framework) que da soporte para crear editores gráficos ricos a partir de un modelo dado y GMF [19] (Graphical Modeling Framework) que es un constructor de editores que utiliza los modelos creados en EMF y GEF. Con este plugin generador de editores gráficos es posible hacer herramientas prácticas que aprovechan ventajas como drag&drop y el uso de íconos que representan los objetos a modelar. El plugin OCL servirá para poder ejecutar las expresiones OCL en el momento de evaluar un predicado perteneciente a una transformación.
Nuestra propuesta se explayará a partir del capítulo cinco, donde explicaremos cómo el editor visual de transformaciones se diseñará para brindar al usuario las herramientas que permiten agregar a una transformación sus relaciones, con sus respectivos predicados *when* y *where*. También, el plugin, proveerá facilidades para poder especificar las expresiones OCL incluidas en cada predicado. Las expresiones podrán hacer referencia a variables declaradas que tendrán un nombre y un tipo. A la transformación se le deberá asociar un metamodelo de entrada y un metamodelo de salida. Las transformaciones

creadas, luego, podrán ser evaluadas por el evaluador de transformaciones. Las expresiones OCL, pertenecientes a los predicados de las relaciones definidas, se aplicarán sobre los modelos asociados a una evaluación de transformación, instancias de los metamodelos ingresados en la creación. Esto significa que para poder evaluar hay que asociar dos modelos: uno de entrada y otro de salida. El plugin evaluará la transformación creada y devolverá los resultados en tres niveles: de cada expresión evaluada, de cada relación y el de la transformación completa.

Los capítulos siguientes al de nuestra propuesta explican como se implementó el editor grafico como así también el evaluador de transformaciones y como se realizó la conexión entre ellos, ya que fueron trabajados como dos plugins independientes.

Finalizando la tesis se encuentra un anexo, en el cual aparece la manera en que se utiliza el plugin, desde la creación del archivo de transformación, pasando por la edición de la misma hasta cómo se debe hacer para validarla.

CAPÍTULO 2
Conceptos Básicos sobre Metamodelos y Lenguajes de Transformación de Modelos

Este capítulo describe los conceptos básicos sobre metamodelos, como así también la arquitectura de metamodelos y los diferentes modelos que contempla MDA (Arquitectura Dirigida por modelos). Por ultimo se explicará los lenguajes de transformación entre modelos.

2.1 - MDE (Ingeniería dirigida por modelos)

A lo largo de esta década la Ingeniería de Software Conducida por Modelos (MDE, acrónimo de *Model Driven Engeenering*) se ha convertido en un nuevo paradigma de software que propone mejorar la construcción de software a través de un proceso guiado por modelos y soportado por potentes herramientas que generan código a partir de modelos. MDE promete una mejora de la productividad y de 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 cuyo éxito se ha comprobado.

2.1.1 - Modelos

MDE contempla cuatro tipos de modelos, como muestra la figura 2.1:

Modelo independiente de computación (CIM)

Un modelo independiente de computación es una vista de un sistema desde un punto de vista independiente de computación. Un CIM no muestra los detalles de la estructura de los sistemas. Un CIM es a veces llamado un modelo de dominio y un vocabulario que es familiar a los profesionales de este dominio en cuestión es usado en esta especificación.
Se supone que el primer usuario del CIM, el profesional del dominio, no conoce sobre los modelos o artefactos usados para realizar la funcionalidad por la cual los requerimientos son articulados en el CIM. El CIM juega un importante papel en ocupar el espacio entre los que son expertos sobre el dominio y sus requerimientos, y los que son expertos del diseño y la construcción de los componentes que juntos satisfacen los requerimientos de dominio, en el otro.

Modelo independiente de la plataforma (PIM)

Un modelo independiente de la plataforma es una vista de un sistema desde el punto de vista independiente de la plataforma. Un PIM muestra un grado específico de independencia de la plataforma para ser apropiado para el uso con un número de diversas plataformas de un tipo similar.
Una técnica muy común para lograr independencia de la plataforma es apuntar a un modelo de sistema para una máquina virtual neutral de tecnología.

Una máquina virtual es definida como un conjunto de partes y servicios (comunicaciones, scheduling, naming, etc.), los cuales son definidos independientemente de cualquier plataforma específica. Una máquina virtual es una plataforma, y tal modelo es específico de esa plataforma. Pero ese modelo es independiente de la plataforma con respecto a la clase de plataformas diferentes en donde esa máquina virtual ha sido implementada. Esto es porque tales modelos no son afectados por la plataforma subyacente.

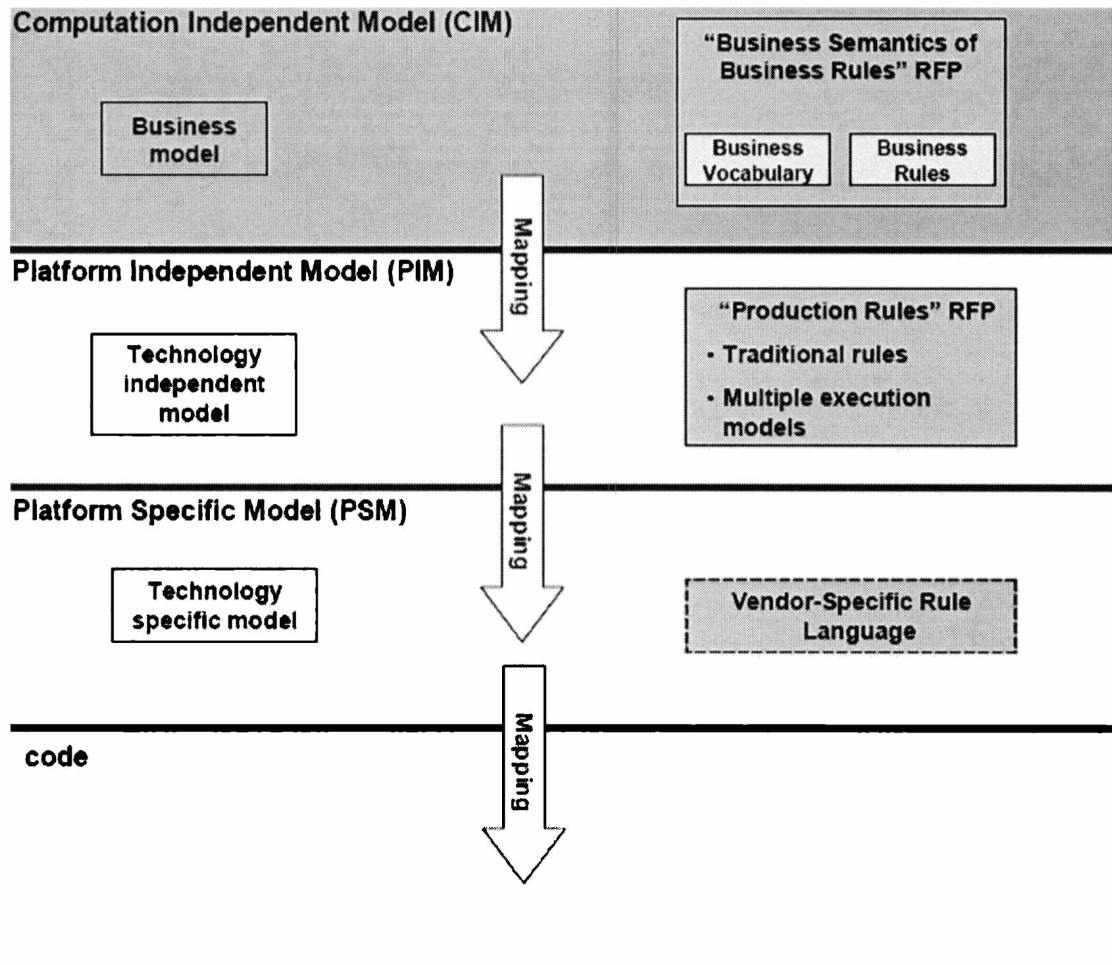


Figura 2.1 – Capas de los tipos de modelos

Modelo específico de la plataforma (PSM)

Un modelo específico de la plataforma es una vista de un sistema desde un punto de vista específico de la plataforma. Un PSM combina las especificaciones en el PIM con los detalles que especifican como ese sistema usa un tipo particular de plataforma.
Un PSM puede proveer más o menos detalles, dependiendo del propósito. Un PSM será una implementación, si provee toda la información necesaria para construir un sistema y ponerlo en operación, o puede actuar como un PIM que es usado para otro refinamiento a un PSM que puede ser directamente implementado.
Un PSM que es una implementación proveerá una variedad de información diferente, que puede incluir código de programa, descriptores de deploy, etc.

Código

Se necesita un código final que trabaje en una plataforma específica. Usando un proceso de traducción en MDA, podemos hacer PSMs para plataformas diferentes. Entonces el PSM puede ser traducido a código para ejecución.

2.1.2 - MDA (Arquitectura Dirigida 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 en noviembre de 2000 con el objetivo de abordar los desafíos de integración de aplicaciones y los continuos cambios tecnológicos.
La Arquitectura dirigida por Modelos propone definir un conjunto de normas que se especifiquen con tecnologías ínter operable, con las cuales se realizara desarrollo a través de modelos.
MDA no depende necesariamente de UML, pero, como es un tipo especializado de MDD [4] (Desarrollo a través de modelo), MDA necesariamente involucra el uso de modelo(s) en el desarrollo, lo cual conlleva a que al menos un lenguaje de modelado deba ser usado. Algún lenguaje de modelado usado en MDA debe ser descrito en términos del lenguaje MOF, para permitir a la meta data ser entendida de manera estándar, lo cual es una precondición para ejecutar transformaciones automatizadas.
MDA comenzó con la conocida idea, y establecida desde hace mucho tiempo, de separar la especificación de un sistema de los detalles en que ese sistema utiliza su plataforma.
MDA provee un acercamiento, y que permite a las herramientas ser proporcionado para:

- Especificar un sistema independiente de la plataforma que lo soporta.
- Especificar plataformas.
- Cambiar una plataforma en particular por el sistema.
- Transformar la especificación del sistema en una para una plataforma particular.

Los problemas más significativos que aparecen en el desarrollo tradicional de software son productividad, interoperabilidad y productividad. MDA trata de resolverlos de la siguiente forma:

- Productividad: El proceso de desarrollo de software tradicional, es conducido a menudo por el diseño de bajo nivel y el código. Aún si el proceso es iterativo e incremental, los documentos y los diagramas se producen sólo en las fases tempranas (requerimientos, análisis). Los documentos y los diagramas correspondientes creados en las primeras fases pierden rápidamente su valor tan pronto como la codificación

comienza. La conexión entre los diagramas y el código se pierde mientras se progresa la fase de codificación. En vez de ser una especificación exacta del código, los diagramas se convierten frecuentemente en dibujos con poca relación. Cuando un sistema cambia, la distancia entre el código, el texto y los diagramas producidos en las primeras fases, crece. Los cambios se hacen a menudo solo en el código, porque el tiempo para actualizar los diagramas y otros documentos de alto nivel no está disponible. En MDE 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 sólo una vez, y puede ser aplicada en el desarrollo de muchos sistemas.

- Portabilidad: La industria del software tiene una característica especial que la diferencia de las otras industrias. Cada año, o cada menos tiempo, aparecen nuevas tecnologías y rápidamente llegan a ser populares (por ejemplo Java, Linux, XML, HTML, UML, NET, JSP, ASP, Flash1, servicios de WEB, etc.). Las nuevas tecnologías ofrecen beneficios concretos para las compañías y muchas de ellas no pueden quedarse atrás, por lo tanto, tienen que utilizarlas muy rápidamente. El software ya existente puede seguir sin cambios, pero necesitará comunicarse con los nuevos sistemas que serán construidos usando una nueva tecnología. Dentro de MDE la portabilidad se lleva a cabo enfocando 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.

- Interoperabilidad: Los sistemas de software raramente funcionan en forma aislada. La mayoría de los sistemas necesitan comunicarse con otros que generalmente ya existen. Incluso cuando los sistemas se construyen completamente, usan a menudo múltiples tecnologías, a veces de versionados o épocas diferentes. Por ejemplo, un sistema WEB necesita usar una base de datos para almacenar información. En MDE se pueden generar muchos PSMs a partir del mismo PIM, y estos frecuentemente pueden estar relacionados. Estas relaciones se llaman *bridges* o puentes.

2.2 - Metamodelos

Un **meta-metamodelo** (OMG, 2003) es un modelo que define el lenguaje formal para representar un metamodelo. La relación entre un meta-metamodelo y un metamodelo es análoga a la relación entre un metamodelo y modelo.

Un **metamodelo** (OMG, 2003) es un modelo que define el lenguaje formal para representar un modelo.

2.2.1 - *MOF (MetaObject Facility)*

La especificación MOF es el fundamento de los ambientes estándares de la industria OMG donde los modelos puede ser exportados de una aplicación, importados desde otra, transportados a través de la red, almacenados en un repositorio y luego recuperado, renderizado en diferentes formatos, transformados y usados para generar código de aplicación. Estas funciones no son restringidas a modelos estructurales, o incluso a modelos definidos en UML (modelos de comportamiento y datos también participan en este ambiente), los lenguajes de modelado no UML también participan, mientras que sean basados en MOF.
La especificación define un núcleo del modelo MOF que incluye un conjunto relativamente pequeño de construcciones para el modelado de información orientado a objetos. El modelo MOF puede ser extendido por herencia y composición para definir un modelo de información más rica que soporte construcciones adicionales. Alternativamente, el modelo MOF puede ser usado como un modelo para definir modelos de información. Esta característica permite al diseñador definir modelos de información que difieran de la filosofía o detalles del modelo MOF. En este contexto, el modelo MOF se refiere como un meta-metamodelo porque es usado para definir metamodelos como por ejemplo el UML.

2.2.2 - *Arquitectura de metamodelo de cuatro capas*

El metamodelo UML esta definido en una de las capas de la arquitectura de meta-modelado de cuatro capas [28]. Esta arquitectura es una arquitectura comprobada para definir las semánticas precisas requeridas por los modelos complejos. Hay otras ventajas asociadas con esta propuesta:

- Valida las construcciones de núcleo recursivamente aplicándoselas a los sucesivas meta-capas.
- Provee bases arquitecturales para definir futuras extensiones de metamodelo UML.
- Suministra bases arquitecturales para alinear el metamodelo UML con otros estándares basados en la arquitectura de modelado de cuatro capas.

El framework conceptualmente aceptado para meta modelado esta basado en una arquitectura de 4 capas (figura 2.2):

1. *Meta-metamodelo*: La capa de meta-meta modelado es el fundamento de la arquitectura de meta modelado. La responsabilidad primaria de esta capa es definir el lenguaje para especificar un metamodelo. Un meta-metamodelo define un modelo en un nivel más alto de abstracción que un metamodelo, y es mas compacto que el metamodelo que describe. Un meta-metamodelo puede definir múltiples metamodelos, y

puede haber múltiples meta-metamodelos asociados con cada metamodelo. Mientras es deseable que los meta-metamodelos y metamodelos relacionados compartan filosofías de diseño y construcciones comunes, esto no es una regla estricta. Cada capa necesita mantener su propia integridad del diseño. Ejemplos de meta-meta objetos en la capa de meta-meta modelado son: Meta clase, Meta atributo y Meta Operación. Si no hay un meta-metamodelo explícito, entonces hay un meta-metamodelo implícito asociado a cada metamodelo.

2. *Metamodelo*: Un metamodelo es una instancia de un meta-metamodelo. La responsabilidad primaria de la capa de metamodelo es de definir un lenguaje para especificar modelos. Los Metamodelos son típicamente mas elaborados que los meta-metamodelos que describen estos, especialmente cuando ellos describen semánticas dinámicas. Ejemplos de meta objetos en la capa de meta modelado son: Clase, atributo, operación y componente.

3. *Modelo*: Un modelo es una instancia de un metamodelo. La primera responsabilidad de la capa de modelo es de definir un lenguaje que describa el dominio de la información. Ejemplos de objetos en la capa de modelado son: Persona (Clase), edad (atributo), getEdad (operación).

4. Objetos de usuario: Los objetos de usuarios son instancia de un modelo. La principal responsabilidad de la capa de objetos de usuario es de describir el dominio de información específico. Ejemplos de objetos en la capa de objetos de usuario son: Juan Pérez (Persona), 38 (Edad)

M3 (MOF)
Class
«instanceOf» «instanceOf» «instanceOf»
M2 (UML)
Attribute
Class
classifier
Instance
«instanceOf» «instanceOf» «instanceOf» «instanceOf»
M1 (User model)
Video
+title: String
«snapshot»
: Video
title = "2001: A Space Odyssey"
«instanceOf»
M0 (Run-time instances)
aVideo

Figura 2.2 - Arquitectura de metamodelo de cuatro capas

2.2.3 - La Infraestructura 2.0

En el nivel más abstracto de la Arquitectura 4 capas de Modelado se ubica MOF que es un Meta-metamodelo cerrado. En consecuencia, para definir lenguajes que incorporen la definición de nuevos conceptos, por ejemplo para transformaciones, a través de extensiones de especificaciones OMG ya existentes, no se debería extender a MOF, por ser una especificación cerrada. La Infraestructura [4] es la especificación más simplificada que define los constructores básicos y conceptos comunes para lenguajes de modelado. Es independiente al lenguaje UML en sí. El metamodelo de UML se complementa con la especificación de la Superestructura, que define los constructores a nivel usuario de UML 2.0.
El caso de la Infraestructura es interesante por su definición recursiva respecto a MOF: por un lado, puede verse definida como instancia de MOF; por otro lado el MOF mismo usa elementos del paquete principal, llamado Core, de la Infraestructura para su definición, situación que permite identificar a la Infraestructura como un meta-metamodelo. El paquete Core puede ser considerado el núcleo arquitectural de MDA. La intención es reusar todo o parte de este paquete al definir otros metamodelos, brindando el beneficio de contar con sintaxis abstracta y semántica que ya han sido definidas.

2.3 - Lenguaje de transformación

2.3.1 - ¿Qué es una Transformación?

Según Kleppe [25], 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 cómo un modelo en el lenguaje fuente puede ser transformado en un modelo en el lenguaje destino.

Una *Regla de Transformación* es una descripción de cómo una o más construcciones en el lenguaje fuente pueden ser transformadas en una o más construcciones en el lenguaje destino.

2.3.2 - ¿Qué es un Lenguaje de Transformación?

Un Lenguaje de Transformación es un lenguaje de computadora diseñado para transformar un texto de entrada escrito en lenguaje formal en un texto de salida modificado que satisface algún objetivo específico.

2.3.3 - Lenguaje de transformación de modelos

La transformación de modelo es el proceso de convertir un modelo Ma conforme al metamodelo MMa en un modelo Mb conforme al metamodelo MMb. Si MMa= MMb, entonces la transformación es *endógena* sino es una transformación *exógena.*
Un lenguaje de transformación de modelos existente es el QVT.
El QVT (Querys, Views, Transformations) es un estándar para transformación de modelo definido por el OMG.
La transformación de modelo es un componente crítico del MDA.
Reconociendo esto, una RFP (Solicitud para propuesta) se emitió por el OMG en MOF QVT para buscar un estándar compatible con el conjunto de componentes de MDA (UML, MOF, OCL, etc.)
Varias respuestas fueron de compañías y por un conjunto de instituciones de investigación que se desarrollaron durante 3 años para producir una propuesta común que fue presentada y aprobada.
Actualmente hay varios productos (comerciales o código abierto) que demandan conformidad al estándar QVT. QVT define una manera estándar para transformar modelos de entrada en modelos de salida. Hay varias ideas en esta propuesta. Una es que los modelos de entrada y salida puedan conformarse a metamodelos MOF arbitrarios. Otra es que el programa de transformación sea considerado como un modelo, y como consecuencia también sea conforme a un metamodelo MOF. Esto significa mas precisamente que la sintaxis abstracta de QVT sea conforme al metamodelo MOF 2.0. De hecho, esta es la parte más compleja. Primero el lenguaje QVT se integra con el estándar OCL 2.0 y también extiende al OCL imperativo. Segundo que

QVT define no uno sino tres lenguajes específicos de dominio llamados Relaciones, Núcleo y Mapeo Operacional (como muestra la figura 2.3) y esos lenguajes son organizados en una arquitectura laminada. Las relaciones y el Núcleo son lenguajes declarativos en dos niveles diferentes de abstracción, con un mapeo normativo entre ellos. El lenguaje de relaciones tiene una sintaxis concreta gráfica. El lenguaje de mapeo operacional es un lenguaje imperativo que extiende el Núcleo y Relaciones. La sintaxis del lenguaje de Mapeo Operacional provee construcciones comúnmente encontradas en los lenguajes imperativos (iteraciones, condiciones, etc.).

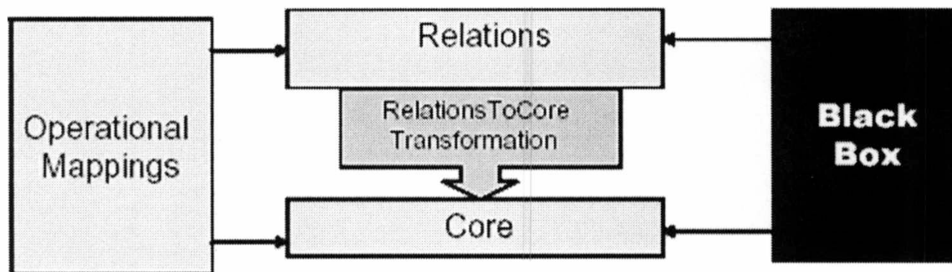


Figura 2.3: Relaciones entre metamodelos QVT

2.4 - Definición de Lenguajes a través de Perfiles

2.4.1 - Estereotipos

Los estereotipos [24] son uno de los tres mecanismos de extensión en UML. Los estereotipos permiten extender el vocabulario de UML para crear nuevos elementos del modelo, derivados de unos existentes, pero que tengan propiedades específicas del dominio. Son usados para la clasificación o marcado de bloques de construcción de UML con el fin de introducir nuevos bloques de construcción que hablen el lenguaje del dominio y que puedan mirar elementos primitivos o básicos del modelo.
Por ejemplo, cuando se necesita modelar una red se necesitan tener símbolos para representar los routers y hubs. Pero usando nodos estereotipados se puede utilizar que esas cosas aparezcan como bloques de construcción primitivos.

2.4.2 - Perfiles

Un perfil [24] en UML proporciona un mecanismo de extensión genérica para la creación de modelos UML en particular dominios. Los perfiles están basados en estereotipos adicionales y valores etiquetados que son aplicados a los elementos, atributos, métodos, etc.
Un perfil es un conjunto de extensiones y restricciones que juntos describen algún problema de modelado en particular y facilitan el modelado de los constructores en el dominio.
La figura 2.4 muestra un ejemplo de perfil UML.

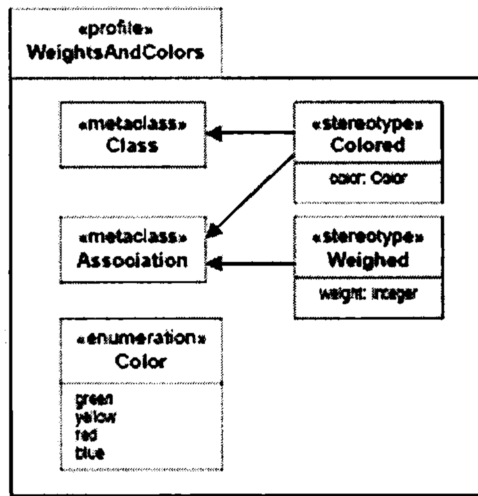


Figura 2.4 – Ejemplo de un Perfil

CAPÍTULO 3
El lenguaje OCL

El lenguaje OCL (Object Constraint Language) es un estándar conocido por la gran mayoría de los diseñadores que utilizan el lenguaje UML para modelar, ya que está integrado a éste. Es la herramienta que le permite al diseñador especificar restricciones. En nuestro trabajo la utilización de las expresiones OCL resulta de mucha utilidad para poder describir transformaciones. Evaluar una transformación consiste en evaluar las expresiones OCL que la transformación contiene.

3.1 - Definición

Es un lenguaje formal que se utiliza para describir expresiones en modelos UML. Estas expresiones, generalmente, describen condiciones invariantes que tienen los sistemas modelados y/o consultas sobre los objetos allí especificados. No generan efectos colaterales al ser evaluadas, es decir, una evaluación no afecta el estado de un objeto.
Los diseñadores pueden realizar consultas escritas en un metalenguaje independizándose así de la plataforma de implementación del sistema.

3.2 - Descripción de OCL

Generalmente los modelos UML no tienen la riqueza suficiente para poder expresar ciertas lógicas del problema, entonces las restricciones se escriben en lenguaje natural para dejarlas aclaradas. El problema es que estas restricciones resultan ambiguas muchas veces, dando lugar a la utilización de lenguajes formales para su reemplazo. Dichos lenguajes son comprendidos por personas con un alto conocimiento matemático lo que dificulta su entendimiento por los diseñadores y programadores.
OCL fue creado para solucionar este problema, ya que es un lenguaje formal, fácil de escribir y de leer. Fue creado como un lenguaje de modelado de negocios.
Es un lenguaje de especificación, lo que implica que una expresión OCL garantiza la ausencia de efectos laterales. Las expresiones al ser evaluadas devuelven un valor y no modifican nada en el modelo, es decir que el estado del sistema no cambia cuando una expresión es evaluada; sin embargo una expresión OCL puede ser utilizada para representar un cambio de estado (por ejemplo una post-condición).
No es un lenguaje de programación, es un lenguaje de modelado. Por este motivo no se pueden realizar flujos de control y llamados a funciones. Todas las expresiones tienen un tipo, ya que es un lenguaje tipado.
OCL puede ser utilizado para varios propósitos:

- Como un lenguaje de consulta
- Para especificar invariantes en clases y tipos de un modelo de clases.

- Para describir pre y post condiciones en Operaciones y Métodos.

- Para definir operaciones y variables adicionales para los tipos de un modelo de clases.

- Para describir guardas (Guards)

- Para especificar reglas de derivación para una propiedad.

- Para especificar los valores iniciales de las propiedades.

- Para especificar cualquier expresión de un modelo.

- Para especificar restricciones en operaciones.

Ejemplo de una expresión OCL

La palabra context denota el contexto donde la expresión va a ser evaluada. La palabra inv, pre y post denotan los estereotipos <<invariant>>, <<precondition>> y <<postcondition>> respectivamente.

context TypeName **inv**:

3.3 - Expresiones OCL

Cualquier cosa en la especificación UML donde se utilice el término expresión hace referencia a una expresión OCL. Pueden ser utilizadas como invariantes, precondiciones y postcondiciones pero también podrían tener otro uso. Para cada ocurrencia de una expresión OCL hay tres aspectos que deben ser separados: el lugar, el clasificador contextual y la instancia de *self* en una expresión OCL.

- El lugar es la posición donde es usada la expresión OCL dentro del modelo UML. Como por ejemplo una invariante conectada a la clase Persona.

- El clasificador contextual define el nombre en el cual la expresión es evaluada. Por ejemplo, el clasificador contextual de una precondición es el clasificador que es dueño de la operación para la cual fue definida la precondición.

- La instancia de *Self* es la referencia al objeto que evalúa la expresión. Siempre es una instancia del clasificador contextual.

3.3.1 - Invariantes

Una expresión OCL puede ser parte de una invariante que no es más que una Restricción estereotipada como <<invariant>>. Cuando una invariante está asociada a un Clasificador, éste es llamado "tipo". Una expresión OCL es una invariante de tipo y debe ser verdadera para todas las instancias de ese tipo en todo momento. (Notar que todas las expresiones OCL que expresan invariantes son de tipo Boolean). En la figura 3.1 se representa la expresión OCL utilizada como invariante.
Por ejemplo la siguiente expresión puede ser una invariante que determina que el número de empleados en una empresa (representada con el tipo *Company*) siempre debe ser mayor a 50:

```
self.numberOfEmployees > 50
```

Donde *self* es una instancia del tipo *Company*. Esta invariante vale para todas las instancias de Company. El tipo de una instancia contextual de una expresión OCL es escrito con la palabra clave *context*, seguido del nombre del tipo. La etiqueta *inv:* declara que la restricción es una restricción <<invariant>>.

```
context Company inv:
self.numberOfEmployess > 50
```

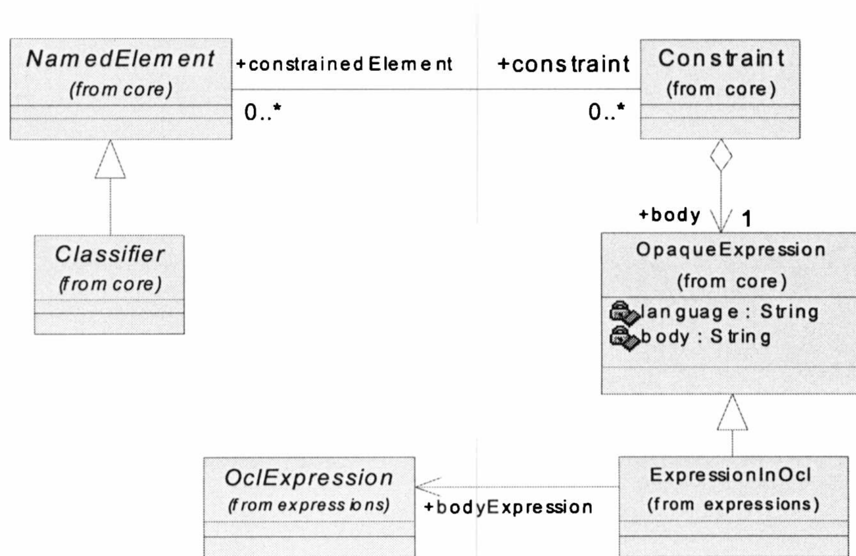


Figura 3.1 - Expresión OCL utilizada como Invariante o Definición

En la mayoría de los casos la palabra clave *self* puede ser quitada ya que el contexto es claro. Para reemplazar a *self* puede definirse un nombre diferente.

```
context c : Company inv:
c.numberOfEmployees > 50
```

Esta invariante es equivalente a la anterior.

Opcionalmente, el nombre de la restricción puede escribirse luego de la palabra clave *inv*, permitiendo que pueda ser referenciada por nombre. En el ejemplo el nombre es enoughEmployees

```
context c : Company inv enoughEmployees:
c.numberOfEmployees > 50
```

3.3.2 - Definición

Una definición es un Constraint que se liga a un Classifier. La variable o función definida puede utilizarse como una propiedad o una operación del correspondiente Classifier. El propósito de esta restricción es definir expresiones OCL reusables.
La restricción tiene el estereotipo <<definition >>. Las definiciones se ligan a un solo Classifier, y este es el tipo de la variable contextual.
A continuación se define su sintaxis:

```
context [VariableName:] TypeName
def : [VariableName] | [OperationName(ParameterName1: Type, ...)] :
ReturnType = < OclExpression >
```

En la siguiente expresión OCL se define una variable llamada capacidad, que retorna la capacidad del avión asignado al correspondiente vuelo.

```
context Vuelo
def : capacidad : Integer = self.miAvion.miTipo.capacidad
```

La variable capacidad es conocida en el contexto de Vuelo.

```
context Vuelo inv:
self.misReservas -> size() <= capacidad
```

3.3.3 - Pre y Post Condiciones

Las expresiones OCL pueden ser parte de una Precondición o de una Postcondición, correspondientes a los estereotipos, de restricción (Constraint) asociados con una operación (Operation), <<precondition>> y

<<postcondition>>.
Para señalar que una expresión OCL es una Pre o Post condición se utilizan las palabras claves 'pre:' y 'post:' de la siguiente manera:

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

El nombre *self* puede ser utilizado en la expresión refiriéndose al objeto sobre el cuál la operación fue llamada. La palabra reservada *result* indica el resultado de la operación, si es que hay uno. Los nombres de los parámetros (param1) pueden ser utilizados en la expresión OCL. Un ejemplo:

```
context Person::income(d : Date) : Integer
post: result = 5000
```

Opcionalmente se puede especificar el nombre de la precondición o postcondición después de la palabra clave pre o post, permitiendo a la restricción ser referenciada por nombre. En la figura 3.2 se presenta la expresión OCL utilizada como postcondición.
En el ejemplo la precondición se llama parameterOk y el de la postcondición resultOk. En el metamodelo de UML estos nombres son valores del atributo name de la metclass Constraint.

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

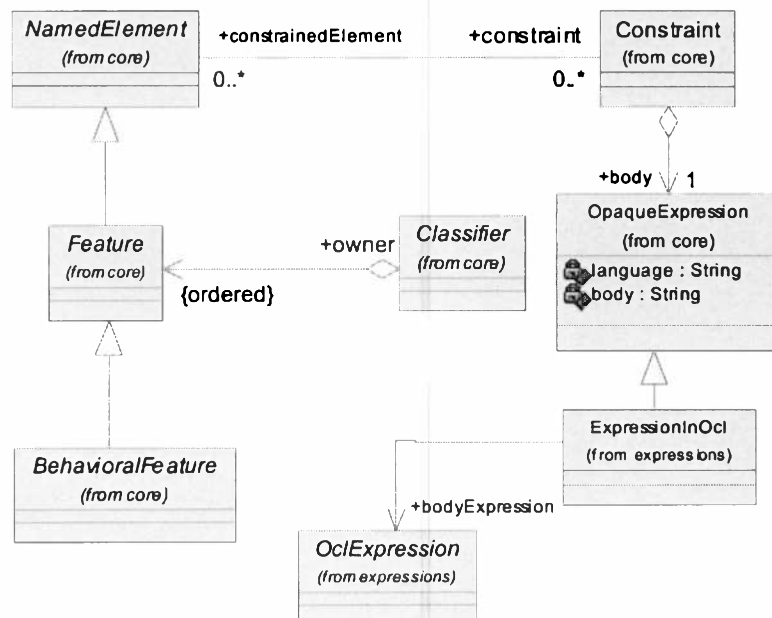


Figura 3.2 - Expresión OCL utilizada como precondición o postcondición

3.3.4 – Expresión de Valor Inicial

Una expresión de valor inicial es una expresión que se liga a un Property. Una expresión OCL que actúa como el valor inicial debe conformar al tipo definido por la propiedad. Además hay que tener en cuenta su multiplicidad, es decir si la multiplicidad es mayor que uno el tipo es un Set u OrderedSet del tipo de la propiedad. En la figura 3.3 se representa la expresión utilizada para definir el valor inicial de una propiedad.
Su sintaxis es:

```
context Typename:: propertyName: Type
init: -- alguna expresión representando el valor inicial de la propiedad
```

La propiedad llamada cancelado de la clase Vuelo tiene asignado un valor inicial de falso.

```
context Vuelo :: cancelado : Boolean
init: false
```

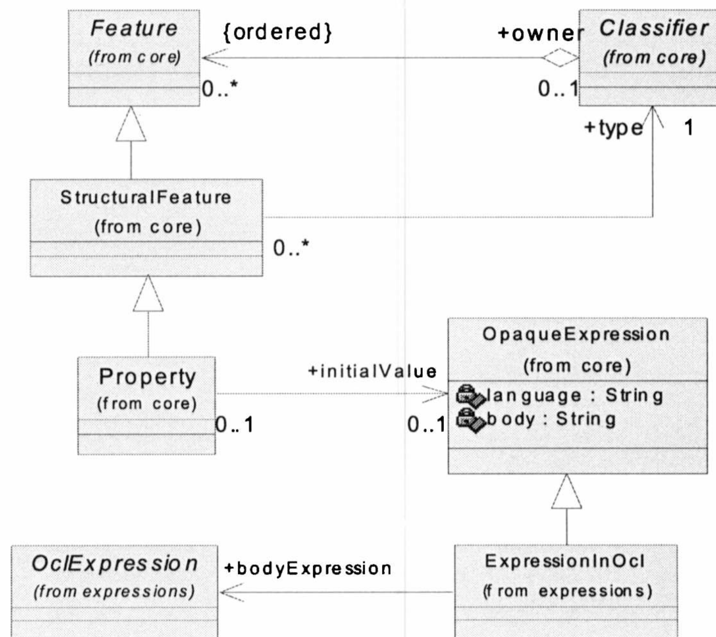


Figura 3.3 - Expresión OCL utilizada como Valor Inicial de una Propiedad

3.3.5 - Expresión de Valor derivado

Una expresión de valor derivado es una expresión que se liga a una propiedad. La expresión OCL que actúa como el valor derivado de una propiedad debe conformar al tipo de esta. De igual modo que cuando definimos el valor inicial tenemos que tener en cuenta la multiplicidad de la propiedad. Si esta es mayor que uno el tipo es un Set u OrderedSet del tipo de la propiedad actual.

Una expresión de valor derivado se metamodela como un invariante, el cual especifica el valor de la propiedad.
Su sintaxis es:

context Typename:: propertyName: Type
derive: -- alguna expresión representando la regla de derivación

Ejemplo: Se define una propiedad derivada llamada tipoAvion, en la clase Vuelo.

context Vuelo :: tipoAvion
derive: self.miAvion.miTipo

3.3.6– Expresión de Consulta

Una expresión de consulta es una expresión que se liga a una operación de consulta definida en un Classifier. Para indicar que es una operación de consulta se utiliza el atributo isQuery. En la figura 3.4 se representa el metamodelo dicha expresión.

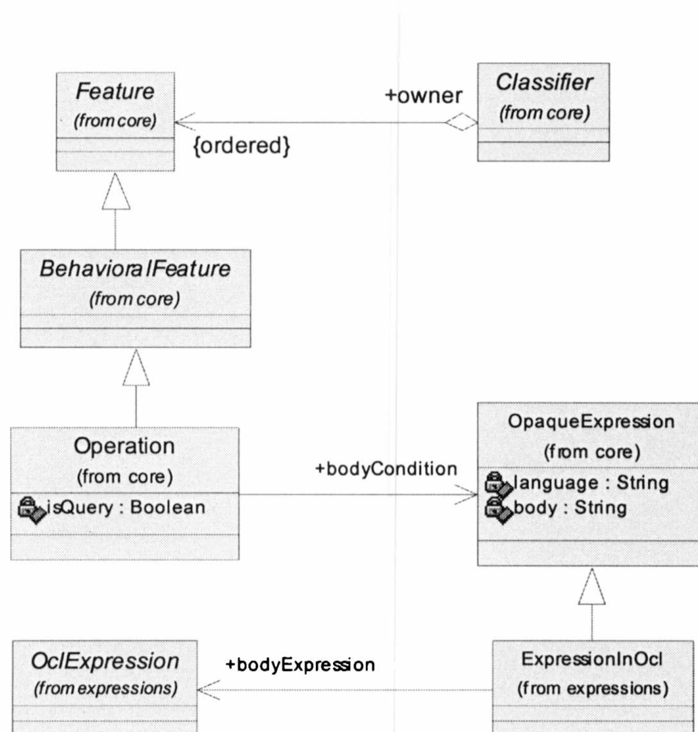


Figura 3.4 - Expresión OCL utilizada en una Operación de Consulta

Su sintaxis es:

context Typename:: operationName(parameter1: Type1, . . .) : ReturnType
body: -- alguna expresión

La expresión debe ser conforme con el tipo de la operación. Al igual que en las precondiciones y postcondiciones los parámetros pueden ser utilizados en la expresión. Las precondiciones, postcondiciones y expresiones de consulta pueden ser combinadas luego de especificar el contexto de una operación.

Se define la operación de consulta getCapacidad, definida en la clase avión

context Avion: getCapacidad() : Integer
body: self.miTipo.capacidad

3.3.7 – Guarda

Una guarda es una expresión del tipo Boolean que se liga a una transición de una maquina de estados. Una expresión OCL que actúa como guarda se utiliza para restringir la transición. Es decir, la condición debe ser satisfecha para facilitar la activación de la transición asociada. El metamodelo para la guarda se ilustra en la figura 3.5.

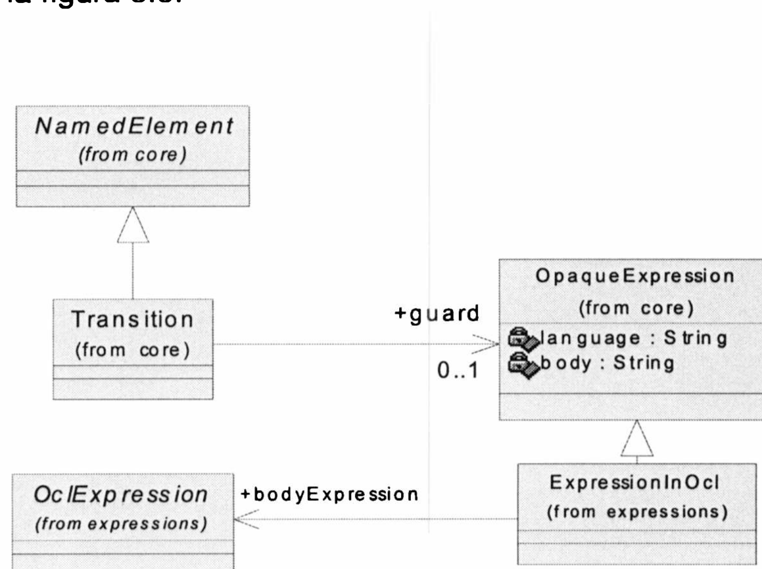


Figura 3.5 - Expresión OCL utilizada como una Guarda

3.4 - El paquete Expressions:

En *Expressions* se describe la estructura que pueden tener las expresiones OCL. En la figura 3.6 se ilustra el conjunto de componentes que integran el paquete Expressions.

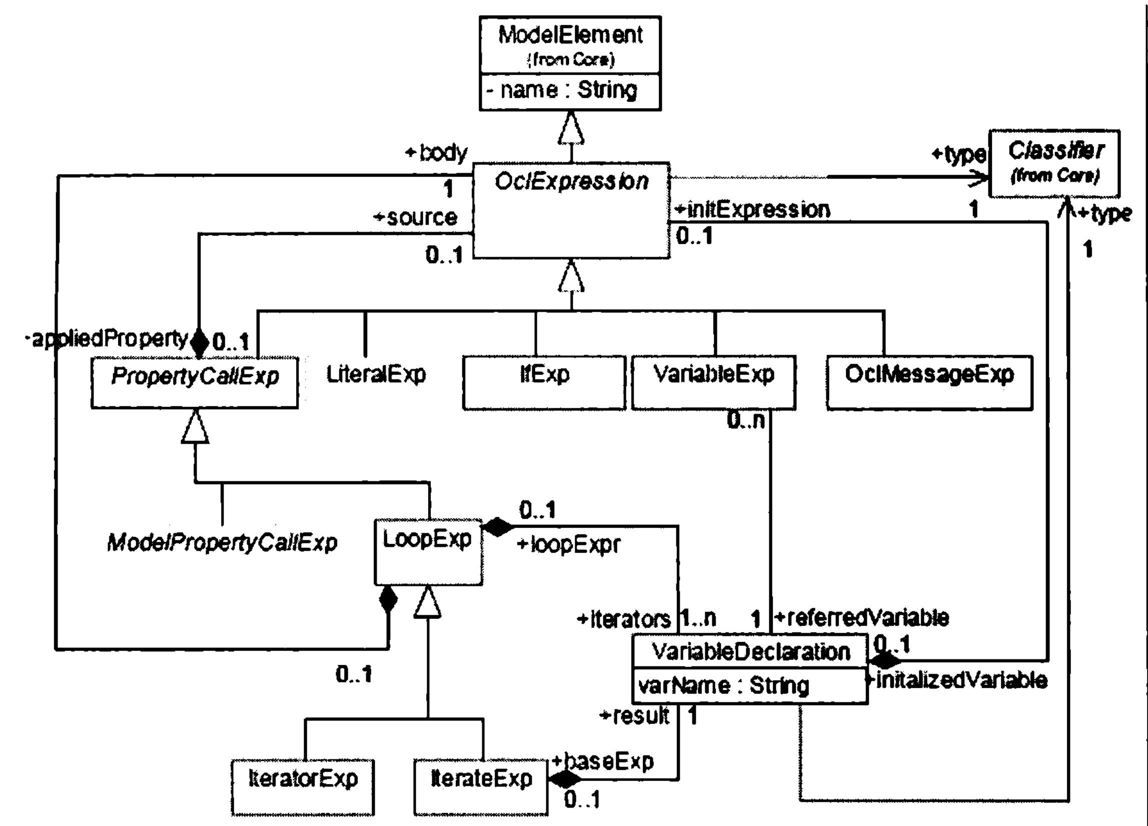


Figura 3.6 -: El Paquete Expressions

<u>Expressions Core</u>

Una OclExpression siempre tiene asociado un tipo que no necesariamente está modelado, pero si derivado.

<u>PropertyCallExp</u>

Una PropertyCallExp es una expresión que se refiere a un Feature (operación, propiedad) o a un iterator predefinido para las colecciones. El resultado se obtiene a partir de la evaluación del feature correspondiente. Esta es un metaclase abstracta.

Asociaciones

- Source: el receptor de la invocación de la propiedad.

<u>ModelPropertyCallExp</u>

Un FeatureCallExp es una expresión que se refiere a un Feature definida en un Classifier del modelo. El resultado es la evaluación de la propiedad correspondiente.

LoopExp

Un LoopExp es una expresión que representa un bucle sobre una colección. Tiene una variable que se utiliza para iterar sobre los elementos de dicha colección. La expresión body es evaluada para cada elemento contenido en la colección. El tipo del resultado de la expresión loop se indica en sus subclases.

Asociaciones

- iterator: Representa la variable que se utiliza para iterar sobre los elementos de la colección en el momento de la evaluación.

- body: La expresión OCL que es evaluada para cada elemento de la colección receptora.

IfExp

La expresión "if" ofrece dos alternativas a seguir, en base a la comprobación de la condición. Tanto el thenExpression y elseExpression son obligatorios ya que una expresión siempre debe resultar en un valor. El tipo de la expresión es el supertipo común a las dos expresiones alternativas.

Asociaciones

- *condition:* El OclExpression que representa la condición. Si esta condición evalúa a true, el resultado del la expresión "if" es idéntica al resultado del thenExpression. En caso contrario el resultado del la expresión "if" es idéntica al resultado del elseExpression.

- *thenExpression:* La OclExpression que representa la parte del then de la expresión "if".

- *ElseExpression:* La OclExpression que representa la parte del else de la expresión "if".

IterateExp

Una IterateExp es una expresión que evalúa la expresión indicada por la asociación body para cada elemento de la colección receptora. Cada uno de los valores resultantes de la evaluación forma parte de un nuevo valor de la variable result. El resultado puede ser de cualquier tipo y es definido por la asociación result.

Asociaciones

- *result:* representa la variable resultante.

IteratorExp

Una IteratorExp es una expresión que evalúa la expresión indicada por la asociación body para cada elemento de la colección receptora. El resultado de la evaluación es un valor cuyo tipo depende del nombre de la expresión. En algunos casos puede ser del mismo tipo que el receptor. La metaclase IteratorExp representa todas las operaciones predefinidas de las colecciones que se definen a través de la expresión iterator, por ejemplo select, exists, collect, forAll, etc.

LiteralExp

Un LiteralExp es una expresión sin argumentos que representa un valor. Algunas de sus subclases son: IntegerLiteralExp, BooleanLiteralExp, etc.

OclExpression

Una OclExpression es una expresión que puede ser evaluada en un ambiente dado. Esta metaclase es la superclase de todas las expresiones en el metamodelo. Toda expresión OCL tiene un tipo que se puede determinar estáticamente analizando la expresión y su contexto. La evaluación de la expresión retorna un valor. Las expresiones cuyo tipo es un boolean se puede utilizar en las restricciones. En caso contrario para operaciones query, valores iniciales de atributo, etc.
El ambiente (Environment) de una OclExpression define que elementos del modelo son visibles y pueden ser referenciados en una expresión. El ambiente puede ser definido por el elemento del modelo que está ligado a la expresión OCL, por ejemplo un Classifier si la restricción es un invariante. Los iteradores de la expresión también pueden ser introducidos en el ambiente.

Asociaciones

- appliedProperty: propiedad que es aplicada a la instancia que resulta de la evaluación de esta OCLExpression.

- type: el tipo del valor que es el resultado de evaluar esta OCLExpression.

- parentOperation: La OperationCallExp donde esta OCLExpression es un argumento.

- initializedVariable: la variable que tiene como valor inicial el resultado de esta expresión.

VariableDeclaration

Una VariableDeclaration declara un nombre de una variable y lo liga a un tipo. Esta variable puede ser utilizada en expresiones donde tenga alcance. Esta metaclase representa entre otras las variables self y result.

Asociaciones

- *initExpression:* expresión OCL que representa el valor inicial de la variable.
- *type:* El Classifier que representa el tipo de la variable.

Atributos

- *varName:* Una cadena que representa el nombre de la variable.

VariableExp

La VariableExp es una expresión que consiste en una referencia a una variable.

Asociaciones

- *referredVariable:* La VariableDeclaration a la que esta expresión se refiere.

3.5 - ExpressionInOcl

Es subclase de la metaclase Expression del núcleo de UML y su asociación *bodyExpression* es una OCLExpression. La asociación *contextualClassifier* es el clasificador que corresponde al contexto de la expresión OCL (Self siempre es una instancia de este clasificador). El atributo *body* (heredado de Expression) se utiliza para almacenar el texto de la expresión OCL y el atributo *language* (también heredado de Expression) tiene el valor 'OCL'. La figura 3.7 ilustra una ExpresiónInOcl con sus composiciones.

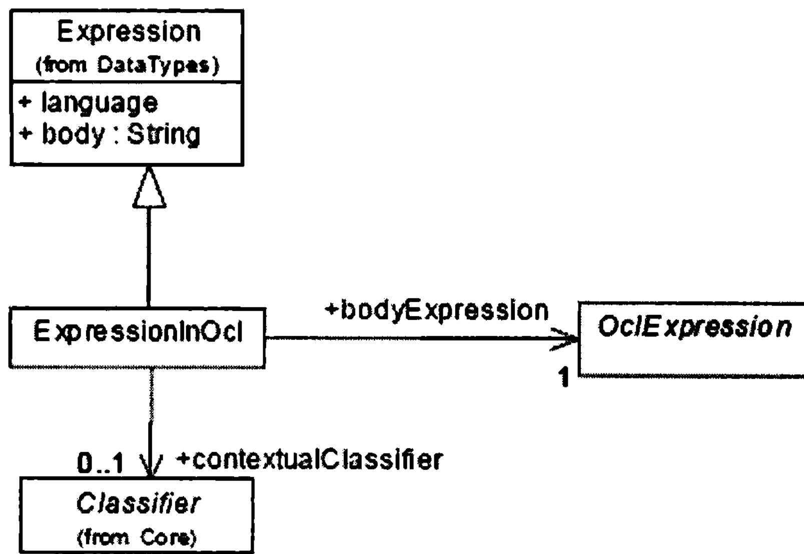


Figura 3.7 - ExpressionInOCL

CAPÍTULO 4

Ambiente de Desarrollo y Herramientas Utilizadas

Este capítulo describirá el ambiente de desarrollo Eclipse (plataforma, arquitectura, proyecto, etc.). También se explicará como es el manejo de plugins en dicho ambiente. Por ultimo se expondrán los plugins utilizados para la creación del editor.

4.1 - Eclipse

4.1.1 - La Plataforma

Eclipse es una plataforma de software de código abierto independiente de una plataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Esta plataforma, típicamente ha sido usada para desarrollar entornos integrados de desarrollo (del inglés IDE).

La plataforma eclipse, figura 4.1, está construida sobre un mecanismo para descubrir, integrar y correr módulos llamados plugins. Un proveedor de herramientas escribe una herramienta como un plugin separado, que opera sobre archivos en el workspace y su interfaz de usuario específica trabaja sobre la superficie del workbench. Cuando la plataforma es cargada, el usuario es presentado con un ambiente de desarrollo integrado (IDE) compuesto de un conjunto de plugins habilitados.

La Plataforma Eclipse esta diseñada y construida para cumplir con los siguientes requerimientos:

- Soportar la construcción de una variedad de herramientas para el desarrollo de aplicaciones.
- Soportar un irrestricto conjunto de proveedores de herramientas, incluyendo vendedores de software independiente (ISVs).
- Soportar herramientas para manipular tipos de contenido arbitrario (por ej. HTML, Java, C, JSP, EJB, XML, y GIF).
- Permitir una fácil integración de las herramientas entre sí, a través de los diferentes tipos de contenidos y sus proveedores.
- Soportar el desarrollo de aplicaciones basadas y no, en interfaz gráfica de usuario (en inglés Graphical User Interface, GUI y non-GUI-based).
- Correr en un ancho rango de sistemas operativos.

- El principal objetivo es el de proveer facilidades a los proveedores de herramientas para desarrollar las mismas con mecanismos de uso y reglas para seguir. Estos mecanismos son provistos por una API (Application Programming Interface - Interfaz de Programación de Aplicaciones) de interfaces bien definida, clases y métodos.

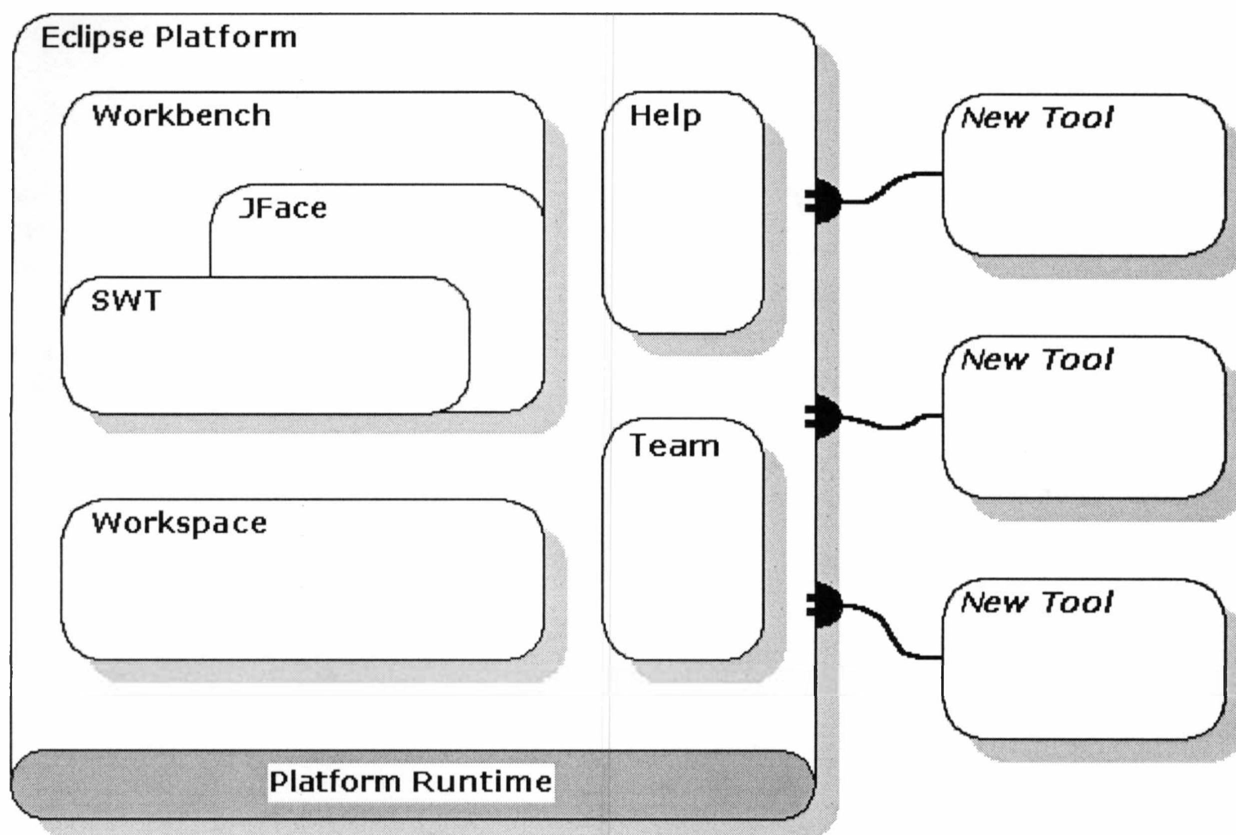


Figura 4.1 – Plataforma Eclipse

4.1.2 - La Plataforma Runtime y la Arquitectura de Plugin

Un Plugin es la mínima unidad de la Plataforma Eclipse que puede ser desarrollada y distribuida separadamente. Usualmente una pequeña utilidad se escribe como un simple plugin, mientras que una utilidad compleja tiene su funcionalidad repartida entre varios plugins. Salvo por el pequeño kernel conocido como Plataforma Runtime, toda la funcionalidad de la Plataforma Eclipse está realizada con plugins.

Estos se desarrollan en código Java que se guarda en una librería JAR, junto con algunos recursos como imágenes, páginas HTML, etc. e información de solo lectura. Estas librerías de plugins junto a la información de solo lectura son guardadas en un directorio del sistema de archivos o en una URL dentro de un servidor. Este es un mecanismo que permite que los plugins puedan ser sintetizados por muchos fragmentos separados, cada uno en su propio directorio o URL.

Cada plugin tiene un archivo llamado *manifest* que declara las interconexiones a otros plugins. El modelo de interconexiones es simple: un plugin declara un número de puntos de extensiones, y un número de extensiones a uno o más puntos de extensión en otros plugins.

Un punto de extensión puede ser extendido por otro plugin. Por ejemplo, el workbench plugin declara un punto de extensión para las preferencias del usuario. Cualquier plugin puede contribuir con sus propias preferencias para el usuario definiendo extensiones a este punto de extensión.

4.1.3 - El Proyecto Eclipse

El Proyecto Eclipse [13] es un proyecto de desarrollo de software de código abierto dedicado a proporcionar una plataforma industrial robusta, con amplias características y con calidad comercial para el desarrollo de herramientas altamente integradas.
Está compuesto de tres sub-proyectos: la Plataforma Eclipse, la Java Development Tool y el Plug-in Development Environment. El éxito de la Plataforma Eclipse depende de cómo sea capaz de admitir una amplia gama de herramientas de desarrollo para reproducir lo mejor posible las herramientas existentes en la actualidad.

4.1.4 - La Arquitectura Eclipse

Eclipse lo forman el núcleo, el entorno de trabajo (Workspace), el área de desarrollo (Workbench), la ayuda al equipo (Team support) y la ayuda o documentación (Help).

- Núcleo: su tarea es determinar cuales son los plugins disponibles en el directorio de plugins de Eclipse. Cada plugin tiene un fichero XML manifest que lista los elementos que necesita de otros plugins así como los puntos de extensión que ofrece. Como la cantidad de plugins puede ser muy grande, solo se cargan los necesarios en el momento de ser utilizados con el objeto de minimizar el tiempo de arranque de Eclipse y recursos.

- Entorno de trabajo: maneja los recursos del usuario, organizados en uno o más proyectos. Cada proyecto corresponde a un directorio en el directorio de trabajo de Eclipse, y contienen archivos y carpetas.

- Interfaz de usuario: muestra los menús y herramientas, y se organiza en perspectivas que configuran los editores de código y las vistas. A diferencia de muchas aplicaciones escritas en Java, Eclipse tiene el aspecto y se comporta como una aplicación nativa. No está programada en Swing, sino en SWT (Standard Widget Toolkit) y Jface (juego de herramientas construida sobre SWT), que emula los gráficos nativos de cada sistema operativo. Este ha sido un aspecto discutido sobre Eclipse, porque SWT debe ser portada a cada sistema operativo para interactuar con el sistema gráfico. En los proyectos de Java puede usarse AWT y Swing salvo cuando se desarrolle un plugin para Eclipse.

- Ayuda al grupo: este plugin facilita el uso de un sistema de control de versiones para manejar los recursos en un proyecto del usuario y define

el proceso necesario para guardar y recuperar de un repositorio. Eclipse incluye un cliente para CVS.

- Documentación: al igual que el propio Eclipse, el componente de ayuda es un sistema de documentación extensible. Los proveedores de herramientas pueden añadir documentación en formato HTML y, usando XML, definir una estructura de navegación.

4.1.5 - RCP (Plataforma de Cliente Rico)

El término cliente rico surge del fenómeno de la década del 90 con las aplicaciones para clientes basadas en las tecnologías Visual Basic y Delphi, el incremento y el éxito de estas aplicaciones fue debido a la "rica" experiencia del usuario.

Las aplicaciones de cliente rico [14] son productos de alta calidad y proveen interfaces de usuario ricas (que permitan drag&drop, sistema de portapapeles, navegación y personalización) y gran velocidad de procesamiento. El término cliente rico se utiliza para diferenciarlos de aquellos clientes de aplicaciones de terminal, llamados clientes simples.

Las plataformas de cliente rico sirven para unir las aplicaciones de cliente rico con los sistemas operativos, haciendo más fácil la integración con las interfaces nativas y el acceso a las bases de datos, proveyendo frameworks e infraestructura para que el programador se dedique solamente a programar la lógica de negocio.

Las aplicaciones de cliente rico entonces empiezan a ser populares pero tienen problemas con las actualizaciones y la implantación, ya que son tareas manuales, por ejemplo el usuario instalaba en lugares equivocados en el sistema de archivos o reemplazaba las librerías compartidas. Es en este momento fue cuando llegó Internet y con el surgen las aplicaciones de cliente fino, que son las aplicaciones basadas en la tecnología Web. Estas aplicaciones solucionaron el problema de la instalación y actualizaciones ya que están centralizadas en un servidor. Pero el problema es que estas aplicaciones pierden la riqueza de las interfaces de usuario y la velocidad de procesamiento, respecto a las aplicaciones de cliente rico porque tienen que adaptarse al modelo pregunta y respuesta *http* que requiere gran tecnología para la red. Esta última característica es fundamental para los dispositivos móviles que poseen escasez de recursos. La inserción en el mercado de las tecnologías móviles, sumado a la complejidad de los dominios que permiten el incremento de información para manipular y visualizar, conlleva a volver a utilizar aplicaciones de clientes ricos para las aplicaciones móviles. Para solucionar el problema de la actualización y la implantación surgidas en un principio ahora existen mecanismos de manejo de componentes que facilitan estas tareas. Como por ejemplo la forma de trabajar con plugins que provee Eclipse.

La plataforma de cliente rico Eclipse es una plataforma genérica para correr aplicaciones. El IDE Eclipse es solo una de las aplicaciones que puede correrse.

Eclipse Rich Client Platform es el framework para la creación rápida de aplicaciones de escritorio basadas en la plataforma Eclipse. Este framework

está formado por el conjunto mínimo de plugins de Eclipse necesarios para ejecutar una aplicación basada en este entorno de manera independiente.
El framework Eclipse RCP destaca por su facilidad de uso. Ofrece un sistema de gestión de ventanas basado en perspectivas, vistas y editores, con todo su ciclo de vida automatizado y con una gran cantidad de funcionalidad lista para aprovechar. Además, la mayoría de los componentes que ofrece actualmente Eclipse pueden ser aprovechados e incluidos en nuestras propias aplicaciones. Prácticamente todas las piezas que incluye Eclipse han sido separadas del núcleo e incluidas dentro de Eclipse RCP. Es el caso, por ejemplo, del sistema de ayuda, de la gestión de asistentes, del sistema de manejo de tareas o la gestión de errores.
Otra de las ventajas de Eclipse RCP, es que se puede aprovechar la enorme cantidad de plugins disponibles en el mercado, e integrar éstos dentro de las aplicaciones, de modo que se puede aprovechar gran cantidad de esfuerzo ya realizado previamente e incrementar de este modo la productividad.

4.2 - Eclipse Modeling Framework Project (EMF)

El proyecto EMF es un framework de modelado y generación de código para construir herramientas y otras aplicaciones basadas en un modelo de dato estructurado. De una especificación de modelo descripta en XMI [33], EMF provee herramientas y soporte runtime para producir un conjunto de clases Java para el modelo, junto con un conjunto de clases adaptadoras que permiten la visualización y edición vía comandos del modelo, y un editor básico.
Los modelos pueden ser especificados usando anotación Java, documentos XML, o herramientas de modelado como Rational Rose, y después ser exportados a EMF. Lo más importante de todo, EMF suministra las bases para la interoperabilidad con otras herramientas y aplicaciones basadas en EMF.
El EMF incluye XML Schema Infoset Model (XSD), ahora un componente del proyecto Model Development Tools (MDT), y una implementación basada en EMF de Service Data Objects (SDO). XSD provee un modelo y una API para manipular componentes de un esquema XML, con acceso a representación DOM del documento del esquema.

EMF consiste de tres piezas fundamentales:

- *EMF:* El core del framework EMF incluye un meta modelo (ECore) para describir modelos y soporte runtime para los modelos incluyendo notificaciones de cambio, soporte de persistencia con serialización XMI [33] (XML Metadata Interchange) por defecto, y una muy eficiente API para manipular objetos EMF genéricamente.

- *EMF.Edit*: El framework EMF.Edit incluye clases reutilizables genéricas para construir editores para modelos EMF. Esto provee:

    I. Clases proveedoras de contenido y etiquetas, y otras clases que permiten a los modelos EMF ser mostrados usando visualizadores de escritorio estándar.

II. Un framework de comandos, incluyendo un conjunto de clases de implementación de comandos genéricos para editores de construcción que soporten completamente el "rehacer" y "deshacer" automáticos.

- EMF.Codegen: La generación de código EMF es capaz de generar todo lo necesitario para construir un editor completo para un modelo EMF. Esto incluye un GUI desde el cual las opciones de generación pueden ser especificadas, y los generadores a ser invocados. La generación se basa en JDT (Java Development Tooling), componente de Eclipse. Hay tres niveles de generación de código que son soportadas:

  I. Modelo: Proporciona clases Java de implementación e interfaz para todas las clases del modelo, además una clase de implementación de factory y package (meta data).
  II. Adaptadores: Genera clases de implementación (llamadas ItemsProviders) que adaptan las clases del modelo para ser editadas y mostradas.
  III. Editor: Produce un editor estructurado adecuadamente que se ajusta al estilo recomendado por los editores de modelos EMF Eclipse y sirve como punto de partida al comienzo de la personalización.

Todos los generadores soportan regeneración de código mientras preservan las modificaciones del usuario. Los generadores pueden ser invocados a través del GUI o desde una línea de comando.

4.3 - Graphical Editing Framework (GEF)

Permite a los desarrolladores tomar un modelo de aplicación existente y crear fácilmente un editor grafico rico. GEF permite a los desarrolladores mapear rápidamente cualquier modelo existente con un ambiente de edición grafico. El ambiente grafico es el SWT basado en el plugin de dibujo Draw2D (el cual es parte del componente GEF). El desarrollador puede sacar ventaja de muchas operaciones comunes previstas en GEF y/o extendiéndolas para un dominio especifico.
GEF es apropiado para crear gran variedad de aplicaciones, incluyendo:

I. constructores GUI
II. editores de diagramas UML (como por ejemplo workflow y diagramas de modelado de clases)
III. editores de texto WYSIWYG como HTML.

GEF no asume que se tiene que construir alguna de estas aplicaciones por lo que el dominio de la aplicación es neutro.

4.3.1 - Resolver un problema con GEF

Se plantea un problema, el cual es resuelto con GEF. El problema planteado se resuelve de dos formas diferentes:

1) Se quiere construir una aplicación con los requerimientos siguientes:

- Por un lado hay un modelo compuesto de objetos de contención de datos
- Por otro lado una vista compuesta de objetos de definición de algunos cuadros en la pantalla
- El usuario debe permitir modificar la vista con el mouse y el teclado
- Algún link entre la vista y el modelo define que ocurre en el modelo cuando la vista es modificada y viceversa.

Tal aplicación le proporciona al usuario una forma de modificar un modelo gráficamente (esto es llamado editor gráfico).
La vista, el modelo y el link entre ellos son ilustrados en la figura 4.2.

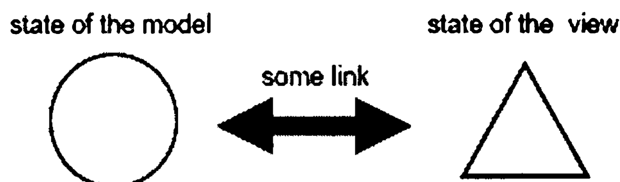


Figura 4.2 – Conexión entre el modelo y la vista

En este problema, el link entre el modelo y la vista puede ser cualquier cosa, es posible que para dos estados idénticos del modelo, la vista mostrada en el usuario sea diferente. Por ejemplo, si el usuario puede mover las diferentes figuras de la vista y que el resto se mantenga en sus posiciones, allí será necesario varias vistas para el mismo estado de modelo (uno para cada localización posible que el usuario pueda cambiar las figuras en la vista).

El editor gráfico debería trabajar como muestra la figura 4.3:

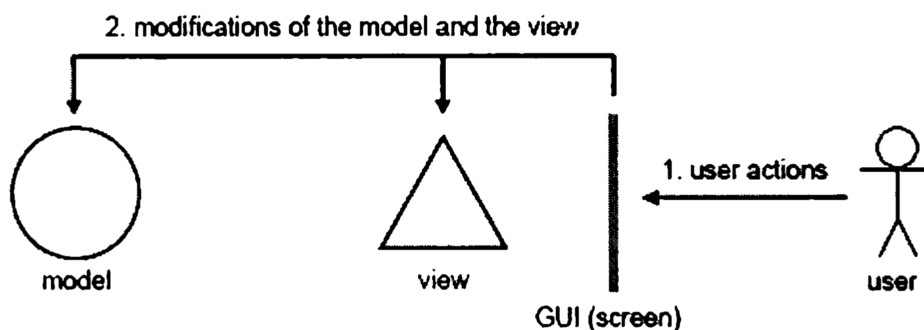


Figura 4.3 – Modificaciones del modelo y la vista

Se piensa que esta solución es confusa ya que para el mismo estado del modelo se tendrían muchas vistas asociadas, lo cual no es eficiente.

2) Otra solución para el mismo problema que el anterior es que ahora el link entre el modelo y la vista sea más específico: Para cada estado de modelo, se tiene definido una vista que puede ser mostrada por el usuario para ese particular estado. Como lo describe la figura 4.4.
De esta manera la vista se muestra al usuario dependiendo solamente del estado actual del modelo y está definido completamente por ella: la vista es una función del estado del modelo.

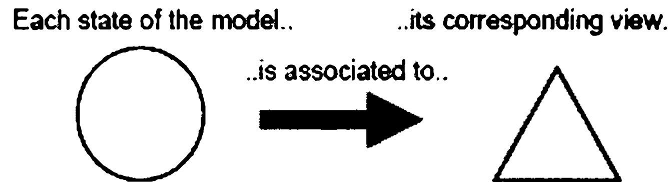


Figura 4.4 – Asociación entre el modelo y la vista

Ahora las acciones del usuario en la interfaz gráfica pueden ser interpretadas en términos de modificaciones del modelo solamente porque se sabe que la nueva vista será definida por el nuevo estado del modelo, cualquiera sean las acciones del usuario en la GUI. Ahora se puede marcar una separación clara entre:

- Las modificaciones del modelo disparadas por las acciones del usuario en la interfaz gráfica.
- Las actualizaciones de la vista disparadas por las modificaciones del modelo acorde a este nuevo estado.

La figura 4.5 ejemplifica lo anteriormente mencionado:

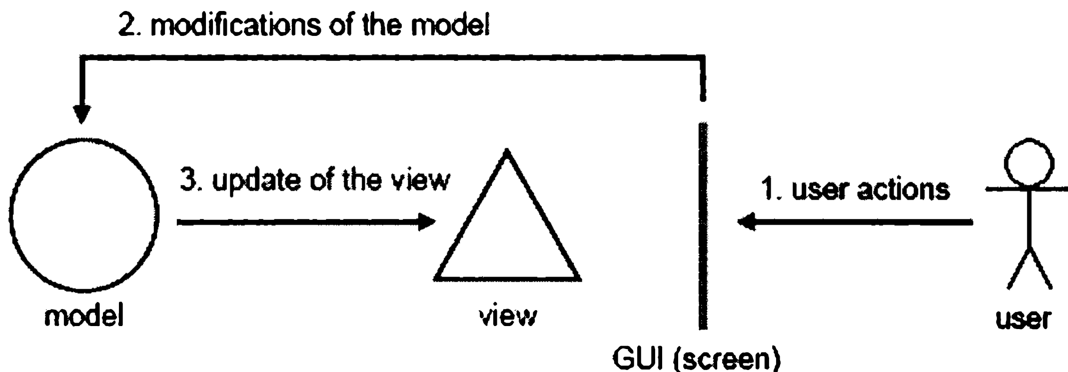


Figura 4.5 – Actualización de la vista al modificarse el modelo

4.3.2 - *La arquitectura Model - View - Controller:*

La arquitectura model - view - controller (MVC) es una arquitectura de software que aplica a los editores gráficos. Todas las partes de GEF son construidas para tomar lugar dentro de esta arquitectura.

- Modelo: hay varios requerimientos sobre el modelo que deben ser conocidos:
    I. El modelo no debe conocer nada sobre la vista o sobre cualquier otra parte del editor. El modelo no debe tener alguna referencia a su vista. Esto es muy importante. El modelo es un contenedor de datos que serán modificados durante el proceso de edición e indicarán estos cambios a través de un mecanismo de modificación.

    II. El modelo debe implementar algún tipo de mecanismo de notificación. Este debe disparar eventos cuando cambie y debe ser posible registrar los escuchadores o listeners para capturar estos eventos, como muestra la figura 4.6:

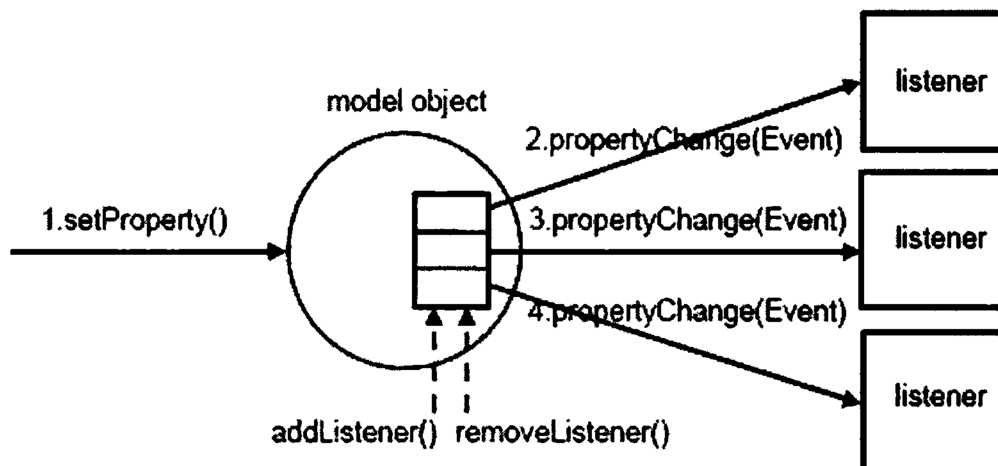


Figura 4.6 – Actualización de los listener al producirse un evento

GEF no asume algo sobre el modelo que se usa. Esto significa que se podrán utilizar casi todos los modelos con GEF. Pero, por otro lado, también significa que se tendrá que tener cuidado de los requerimientos anteriores. En particular se tendrá que implementar el mecanismo de notificación, y los listener para ésto.

- Vista: La vista es el conjunto de bloques de construcción que componen la interfaz gráfica. Como para el modelo, la arquitectura MVC especifica algunos requerimientos para la vista:

I. La vista no debe contener ningún dato importante que ya esté almacenado en el modelo. Esto es justo una consecuencia de los requerimientos del modelo.
II. La vista no debe conocer nada sobre el modelo o sobre cualquier otra parte del editor. La vista no debe contener ninguna referencia al modelo. Esto es muy importante. La vista debe ser completamente boba y no participar de ninguna manera en la lógica del editor. La vista puede ser distinguida como un map usado por el algoritmo de pintado para pintar la interfaz gráfica. Es una buena práctica acceder a las propiedades de los bloques de construcción de la vista a través de interfaces (como se ve en la figura 4.7):

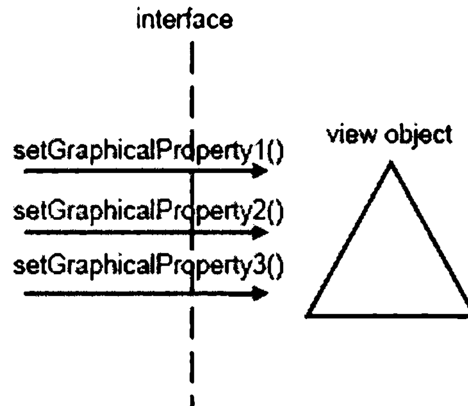


Figura 4.7 – Seteo de propiedades a la vista

- Controladores: Como el modelo no controla ninguna referencia a la vista y la vista no controla ninguna referencia al modelo, habría que saber cuál es el link entre los dos. Este link esta hecho por los controladores. En GEF los controladores son subclases de EditPart.
  Hay un EditPart entre cada objeto del modelo que ha sido representado gráficamente y su vista. El EditPart conoce sobre los dos (éste contiene una referencia al objeto del modelo y a su vista) así puede reunir información sobre el objeto del modelo y setear las propiedades gráficas a la vista. El EditPart es registrado como un listener sobre estos objetos del modelo para ser informado sobre sus cambios, y cuando este cambio ocurra, conocer como actualizar la vista acorde al nuevo estado del objeto del modelo. El controlador está también involucrado en el proceso de edición de su objeto de modelo.

4.3.3 - El funcionamiento en conjunto del model - view - controler

La figura 4.8 resume gráficamente todo el proceso anteriormente descrito.

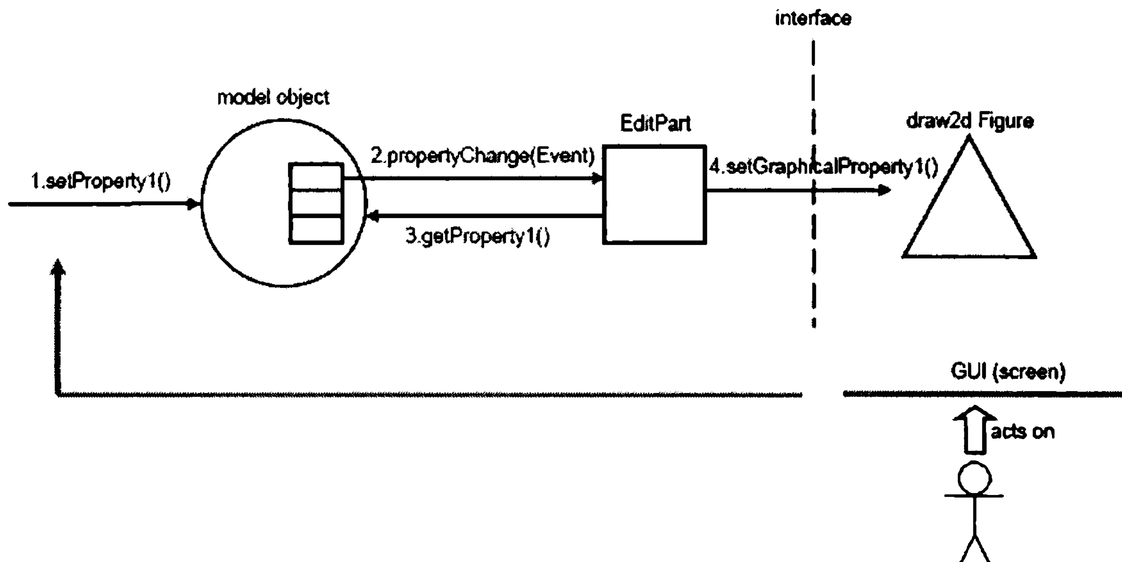


Figura 4.8 – Funcionamiento del model – view - controler

4.4 - Graphical Modeling Framework (GMF)

El Eclipse Graphical Modeling Framework (GMF) provee un componente generativo e infraestructura runtime para desarrollar editores gráficos basados en EMF y GEF. El proyecto tiene por objetivo proveer estos componentes, mas una herramienta para seleccionar modelos de dominio que ilustren estas capacidades.

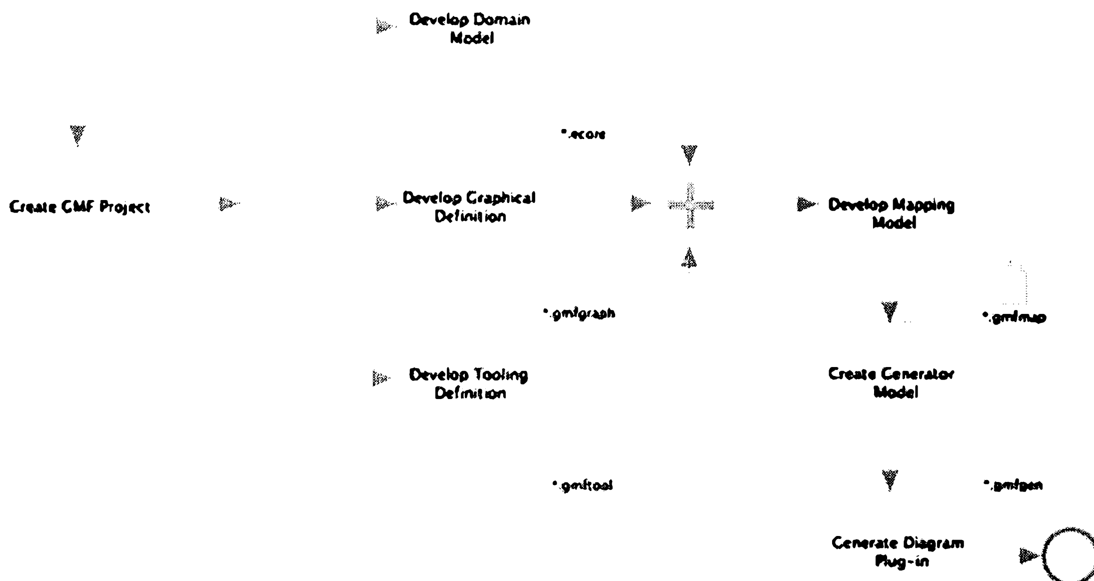


Figura 4.9 – Vista del plugin GMF

La figura 4.9 es un diagrama que ilustra los componentes principales usados durante el desarrollo basado en GMF. El núcleo de GMF es el concepto de un modelo de definición grafica. Este modelo contiene información relacionada a los elementos gráficos que aparecerán en GEF durante el tiempo de ejecución, pero tienen conexión no directa con los modelos de dominio para que ellos provean representación y edición. Una opcional herramienta de definición de modelo es usada para diseñar la paleta y otras periferias (menús, barra de herramientas, etc.).
Es de esperar que una definición gráfica o de herramienta pueda trabajar igual de bien para varios dominios. Por ejemplo, el diagrama de clases UML tiene muchas contrapartes, de las cuales todas son sorprendentemente similares en su apariencia básica y estructura. Un objetivo de GMF es permitir que la definición gráfica sea rehusada por varios dominios. Esto es logrado por utilizar un modelo de mapeo separado para conectar las definiciones gráficas y de herramientas a los modelos de dominio seleccionados.
Una vez que los mapeos apropiados están definidos, GMF provee un modelo generador que permite a los detalles de la implementación ser definidos en la fase de generación. La producción de un editor de plugin basado en el generador de modelo apuntara al modelo final; que es; el modelo de diagrama en tiempo de ejecución. El tiempo de ejecución será puente entre la notación y el modelo de dominio cuando el usuario este trabajando con un diagrama, y también proveyendo para la persistencia y sincronización de ambos. Un importante aspecto de este runtime es que provee servicios basados en funcionalidad EMF y GEF.

4.5 - El Plugin UML2

4.5.1 - Introducción

Al momento de desarrollar el nuevo estándar 2.0 de UML, la OMG se planteó, entre otros, dos objetivos que se podrían considerar principales, debido a la influencia de éstos en la nueva versión del estándar:

1. Hacer el lenguaje de modelado más extensible.
2. Permitir la validación y ejecución de modelos.

UML 2.0 se desarrolla sobre la base de estos dos objetivos, causando un quiebre respecto a versiones anteriores.

4.5.2 - El UML y la industria de software

El UML se ha vuelto el estándar de facto (impuesto por la industria y los usuarios) para el modelado de aplicaciones de software. En los últimos años, su popularidad trascendió al desarrollo de software y, en la actualidad, el UML es utilizado para modelar muchos otros dominios, como por ejemplo el modelado de procesos de negocios.

BIBLIOTECA
FAC. DE INFORMÁTICA

UML son las siglas para *Unified Modeling Language*, que en castellano quiere decir: Lenguaje de Modelado Unificado. Para comprender qué es el UML, basta con analizar cada una de las palabras que lo componen, por separado.

- *Lenguaje*: el UML es, precisamente, un lenguaje. Lo que implica que éste cuenta con una sintaxis y una semántica. Por lo tanto, al modelar un concepto en UML, existen reglas sobre cómo deben agruparse los elementos del lenguaje y el significado de esta agrupación.
- *Modelado*: el UML es visual. Mediante su sintaxis se modelan distintos aspectos del mundo real, que permiten una mejor interpretación y entendimiento de éste.
- *Unificado*: unifica varias técnicas de modelado en una única.

Ya que el UML proviene de técnicas orientadas a objetos, se crea con la fuerte intención de que este permita un correcto modelado orientado a objetos.

4.5.3 - El nuevo enfoque de UML 2.0

En las versiones previas de UML, se hacía un fuerte hincapié en que UML *no* era un lenguaje de programación. Un modelo creado mediante UML no podía ejecutarse. En el UML 2.0, esta asunción cambió de manera drástica y se modificó el lenguaje, de manera tal que permitiera capturar mucho más comportamiento (Behavior). De esta forma, se permitió la creación de herramientas que soporten la automatización y generación de código ejecutable, a partir de modelos UML.

4.5.4 - Estándares que conforman UML

- Superestructura: La superestructura de UML es la definición formal de los elementos de UML.
- Infraestructura: Conceptos de bajo nivel. *Meta-Modelo* da soporte a la superestructura, entre otras.
- OCL: Lenguaje de restricción. De utilidad para especificar conceptos ambiguos sobre los distintos elementos del diagrama.
- XMI / Intercambio de diagramas: Permite compartir diagramas entre diferentes herramientas de modelado UML.

4.5.5 - Organización de la superestructura

El bloque de construcción básico del UML es un diagrama. La estructura de los diagramas UML está reflejado por el diagrama de la figura 4.10, de acuerdo con la especificación del UML 2.0 del Object Development Group. Los detalles sobre estos diagramas específicos se organizan de acuerdo a esta estructura taxonómica, que da la perspectiva a los diagramas y a sus interrelaciones. Los

diagramas de interacción comparten propiedades y atributos similares, como lo hacen los diagramas estructurales y de comportamiento.

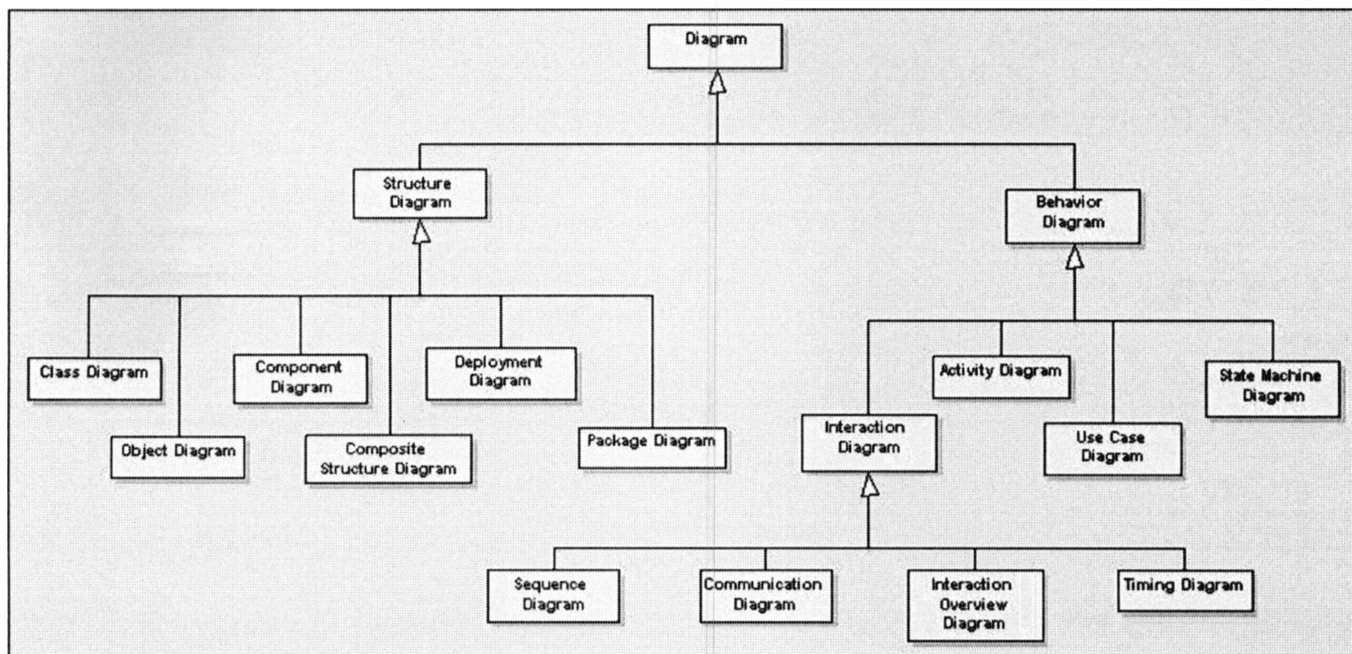


Figura 4.10 – Organización de la superestructura

4.5.6 - Diagramas de estructura y de comportamiento

Los diagramas estructurales representan elementos y así componen un sistema o una función. Estos diagramas pueden reflejar las relaciones estáticas de una estructura, como lo hacen los diagramas de clases o de paquetes, o arquitecturas en tiempo de ejecución, tales como diagramas de Objetos o de Estructura de Composición. Los diagramas de comportamiento representan las características de comportamiento de un sistema o proceso de negocios y, a su vez, incluyen a los diagramas de: actividades, casos de uso, maquinas de estados, tiempos, secuencias, repaso de interacciones y comunicaciones.

El siguiente cuadro muestra una breve descripción sobre los diagramas:

Diagrama	**Descripción**
Diagrama de Clases	Muestra una colección de elementos de modelado declarativo (estáticos), tales como clases, tipos y sus contenidos y relaciones.
Diagrama de Componentes	Representa los componentes que componen una aplicación, sistema o empresa. Los componentes, sus relaciones, interacciones y sus interfaces públicas.
Diagrama de Estructura de Composición	Representa la estructura interna de un clasificador (tal como una clase, un componente o un caso de uso),

	incluyendo los puntos de interacción de clasificador con otras partes del sistema.
Diagrama de Despliegue Físico	Un diagrama de despliegue físico muestra cómo y dónde se desplegará el sistema. Las máquinas físicas y los procesadores se representan como nodos y la construcción interna puede ser representada por nodos o artefactos embebidos. Como los artefactos se ubican en los nodos para modelar el despliegue del sistema, la ubicación es guiada por el uso de las especificaciones de despliegue.
Diagrama de Objetos	Un diagrama que presenta los objetos y sus relaciones en un punto del tiempo. Un diagrama de objetos se puede considerar como un caso especial de un diagrama de clases o un diagrama de comunicaciones.
Diagrama de Paquetes	Un diagrama que presenta cómo se organizan los elementos de modelado en paquetes y las dependencias entre ellos, incluyendo importaciones y extensiones de paquetes.
Diagrama de Actividades	Representa los procesos de negocios de alto nivel, incluidos el flujo de datos. También puede utilizarse para modelar lógica compleja y/o paralela dentro de un sistema.
Diagrama de Comunicaciones (anteriormente: Diagrama de colaboraciones)	Es un diagrama que enfoca la interacción entre líneas de vida, donde es central la arquitectura de la estructura interna y cómo ella se corresponde con el pasaje de mensajes. La secuencia de los mensajes se da a través de un esquema de numerado de la secuencia.
Diagrama de Revisión de la Interacción	Los Diagramas de Revisión de la Interacción enfocan la revisión del flujo de control, donde los nodos son Interacciones u Ocurrencias de Interacciones. Las Líneas de Vida los Mensajes no aparecen en este nivel de revisión
Diagrama de Secuencias	Un diagrama que representa una interacción, poniendo el foco en la secuencia de los mensajes que se intercambian, junto con sus correspondientes ocurrencias de eventos en las Líneas de Vida.
Diagrama de Máquinas de Estado	Un diagrama de Máquina de Estados ilustra cómo un elemento, muchas veces una clase, se puede mover entre estados que clasifican su comportamiento, de acuerdo con disparadores de transiciones, guardias de restricciones y otros aspectos de los diagramas de Máquinas de Estados, que representan y explican el movimiento y el comportamiento.
Diagrama de Tiempos	El propósito primario del diagrama de tiempos es mostrar los cambios en el estado o la condición de una línea de vida (representando una Instancia de un

	Clasificador o un Rol de un clasificador) a lo largo del tiempo lineal. El uso más común es mostrar el cambio de estado de un objeto a lo largo del tiempo, en respuesta a los eventos o estímulos aceptados. Los eventos que se reciben se anotan, a medida que muestran cuándo se desea mostrar el evento que causa el cambio en la condición o en el estado.
Diagrama de Casos de Uso	Un diagrama que muestra las relaciones entre los actores y el sujeto (sistema), y los casos de uso.

4.5.7 - El Plugin

Es una implementación basada en EMF del metamodelo 2.x para la plataforma eclipse. Los objetivos de este subcomponente son:

- Proveer una implementación utilizable del metamodelo para soportar el desarrollo de herramientas de desarrollo.
- Un esquema XMI [33] en común para facilitar el intercambio de modelos semánticos.
- Casos de prueba como medio de validación de la especificación y reglas de validación como medio de definición.
- Imponer niveles de conformidad.

CAPÍTULO 5
Nuestra propuesta: un Plugin Eclipse para especificar transformaciones visualmente

Este capítulo es una introducción a nuestra propuesta. Se explica brevemente las bases que inspiraron nuestro trabajo y muestra una descripción general de lo que se pretende hacer.

El generador de transformaciones tiene dos soportes sobre las cuales apoya sus fundamentos. Una base teórica que surge de la idea de crear un lenguaje de transformaciones a partir de la creación de un perfil UML [28], basado en la definición de estereotipos. Y una base práctica que nace de la idea de crear una herramienta que pueda ser ensamblada en la herramienta e-Platero que está desarrollada como Plugin de Eclipse.

5.1 - Lenguajes de transformaciones

Las transformaciones entre modelos requieren de lenguajes específicos para su definición. Estos lenguajes deben tener base formal, por ejemplo, tener un metamodelo que los sustente, y permitir un tratamiento automatizado. En este trabajo se decidió utilizar un lenguaje puramente declarativo para expresar transformaciones entre modelos que inspira su metamodelo inicial en el lenguaje QVT que es la especificación de OMG para transformaciones, aún en proceso de definición. El lenguaje que se propone utilizar está orientado a ser el mínimo para poder expresar relaciones y consultas de transformación entre modelos.

5.1.1 – Un ejemplo de transformación usando QVT

Para un mejor entendimiento del concepto de transformación de modelos, se presenta un ejemplo extraído del manual de QVT. Este ejemplo muestra la especificación textual de una transformación llamada *UMLToRdbms* que define una relación top-level llamada *Class2Table* que transforma clases UML (que cumplan la restricción de ser persistentes, indicado en la cláusula *when* de esta relación) a tablas del Modelo Relacional con el mismo nombre. Además la transformación tiene otra relación *Attr2Col*, la cual especifica que los atributos (no multivaluados con tipo de datos básico) de la clase, se corresponden con columnas del mismo nombre y tipo en la tabla correspondiente. Esta relación es invocada en la cláusula *where* de la relación *Class2Table* y significa que cada vez que *Class2Table* se cumple para una clase y una tabla, la relación *Attr2Col* para sus atributos y columnas respectivamente, se cumple también:

```
Transformation UML2Rdbms (Uml: UML2.0, Rel: RDBMS)
{
        Top Relation Class2Table {
        checkonly domain Uml c: Class {name = n }
        checkonly domain Rel t: Table {name = n }
        when { isPersistent = true }
        where { Attr2Col (c, t) }
}
Relation Attr2Col
{
        checkonly domain Uml c: Class {
        attribute = a: Attribute {name = an, type = p: DataType {name = dt}}}
        checkonly domain Rel t: Table {column: col:Column
        {name = an, type.name = dt } }
        when { not a.isMultivalued() }
        where { col.type = a.type } }
}
```

Gráficamente, una instanciación de esta transformación podría ser, por ejemplo, la que se muestra en la figura 5.1 entre dos modelos concretos Uml1 (de UML) y Rel1 (de Rdbms), donde n = 'Persona' y un nombre de atributo podría ser 'nombre'. O sea, para la clase Persona del modelo Uml1, existe una Tabla en el modelo relacional Rel1 con el mismo nombre. Lo mismo sucede con los atributos: para cada atributo de la clase Persona existe una columna en la tabla Persona con el mismo nombre. O sea, para el atributo 'nombre', existe la columna 'nombre', ambos de tipo String.

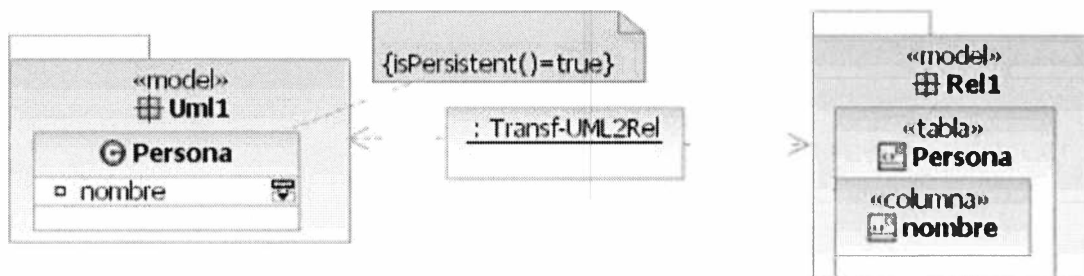


Figura 5.1 -Una Instanciación gráfica de la Transformación

5.1.2 - Una alternativa simple a QVT: SQVT

Lo que propone Giandini en [7] es extender la Infraestructura 2.0 y usar OCL 2.0 para implementar el metamodelo de las transformaciones. Un lenguaje declarativo puro para transformaciones, cuyo metamodelo inicial se inspira en el paquete Relations de QVT. Luego se implementa como una extensión (perfil) de la Infrastructura y se utiliza OCL para completar dicha implementación.
Además permite crear transformaciones con una sintaxis más amigable maximizando el uso del lenguaje OCL.

SQVT se basa en la creación un perfil específico (instancia de la Clase Profile de la Infraestructura) que permite expresar características particulares de la transformación de modelos.

5.1.3 - *El modelo SQVT*

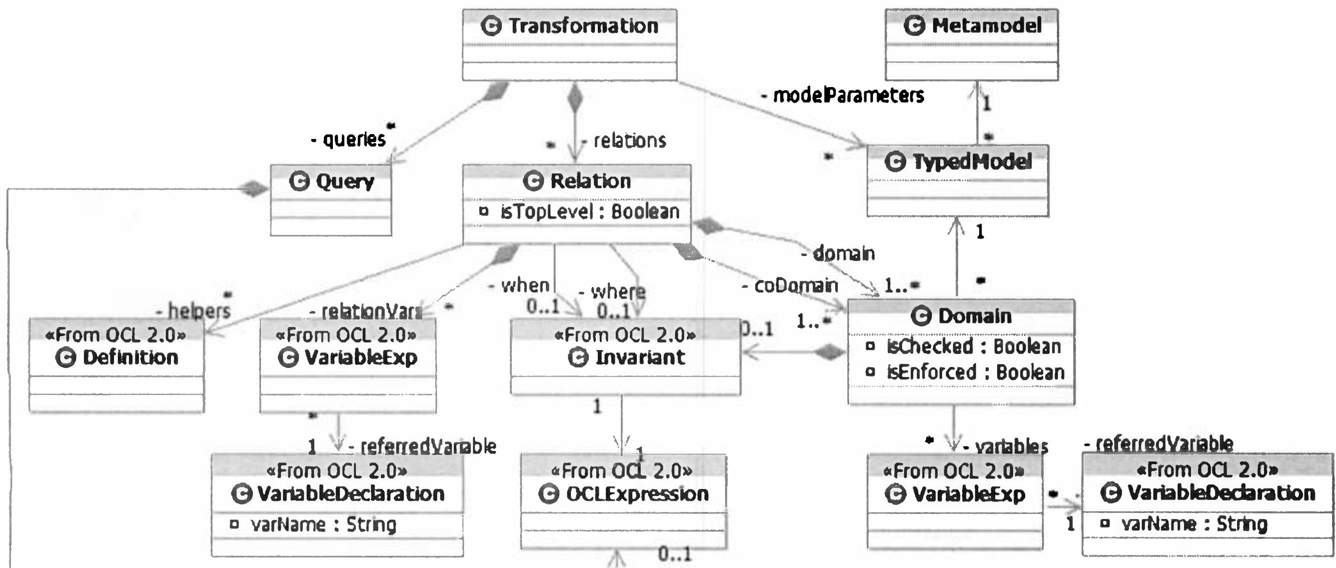


Figura 5.2 – Modelo SQVT

La figura 5.2 muestra el metamodelo propuesto en [7] por Giandini, donde una Transformación se compone de: modelos tipados (typedModel: el tipo es algún metamodelo, instancia de MOF) que participan de la transformación como sources y/o targets, relations y querys. Una *Relation* tiene un atributo booleano llamado *isTopLevel*, representando su nivel. Una relación de alto nivel se ejecuta automáticamente; sino la relación debe ser llamada explícitamente por otra relación para ser ejecutada. Las relations pueden incluir declaraciones de variables, cláusulas *when* y *where*, predicados que deben cumplirse para que se produzca la relación de Transformación y predicados que deben cumplirse al finalizar la ejecución de la transformación, respectivamente y helpers. Un *helper*, es un elemento no considerado en el metamodelo de QVT, define operaciones adicionales para realizar navegación sobre los modelos que participan de la transformación. Los helpers pueden tener parámetros de entrada y pueden usar recursión. Se componen de declaraciones de variables y de expresiones que especifican dichas operaciones adicionales de consulta/navegación. La aplicación de un helper no produce efectos laterales.

5.1.3.1 - *Dominios chequeados y forzados*

Además las relaciones se definen en uno o mas Dominios; cada Dominio se corresponde a un modelo y puede ser sólo chequeado o bien, forzado. Cuando un dominio sólo se chequea, sus restricciones y/o expresiones booleanas serán evaluadas con respecto a los valores actuales del modelo asociado. Cuando un dominio es forzado también es chequeado, pero en este

caso, el modelo asociado puede ser actualizado (es decir puede producirse la creación y /o borrado de objetos) para concretar la transformación.

5.1.3.2 - *Definición en forma completa del modelo SQVT*

Se presenta aquí una definición formal de los nuevos estereotipos mencionados previamente, indicando a que metaclase estereotipan (cual es su base), sus atributos y algunas de las restricciones, expresadas en OCL, que deben cumplirse al utilizarlos.

- *STEREOTYPE* Transformation

  Base: Class

  Constraints:

  1. Una transformation debe contener relaciones o querys.

     ```
     self.relations -> notEmpty() or self.querys -> notEmpty()
     ```

  2. Una transformation debe tener al menos dos modelos como parámetros (un input y un output).

     ```
     self.modelParameter -> size() > 1
     ```

- *STEREOTYPE* Relation

  Base: Class

  OwnedAttributes:

  Name: isTopLevel, Type: Bolean

  Constraints:

  1. Una relation debe definirse en al menos un dominio.

     ```
     self.domains -> notEmpty()
     ```

  2. Una relation debe ser parte de una Transformation

     ```
     self.owningPackage.ownedMember-> exists ( a:Association |
     a.estereotype=<<composite>> and
     a.memberEnd-> exists (e| e.isComposite=true and
     e.type.stereotype= <<Transformation>> and
     e.opposite = self ) )
     ```

  3. Una relation topLevel no puede estar anidada en otra.

```
(self.owningPackage.ownedMember-> select ( r:Class | r.
stereotype= <<relation>>)) -
>forAll (r| self.owningPackage.ownedMember-> exists (
a:Association | a.memberEnd->
exists (e| e.isComposite=true and e.type =r and
e.opposite.stereotype =<<relation>>
implies e.opposite.isTopLevel=false ) ) )
```

- *STEREOTYPE* Domain

  Base: Class

  OwnedAttributes:

  Name: isEnforced, Type: Boolean
  Name: isChecked, Type: Bolean

  Constraints:

  1. Un Domain debe hacer referencia a un TypedModel (existe una DirectedRelationship <<domainModel>> entre Domain y TypedModel)

  ```
  self.owningPackage.ownedMember-> exists (
  d:DirectedRelationship | d.estereotype=<<
  domainModel>> and d.source=self and d.target.type.stereotype=
  <<TypedModel>> )
  ```

- STEREOTYPE typedModel

  Base: Package

  Constraints:

  1. Un modelo tipado debe ser instancia de un metamodelo, o sea debe ser el source de una DirectedRelationship con stereotipo <<type>> y target un paquete <<metamodel>>

  ```
  self.owningPackage.ownedMember-> exists ( d:DirectedRelations
  hip | d.estereotype=<<type>>
  and d.source=self and d.target.type.stereotype= <<metamodel>> )
  ```

  2. Un modelo tipado solo debe contener instancias del metamodelo con el que se relaciona.

  ```
  self.ownedMember-> forAll (e| self.metamodel -> includes
  (e.type))
  ```

  donde la operación metamodel retorna el paquete que contiene al metamodelo del modelo tipado (self):

metamodel: Package -> Package
metamodel = self.owningPackage.ownedMember ->select
(d:DirectedRelationship|
d.stereotype=<<type>> and d.source=self) ->collect
(d:DirectedRelationship| d.target))

- STEREOTYPE type

  Base: DirectedRelationship

  Constraints:

  1. Se establece entre TypedModel y un único metamodelo

     self.source-> forAll (m| m.stereotype= <<TypedModel>>)and
     self.target.stereotype= <<metamodel>>

- STEREOTYPE predicate, subclase del stereotype <<invariant>>

  Base: ExpresionInOCL

  OwnedAttributes:

     Name: isWhen, Type: Boolean
     Name: isWhere, Type: Bolean

  Constraints:

  1. Un predicate debe restringir una relation

     self.constraint.constrainedElement.stereotype=<<relation>>

  2. Un predicate no puede etiquetarse como when y where al mismo tiempo

     Self.isWhen=true implies Self.isWhere=false and Self.isWhere=
     true implies
     Self.isWhen=false

- STEREOTYPE helper, subclase del stereotype <<definition>>

  Base: ExpresionInOCL

  Constraints:

  1. Un helper se define (tiene contexto) en una <<relation>>.

     self.constraint.context.stereotype= <<relation>>

5.1.4 - Un ejemplo de transformación usando SQVT

```
Transformation UMLToRdbms (Uml1: UML2.0, Rel1: RDBMS)
{
	Relation UML2Rel {
		checkonly domain Uml1 c: Class {}
		checkonly domain Rel1 t: Table {}
		when {c.isPersistent( )}
		where {c.name = t.name and c.allAttribute-> forAll (a | (t.column ->
		exists (co |  Attr2Col(a, co)}
	}
	Relation Attr2Col (a, co ) {
		checkonly domain Uml1 a: Attribute{}
		checkonly domain Rel1 co: Column {}
		when {a.isMultivalued( )}
		where {a.type = co.type and co.name = a.name}
	}
}
```

Esta transformación especifica cómo hacer un pasaje de un modelo UML a un modelo Relacional. Para ello describe dos relaciones: la primera, referencia a todas las clases (c:Class) del modelo UML y a todas las tablas (t:Table) del modelo relacional. Y dice que cuando la clase sea persistente entonces el nombre de la clase y la tabla deben ser iguales y además que para todos los atributos de la clase c debe existir una columna en t que cumpla la relación *Attr2Col*.
Para que un atributo (a:Attribute) y una columna (co:Column) cumplan la relación Attr2Col a debe ser multivaluado y además el tipo de "a" tiene que ser el mismo que el de "co" y sus nombres también tienen que coincidir.

5.1.5 - Las desventajas de QVT

En el documento de especificación de QVT, que es la propuesta estándar de OMG para transformaciones, el metamodelo se define como "extensión de MOF y de OCL". Como explica Giandini [7], según lo analizado anteriormente y según las definiciones de OMG no es técnicamente correcto extender a MOF ya que representa un Meta-metamodelo cerrado sobre el que se instancian metamodelos.
Por otra parte, SQVT propone modificar los siguientes aspectos respecto de QVT:

- Las relaciones de transformación se expresan mediante expresiones template en cada uno de los dominios, entonces para concretar la transformación, se debe producir un pattern matching entre estas expresiones. Además, las variables de los dominios pueden definirse recursivamente, por lo que el anidamiento de expresiones template y los pattern matching entre ellas puede propagarse generando especificaciones complicadas de entender.

- Los dominios aparecen precedidos por la palabra checkonly, lo cual indica que sólo se chequea consistencia entre modelos, no permitiendo la creación de nuevos elementos. Al respecto, considerando que en la práctica el proceso de transformación implica la generación de modelos, sugerimos no utilizar los atributos para dominio definidos en QVT para permitir (enforced) o impedir (checkonly) la creación o eliminación de elementos. Propone admitir, en general, la modificación de los modelos cuando sea necesario, para poder cumplir con la transformación.

- La relación Attr2Col sugiere una iteración sobre todos los atributos de la clase, que es pasada como parámetro, pero no lo indica explícitamente. Para denotar esto se puede usar el lenguaje OCL, quitándole ambigüedad.

- Otra desventaja de esta notación, puede verse en la relación Attr2Col, que transforma atributos en columnas, pero es invocada con parámetros Clase y Tabla. Resultaría más claro que cada relación tenga como parámetros los elementos que realmente participan de la transformación.

5.1.6 – Ubicación del modelo SQVT dentro de la Arquitectura de 4 capas de Modelado

En la capa M3 de la arquitectura de 4 capas de modelado (representada gráficamente en la figura 5.3), se ubica MOF, que representa un Meta-metamodelo cerrado sobre el que se instancian metamodelos.
En consecuencia, el metamodelo para Transformaciones debería ubicarse en la capa M2, junto con el resto de los metamodelos (por ejemplo el metamodelo de UML, el de OCL, etc.).
Sin embargo, una instancia específica de transformación se debe ubicar en la capa M2 para poder relacionar metamodelos concretos (que se ubican en la capa M2) como el de UML y el de RDBMS, entre cuyas instancias (modelos tipados) se produce la transformación (por ejemplo una Class de UML y una Table de RDBMS). Es decir, los modelos que concretamente están involucrados en la transformación (capa M1) son parámetros para el lenguaje de transformación.
Por lo tanto, la definición del metamodelo para transformaciones puede pensarse en la capa M3 de esta Arquitectura ya que resulta lógico pensar que el metamodelo y su instanciación no pueden convivir en la misma capa, dado que representan distintos niveles de abstracción.
Siguiendo este razonamiento, si se define el nuevo lenguaje para transformaciones como una extensión de elementos del paquete Core de la Infraestructura, su metamodelo se ubicará en el nivel M3, junto a la Infraestructura. Esto es posible dado que MOF usa elementos del paquete Core para su definición, situación que nos permite identificar a la Infraestructura como un meta-metamodelo. Luego, una instancia del metamodelo para Transformaciones en la capa M2 relaciona elementos de metamodelos, mientras que en la capa M1 existirá un enlace de correspondencia entre modelos, instancias de dichos metamodelos, que son los que concretamente se transforman.

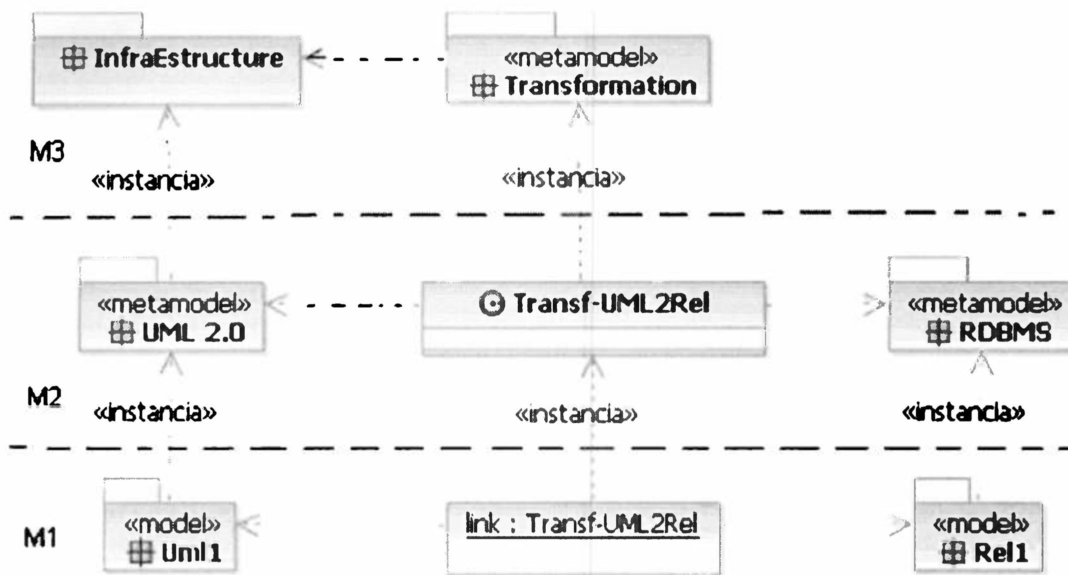


Figura 5.3 - La Transformación de Modelos en la Arquitectura de 4 capas

5.2 - La herramienta 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 Eclipse, que soporta el proceso de desarrollo conducido por modelos usando una notación gráfica con base formal.

5.2.1 - Objetivos de e-Platero

Los principales objetivos de e-Platero 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

5.2.2 - Arquitectura de e-Platero

La arquitectura de e-Platero respeta el estándar de Eclipse para el desarrollo de plugins. Está formado de 9 plugins relacionados entre sí, como se muestra en la figura 5.4:

- 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

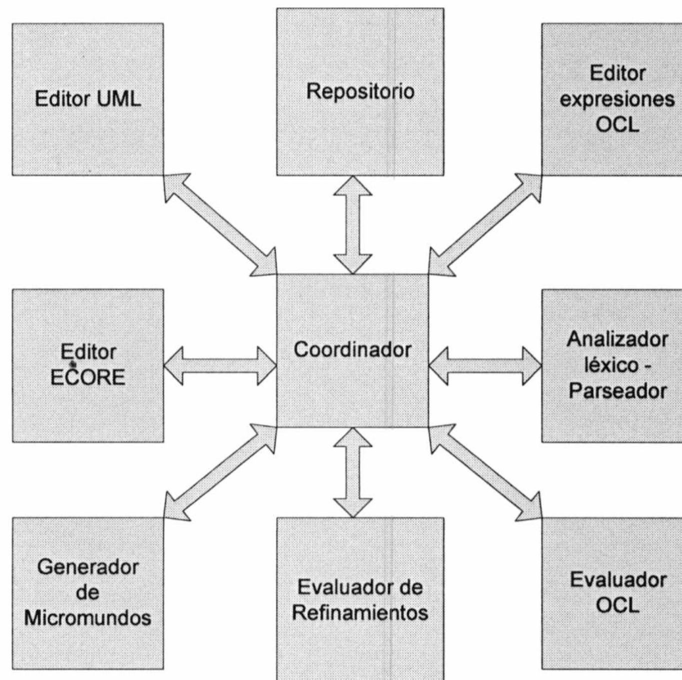


Figura 5.4 - Arquitectura de e-Platero

5.3 - Nuestra propuesta: un Plugin para la creación de transformaciones visualmente

De todo lo leído y analizado, teniendo en cuenta la definición del metamodelo de transformaciones propuesto por Giandini [7] y la implementación de la herramienta e-Platero se decidió crear una herramienta que pueda ser ejecutada como un Plugin de Eclipse, con la intensión de que en un futuro pueda ser agregada a e-Platero, y además que permita la creación y evaluación de transformaciones basado en un metamodelo generado a partir de la extensión (perfil) de la Infrastructura y utilización de OCL para completar dicha implementación.

5.3.1 - Ejemplo de una transformación generada por el plugin de transformaciones

El mismo ejemplo, citado anteriormente, utilizando SQVT, quedaría representado gráficamente como describe la figura 5.5:

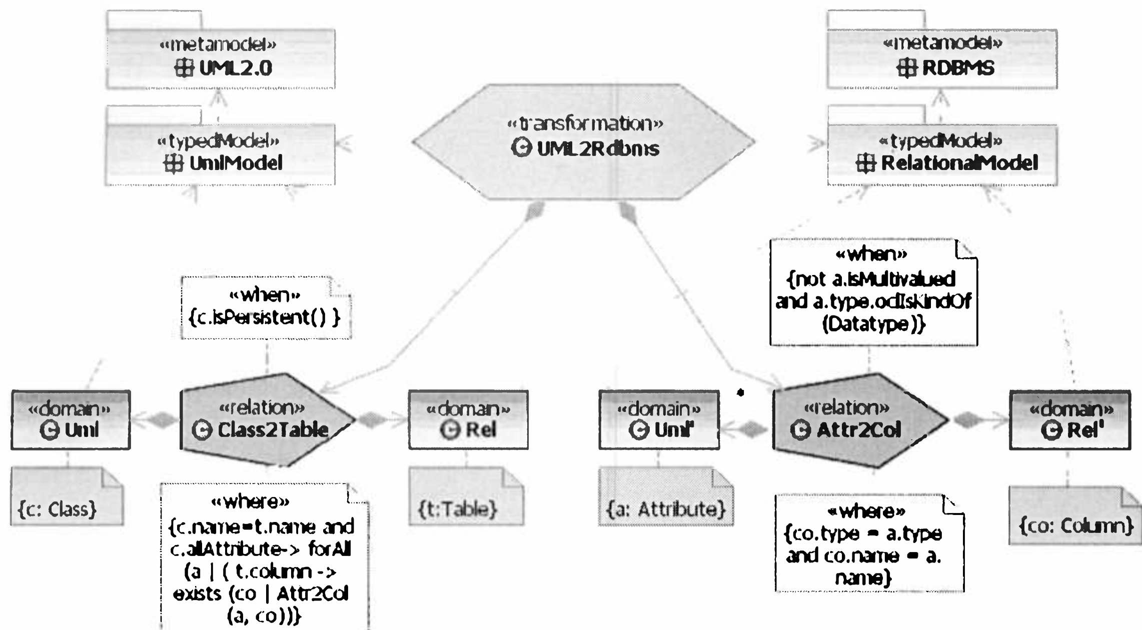


Figura 5.5 - Vista de una transformación visual

Al empezar a estudiar la complejidad del problema a resolver nos dimos cuenta que debíamos dividirlo para poder encontrar pequeñas soluciones mas rápidamente y de a poco ir generando la solución total.
La primera división fue la de separar la vista de la lógica del plugin. La vista consiste en la problemática de cómo mostrar y qué herramientas proveer al usuario final para poder armar su propia transformación y la lógica es el funcionamiento y validaciones que hay que realizar. Es por esto que decidimos separar el diseño en dos fracciones: La Vista y La Lógica.

5.3.1.1 - La vista

El plugin se divide en tres componentes básicos: un editor que provee las herramientas gráficas para poder armar una transformación, un cargador de metamodelos para poder asociarle a la transformación los metamodelos de entrada y salida; y por último, un cargador de modelos para poder evaluar una transformación. Esto es mostrado a continuación por la figura 5.6:

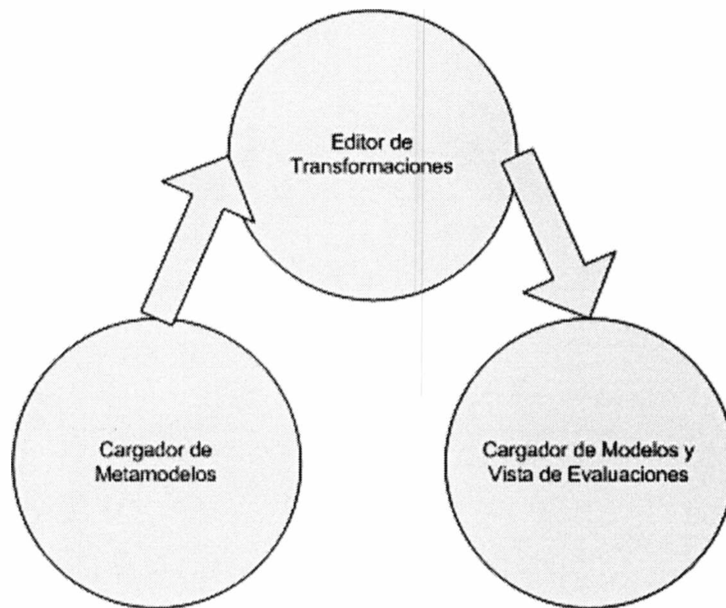


Figura 5.6 – Secuencia de la visualización

El Editor utiliza el Cargador de metamodelos para poder generar las transformaciones visualmente. El Editor le permite al usuario agregar relaciones, predicados, declaraciones de variables y queries a una transformación. Permite también escribir expresiones OCL que representan los predicados o las queries. Al realizar una transformación utilizando el Editor, se genera un archivo con extensión *.transformation*. Estos archivos pueden ser evaluados por el Evaluador que requiere que el usuario elija dos modelos, instancias de los metamodelos cargados por el editor. El Evaluador muestra los resultados de cada una de las evaluaciones de las relaciones definidas en la transformación definida por el Editor.

5.3.1.2 - La lógica

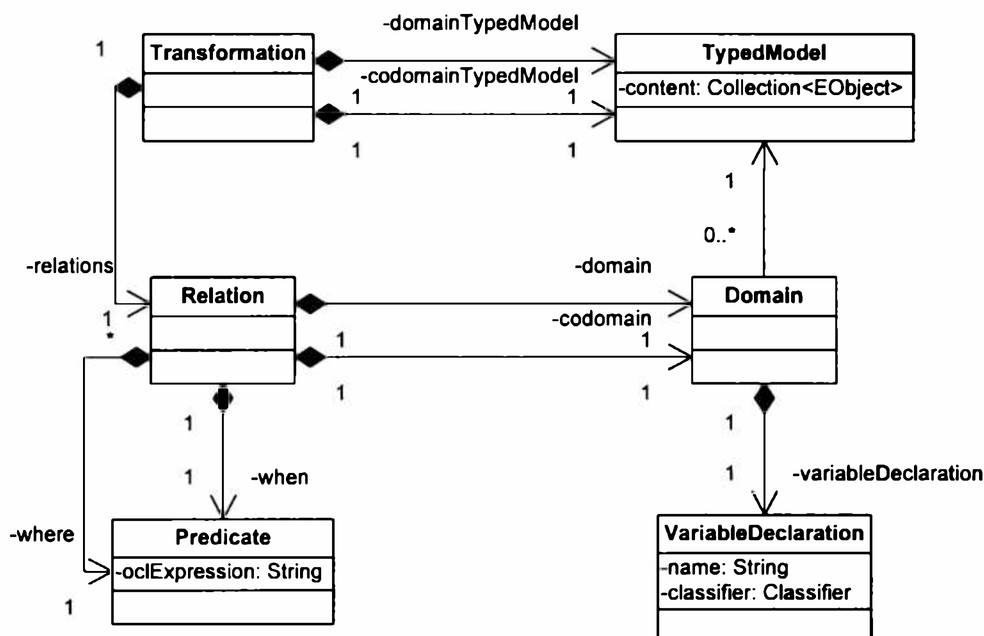


Figura 5.7 – Nuestro modelo de transformación

Nuestra transformación, expuesta en la figura 5.7, tiene relaciones y dos modelos tipados. Cada relación tiene un dominio y un codominio y además dos tipos de predicados: when y where que equivalen a la pre y post condición que debe cumplir una relación para ser verdadera en su evaluación. Cada dominio tiene una declaración de variable que tiene un nombre y un tipo que debe ser un clasificador. Además, cada dominio, pertenece a un modelo tipado. Todos los dominios de todas las relaciones pertenecen al modelo tipado de dominio y todos los codominios pertenecen al modelo tipado de codominio, definidos en la transformación. Cada predicado tiene un texto que representa una expresión OCL que puede hacer referencias a las variables declaradas en el dominio y codominio de la relación. Inclusive puede hacer referencia a otras relaciones invocándolas como si fueran funciones con parámetros formales, si fuera necesario.

Limitaciones del modelo

Con este modelo se puede representar el modelo de transformación propuesto por Giandini, salvando algunas excepciones. Nuestro modelo solo soporta el chequeo de los dominios. Es decir que las relaciones no pueden ser forzadas como propone QVT (atributos de Domain, isCheckeable e isEnforceable) o SQVT que propone modificar o generar un modelo de salida. En nuestra implementación podemos validar si dos modelos cumplen una transformación pero no podemos hacer que el modelo de salida se modifique o se cree a partir de un modelo de entrada. Esta limitación queda propuesta como futura mejora a la herramienta.
El plugin no hace distinción entre las categorías de las relaciones, atributo isTopLevel de una Relation. Esta propiedad no es tenida en cuenta ya que el plugin no deriva código.

CAPÍTULO 6
Implementación del Editor

En este capítulo se explicará como fue implementado el editor gráfico, desde la creación del *ecore* (contiene la información del modelo con las clases definidas) hasta la derivación del código propio del editor.
La implementación del plugin se dividió en 2 partes; por un lado se creo el Editor Gráfico como plugin el cual lleva a la creación de un archivo con extensión *transformation*; por otro lado se creo el validador, el cual se encarga de validar el archivo generado por el editor grafico. Los dos plugins fueron creados separadamente y luego fueron unidos sin ningún tipo de problema.

6.1 - Creación del Ecore

Una vez que se investigó y analizó el modelo del transformación realizado en el paper "Un Lenguaje para Transformación de Modelos basado en MOF y OCL", se prosiguió con la creación del ecore (contiene la información del modelo con las clases definidas), como muestra la figura 6.1, de la siguiente manera:

- Se definió un nombre para el package, el cual va a ser el nombre de la extensión del archivo de transformaciones, en nuestro caso se llamó "*transformation*". Además del nombre al package se le debe definir la URI, en cuyo caso fue "*ar.edu.unlp.info.tesis.transformation*".

- Se definió la *EClass Transformation*, que es la que va a contener todas las demás EClass (clase perteneciente al ecore) del metamodelo. Dicha EClass esta compuesta por un nombre, una colección de relaciones, una colección de dominios, una colección de querys, una colección de invariantes y por una colección de variables de declaración.

- Se definió la EClass Relation, que corresponde con el estereotipo del mismo nombre definido en el paper, dicha EClass esta compuesta por un nombre, un doman y un codomain que hacen referencia a diferentes instancias de la EClass Domain, un when y un where que hacen referencia a diferentes instancias de la EClass Invariant y por último un EAtributte isTopLevel de tipo Boolean.

- Se definió la EClass Domain, la cual es referenciada, por la EClass Relation (nombrada anteriormente). Esta compuesta por un nombre y por una referencia a una instancia de la EClass VariableDeclaration.

- Se definió la EClass Query que esta compuesta por un nombre y una expresión de tipo String. Estas EClass no están referenciadas por ninguna otra instancia y a su vez no hacen referencia a ninguna otra; solamente están dentro de la transformación para consultas.

- Se definió la EClass Invariant, la cual esta referenciada por las relaciones a través de las conexiones when y where. Esta compuesta por una expresión de tipo String.

- Se definió la EClass VariableDeclaration que esta compuesta por un nombre de variable (VarName) y por un clasificador (EClassifier), los dos de tipo String. Esta EClass es referenciada por los dominios.

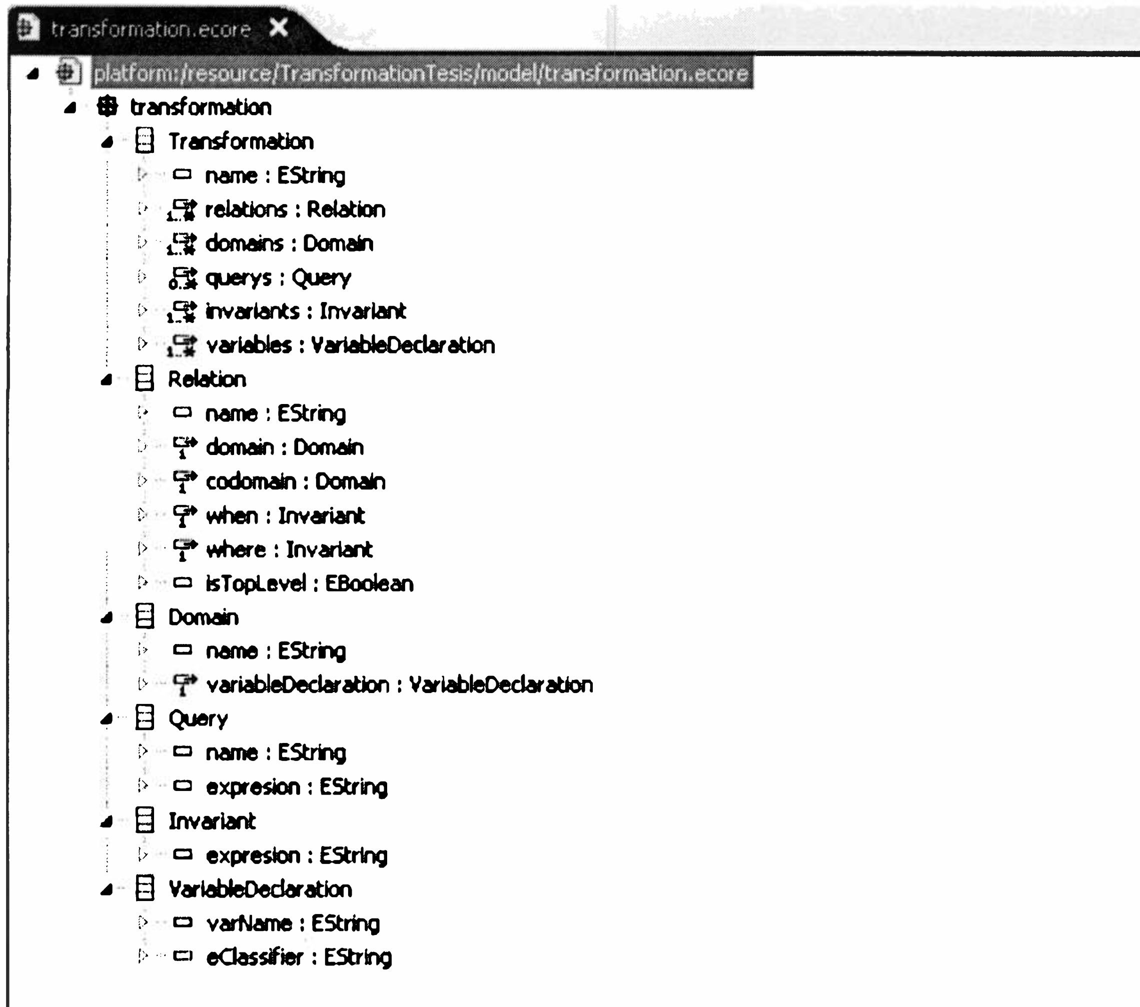


Figura 6.1 – Vista del archivo ecore

Una vez que se crearon todos los componentes de ECore, se puede validar para comprobar si la creación fue correcta o no. Se puede validar componente a componente o directamente sobre el package. Este último caso fue el que se aplico.

6.2 - Creación del genmodel

Al haber finalizado la creación del ECore, se debe proseguir con la creación del GenModel, como muestra la figura 6.2, el cual es un archivo que va a permitir la derivación de código del editor.
Para dicha creación se debe crear un archivo con extensión "*genmodel*", seleccionar el tipo de modelo a importar (que puede ser Annotated Java, Ecore model, Rose class model, UML model y XML schema) y una vez seleccionado (en nuestro caso es un modelo ECore) cargar dicho modelo. El archivo *genmodel* esta compuesto por los mismos componentes que el

ecore, a diferencia, que este archivo nos permitirá la generación del código necesario para la creación del plugin. La derivación del código se divide en tres partes:

6.2.1 - *Model Code*

En esta sección se deriva el código perteneciente a los componentes del genmodel; se crea una interfaz para cada uno, que contiene getters y setters de todas las propiedades y que además extiende una interfaz llamada EObject (perteneciente al paquete org.eclipse.emf.ecore). También se crea una implementación por cada componente que tiene como interfaces las anteriormente nombradas.
Otra clase creada es la TransformationFactory, que es la encargada de crear todos los demás clases (por ejemplo createRelation()). Ésta, al igual que las anteriores, tiene una interfaz y su implementación.
También es creado un paquete de utilidad el cual posee dos clases, una llamada TransformationAdapterFactory, que como su nombre lo indica, es un creador de adaptadores para los diferentes componentes del genmodel; y otra llamada TransformationSwitch que se encarga de realizar el cambio de un EObject entre diferentes EClass.

6.2.2 - *Edit Code*

En esta sección se deriva el código perteneciente a los ítems del plugin, los cuales se corresponden con los ítems del genmodel. Es decir para la clase Domain va a existir un DomainItemProvider y así sucesivamente. El ItemProvider nombrado posee funciones que permiten recuperar los descriptores de las propiedades (las cuales van a ser mostradas en la vista Property), agregar los descriptores de las propiedades, recuperar la imagen y el texto a ser mostrado cuando se selecciones el ítem, también soporta la función de ser notificado cuando ocurra algún cambio en el editor (por ejemplo que se haya seleccionado otro ítem).
En este proyecto se realizo una modificación para poder relacionar el editor con los metamodelos cargados (sección "Lectura de los metamodelos").

Los cambios que se realizaron fueron los siguientes:

- Se creo una clase llamada *ItemPropertyDescriptorTesis* la cual se encuentra en el paquete *ar.edu.info.unlp.tesis.* Esta es subclase de *ItemPropertyDescriptor* y se redefinió el método "getChoiceOfValues", el cual recupera los valores cargados en la lectura de los metamodelos a través de la clase *EcoreLoader.*

- Se modifico la clase *VariableDeclarationItemProvider*, se cambio el método "getPropertyDescriptors", el cual si no existen descriptores se crean. Para la creación del descriptor de la propiedad EClassifier se agrego el método "addEClassifierPropertyDescriptorTesis", el

cual crea una instancia de la clase *ItemPropertyDescriptorTesis* y lo inserta en la lista de descriptores.

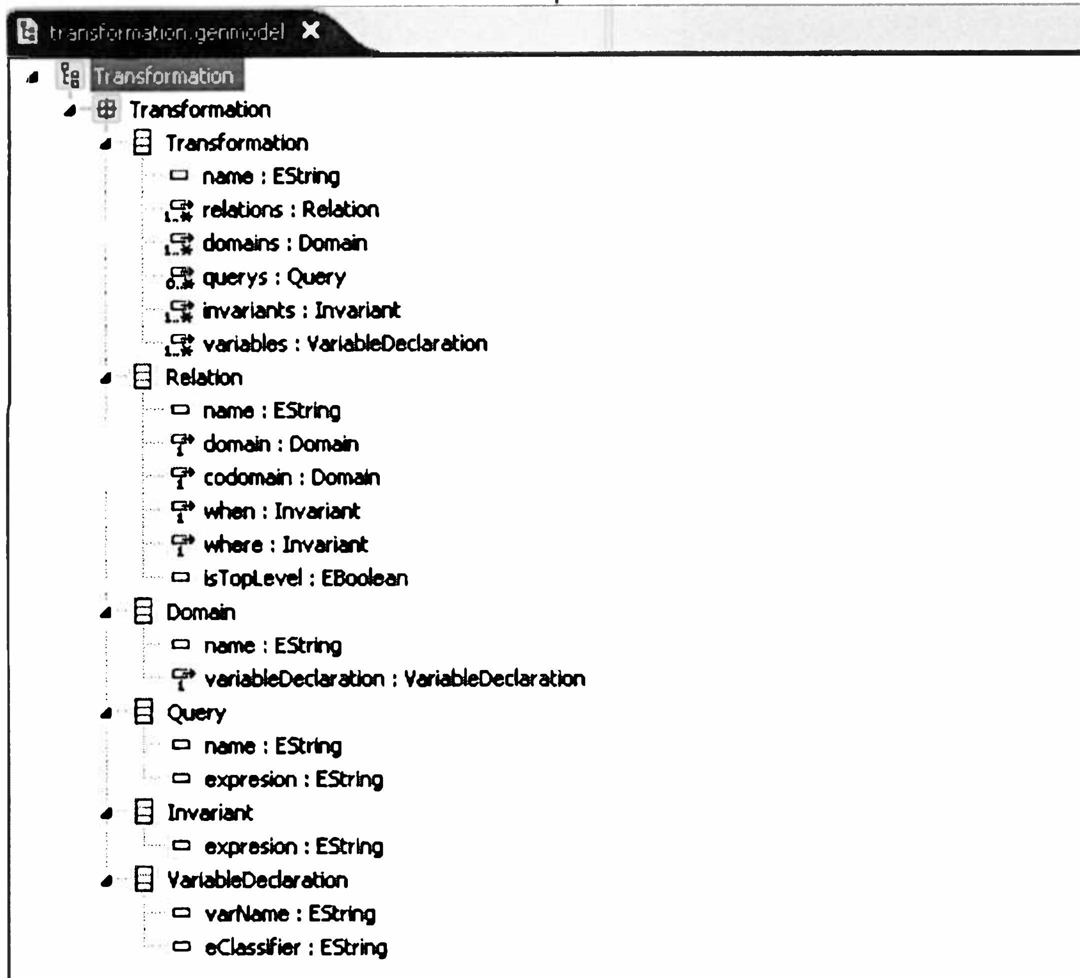


Figura 6.2 – Vista del archivo genmodel

6.2.3 - Editor Code

En esta sección se deriva el código perteneciente al editor en si. Se crean 4 clases que son encargadas de llevar a cabos las diferentes funciones del editor, las cuales son:

- *TransformationActionBarContributor*, que es la encargada de manejar las acciones que se realizan sobre el menú (como por ejemplo ver las propiedades de algún componente, etc.).

- *TransformationEditor*, es la que cumple las funciones del editor (mostrar las diferentes páginas, desplegar los ítems, agregar y eliminar ítems, disparar las actualizaciones, manejar las diferentes vistas, etc.).

- *TransformationEditorPlugin*, es la que cumple las funciones del plugin (adaptarse a otros plugins, iniciar y parar el editor, etc.).

- *TransformationModelWizard*, es la encargada de la creación del archivo "transformation", la cual te permite ingresarle un nombre, elegir el proyecto donde se situará el archivo, etc.

Cabe destacar que con esta derivación de código ya existe el editor y se pueden crear transformaciones, lo que no permite es la creación de las transformaciones de manera grafica.
A partir de las siguientes explicaciones ya se empieza a definir el editor grafico para la creación de transformaciones.

6.3 - Creación del gmftool

El archivo *gmftool* (ver figura 6.3) permitirá configurar el menú de herramientas del editor grafico. En el se definirá los componentes que podrán ser agregados dentro del editor; también se deberá especificar las relaciones que se podrán crear en el editor.
Una vez que se definieron los elementos a ser creados, se les debe asociar un icono, el cual será mostrado en el menú del editor. En nuestro caso, todos los iconos están en el proyecto *TransformationTesis.diagram* dentro de la carpeta "iconos/tesis".

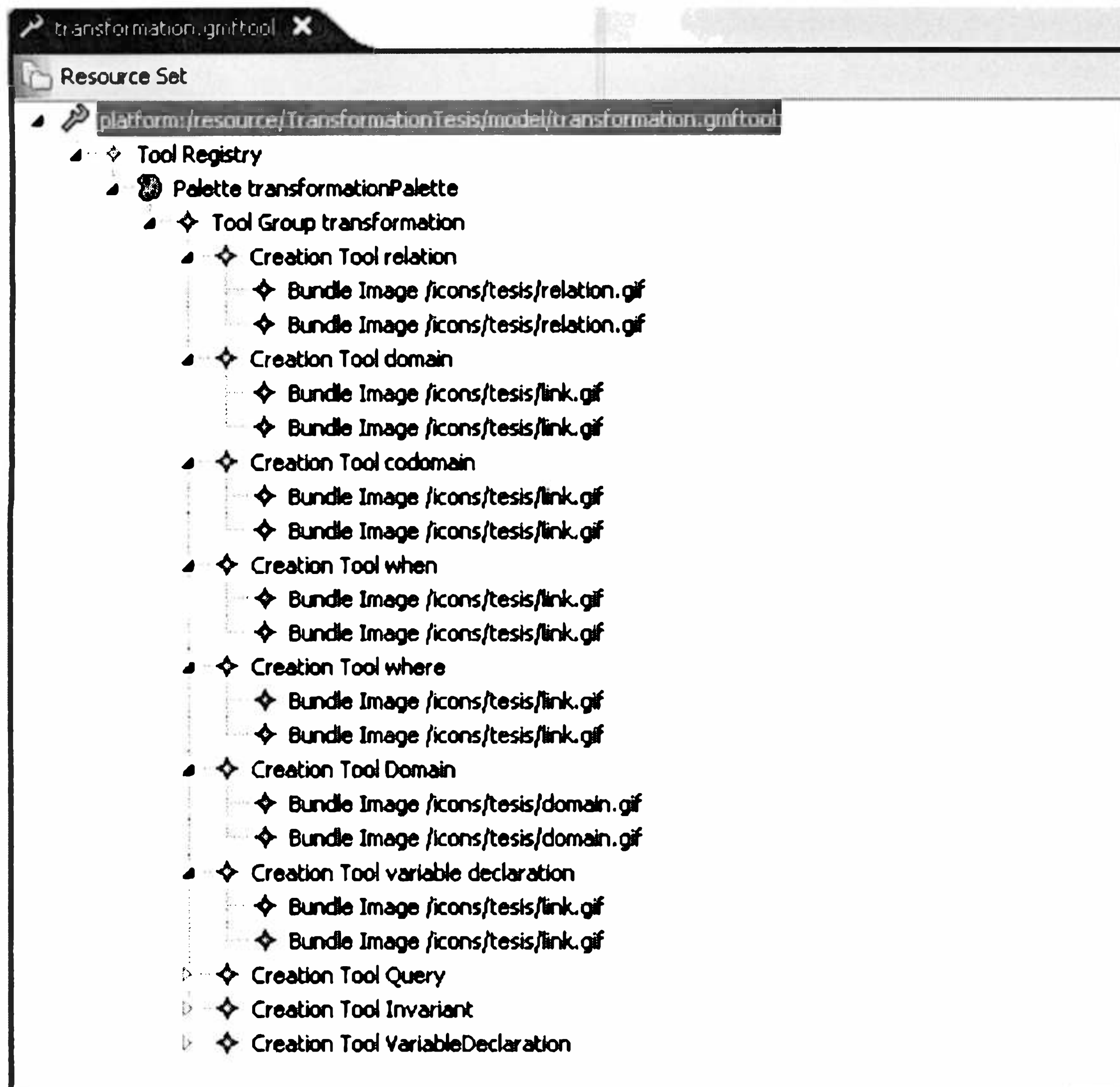


Figura 6.3 – Vista del archivo gmftool

6.4 – Creación del gmfgraph

El archivo *gmfgraph* (ver figura 6.4) es en el cual se definen los gráficos a mostrar dentro del editor. Se divide en dos partes:

- Galería de figuras: En esta galería se definen las formas que tendrán las figuras, en nuestro caso se definieron rectángulos para los *queries* y los dominios, para las relaciones se definieron pentágonos y para las invariantes como las declaraciones de variables se definieron rectángulos con el vértice superior derecho plegado. A cada una de las figuras se le agregaron las propiedades a ser mostradas, las cuales pueden ser dinámicas (que se escriben en el editor) o fijas (solamente de lectura, como por ejemplo el texto 'Query' en la figura del mismo nombre).

  Otras de las figuras que se determinaron fueron las conexiones, para ellas se crearon poli-líneas.

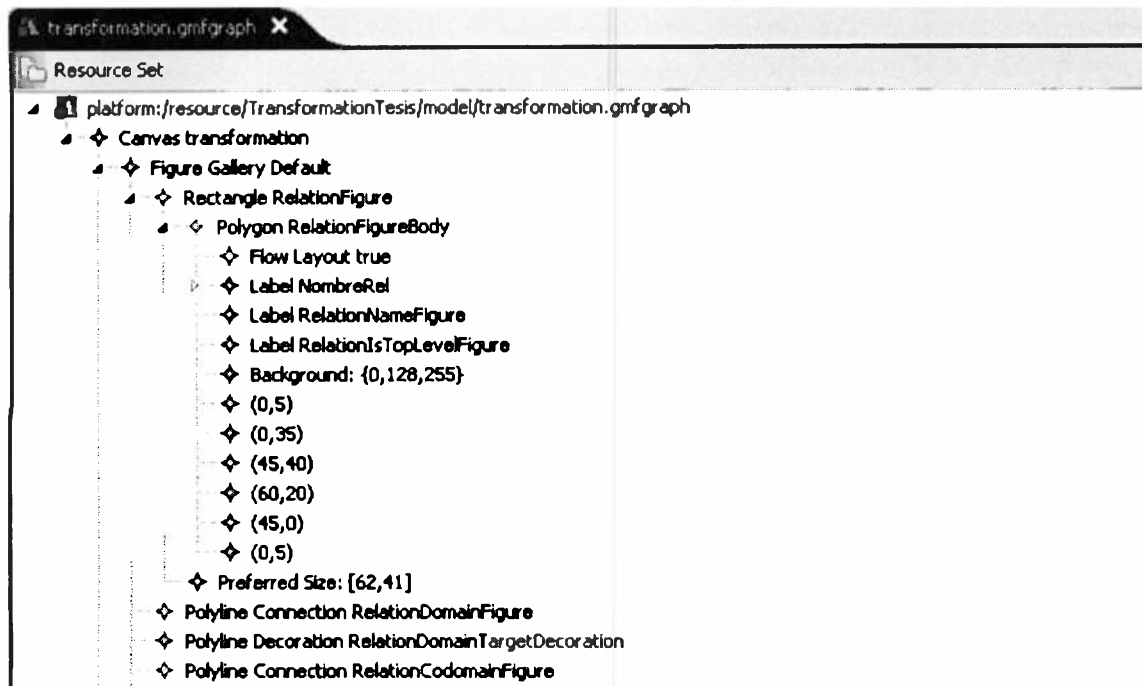


Figura 6.4 – Vista de la galería de figuras del archivo gmfgraph

- Mapeo: En esta parte existen tres tipos de mapeos: los nodos que son relacionados con alguna figura de la galería, las conexiones que son mapeadas con las poli-líneas y por ultimo se pueden los labels, los cuales se asocian con las propiedades dinámicas a mostrar dentro de las figuras. Este mapeo se va a realizar en el archivo *gmfmap*.

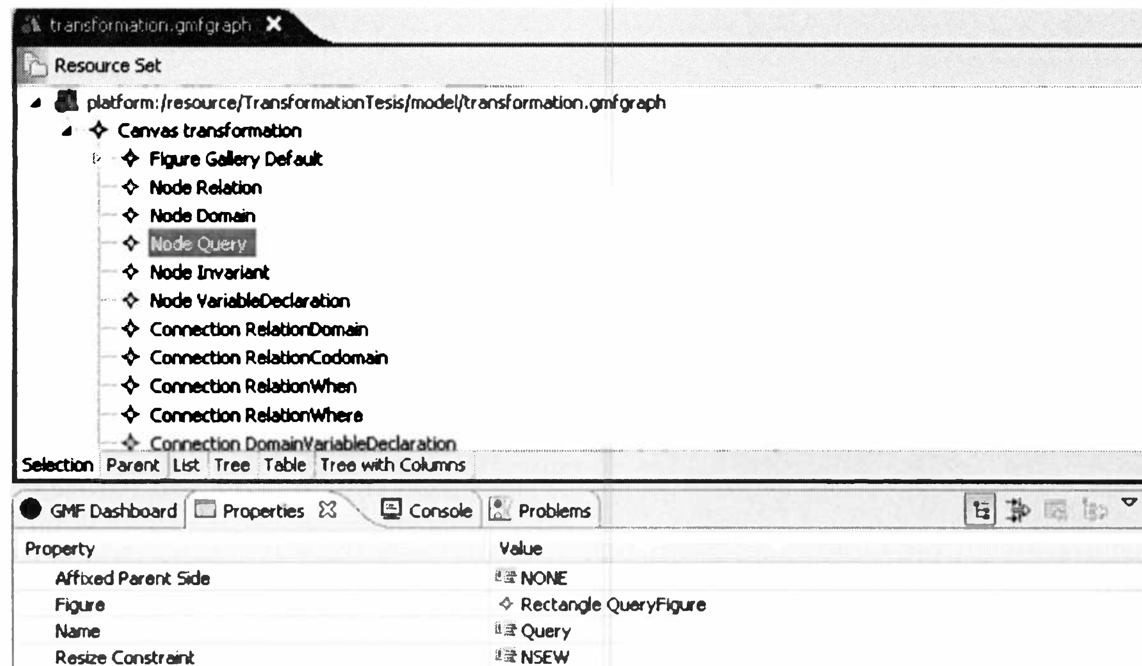


Figura 6.5 – Vista de los nodos del archivo gmfgraph

6.5 – Creación del gmfmap

El archivo gmfmap (ver figura 6.6) es el encargado de relacionar los tres archivos definidos anteriormente: el archivo del modelo (ecore), el archivo de definición de herramientas (gmftool) y el archivo de definición gráfica (gmfgraph).
Para cada nodo, primero, se define a que elemento de la transformación va a pertenecer (Top Node Reference), luego se especifica que nodo del archivo de definición gráfica es y a que herramienta del menú se va a relacionar (Node Mapping), y por ultimo se le asignan las propiedades a ser mostradas con un mapeo a un '*label diagram*' del archivo de definición gráfica.
También existen los mapeos de conexiones llamados *Link Mapping*, los cuales hacen referencias a las relaciones del archivo de definición gráfica que corresponden y a que herramienta del menú va a pertenecer. Una vez que se definieron correctamente todos los nodos y links se pasa a la creación del último archivo de configuración.

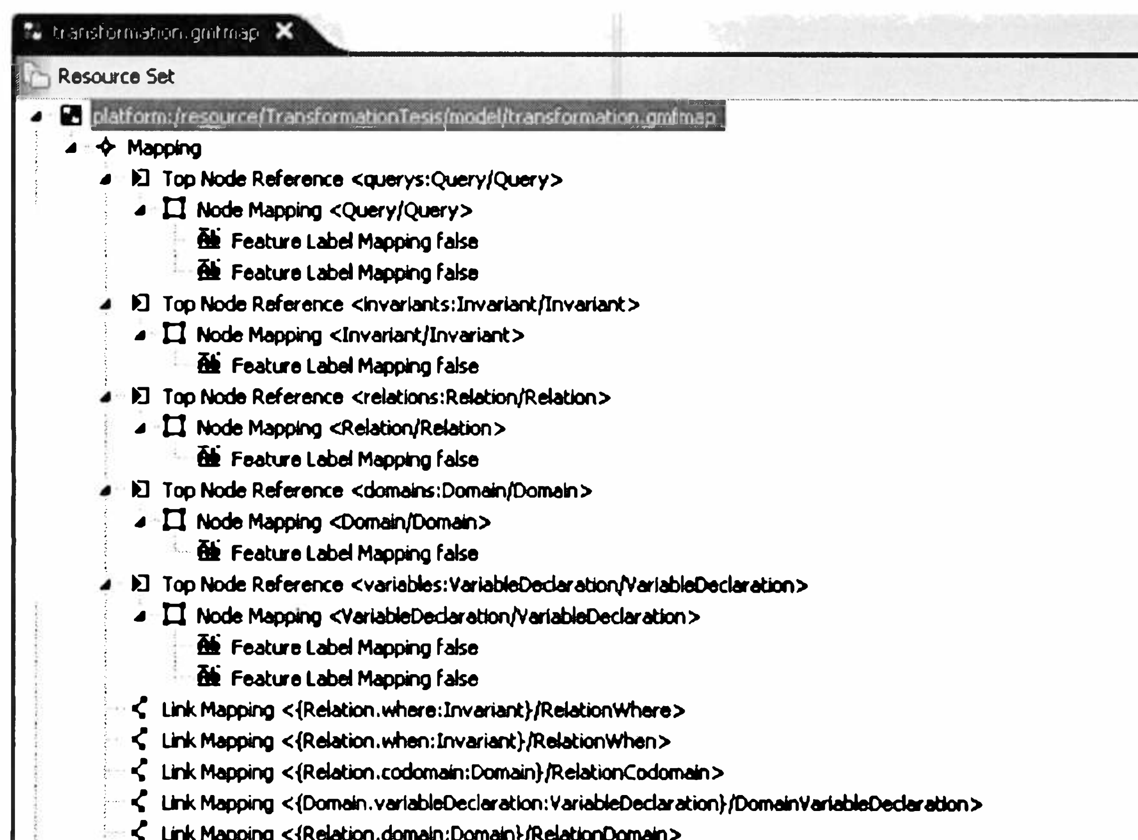


Figura 6.6 – Vista del archivo gmfmap

6.6 - Creación del gmfgen

Una vez que se definieron los elementos gráficos y los mapeos se puede generar el código necesario para crear el editor gráfico. Para poder llevar a cabo esto se debe crear primero el modelo generador (gmfgen) y configurar las propiedades para la generación de código (ver figura 6.7). Una vez que se

configuran todas las propiedades deseadas (por ejemplo nombre de paquetes para los comandos, vistas, etc.) se presiona botón derecho sobre el archivo y se elige la opción *generate diagram code*. Al presionar esa opción se creará un nuevo proyecto con el mismo nombre que el actual (donde se encuentran los archivos) agregándole la palabra "*.diagram*". Este proyecto contendrá todo el código necesario visualizar el editor grafico sobre un archivo "*.transformation_diagram*".

En nuestro caso realizamos modificaciones sobre algunas clases generadas, los cambios fueron los siguientes:

- Se modificó la clase interna **RelationCodomainFigure** (que se encuentra dentro de la clase *RelationCodomainEditPart*), a la que se le agregó un método "addLabel", el cual agrega el label "codomain" para que sea mostrado por encima de la relación *codomain* en el editor gráfico.

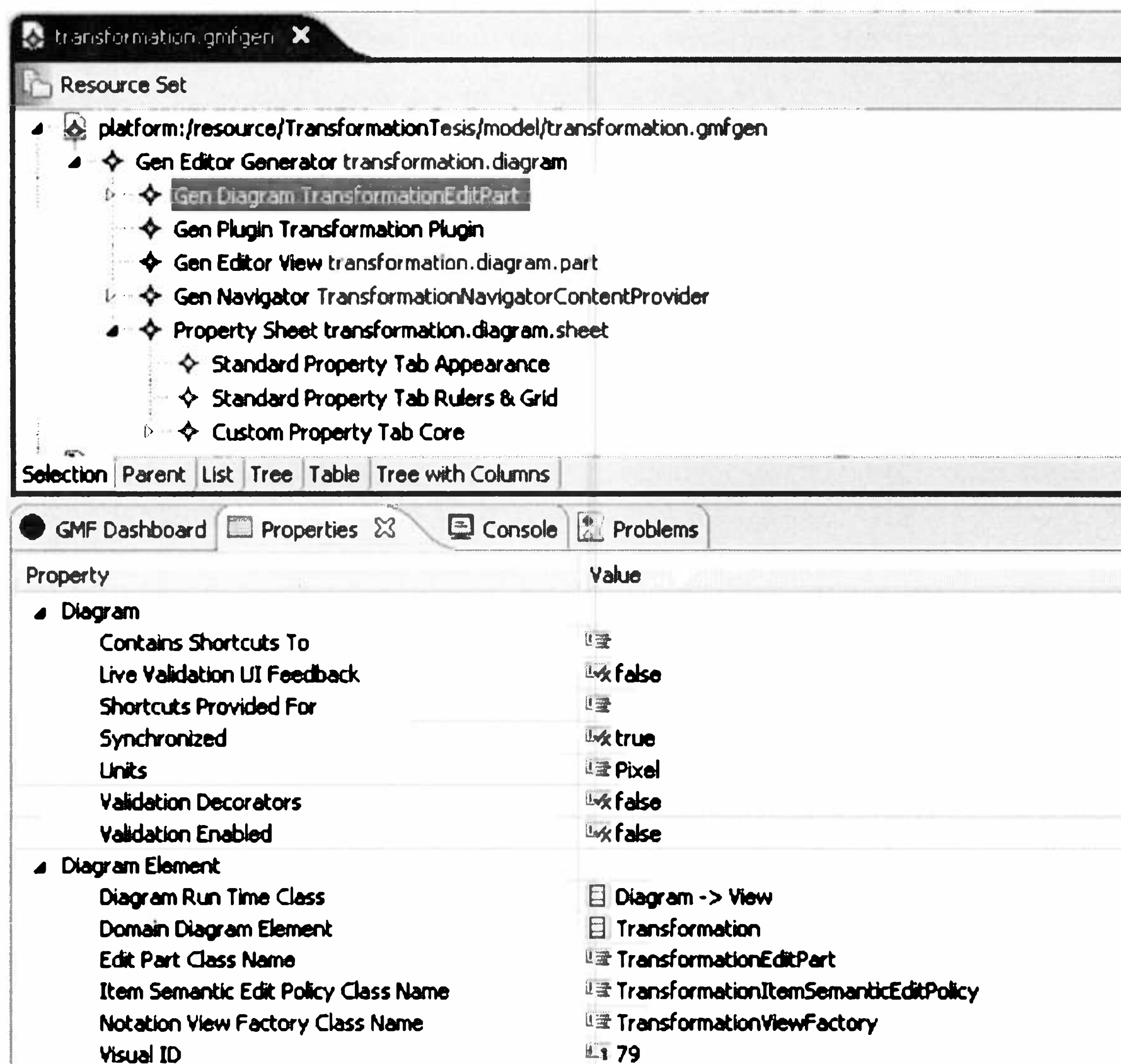


Figura 6.7 – Vista del archivo gmfgen con sus propiedades

- Se modificó la clase interna **RelationDomainFigure** (que se encuentra dentro de la clase RelationDomainEditPart), a la que se le agregó un

método "addLabel", el cual agrega el label "domain" para que sea mostrado por encima de la relación *domain* en el editor gráfico.

- Se modificó la clase interna **RelationWhenFigure** (que se encuentra dentro de la clase RelationWhenEditPart), a la que se le agregó un método "addLabel", el cual agrega el label "when" para que sea mostrado por encima de la relación *when* en el editor gráfico.

- Se modificó la clase interna **RelationWhereFigure** (que se encuentra dentro de la clase RelationWhereEditPart), a la que se le agregó un método "addLabel", el cual agrega el label "where" para que sea mostrado por encima de la relación *where* en el editor gráfico.

  En todos estos casos también se agrego el método "refreshLine", el cual se encarga de centrar el label agregado en la relación y es llamado cuando se mueven los gráficos de lugar.

6.7 – Utilización del Plugin GMF

Para la creación de los archivos nombrados anteriormente como para las derivaciones de código, se utilizó el plugin GMF. Una vez que los archivos era creados se iban configurando para que sean adaptados a las necesidades del plugin de transformaciones.
Cuando se ya estaban creados los archivos *.ecore*, *.gmfgraph* y *.gmftool* se combinaron para la creación del archivo *.gmfmap* (como muestra la figura 6.8). Cuando se finalizó la creación de dicho archivo se procedió a la transformación del archivo *.gmfgen*.
Una vez creado el archivo nombrado anteriormente se ejecuto la acción “Generate Diagram Editor”, lo cual se derivó en la creación del proyecto contenedor del plugin.

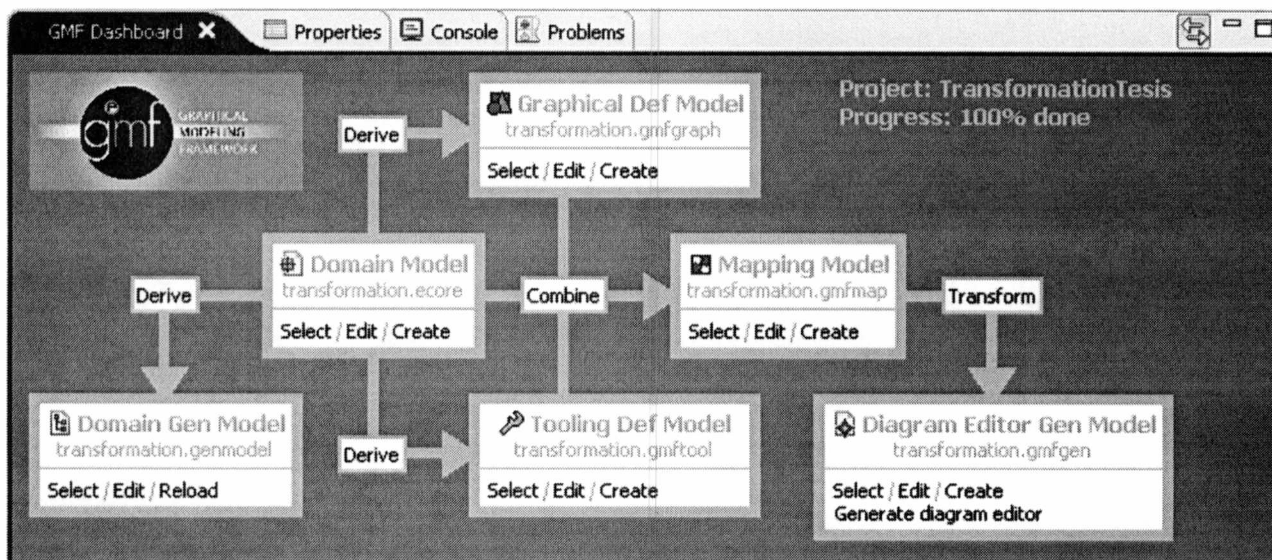


Figura 6.8 – Vista del GMF Dashboard, del plugin GMF.

6.8 - Implementación del cargador de metamodelos

Para poder realizar la carga de los metamodelos se definió un plugin separado al editor; la comunicación entre ellos es mediante una clase que cumple con el patrón Singleton.
Para la creación del plugin para cargar los metamodelos se definieron las siguientes clases:

- *LoaderMetamodelDialog*, es la clase que representa la ventana que se va a abrir una vez que se presiona botón derecho sobre el archivo con extensión *.transformation_diagram* y se selecciona la opción "Cargar Metamodelos". En esta clase se definieron los botones, los inputs, etc. También posee los path que hacen referencia a los metamodelos de entrada y de salida.

- *EcoreLoader*, esta clase tiene doble funcionalidad; la primera es que dado un path de un archivo ecore, retorna todos los nombres de los eObjects de ese ecore, y como segunda funcionalidad, que puede almacenar una lista de nombres de eObject para que sea compartida entre el plugin encargado de leer los metamodelos y el editor gráfico.

- *LoaderMetamodelOpenDialogAction*, es la clase que toma el control cuando se presiona la opción de cargar los metamodelos. Es la encargada de abrir la ventana (clase LoaderMetamodelDialog) y cuando se cierra dicha ventana levanta los eObjects de los dos ecore leídos y se los asigna a la clase EcoreLoader (para que sea leída por el editor gráfico).

CAPÍTULO 7

Implementación del Evaluador de transformaciones

El Evaluador de transformaciones es el que interpreta el modelo creado por el editor de transformaciones. El modelo creado con el editor debe ser instanciado con dos modelos tipados: uno de entrada y otro de salida. Luego esto es enviado al Evaluador que determina si dicha entrada es verdadera o no ejecutando las expresiones OCL descriptas en cada una de las relaciones de la transformación.

7.1 - Diseño del Evaluador

Para la evaluación de una transformación se diseñó un esquema arquitectural basado en dos grandes fases o etapas: la primera es la fase de *construcción de una transformación*. En esta etapa entra en juego el diseño de una transformación; nuestro modelo. La segunda fase es la de la *ejecución de la evaluación*, en la que se utiliza un contexto de evaluación que contiene los elementos necesarios para evaluar una transformación.

7.2 - Construcción de una transformación

La figura 7.1 muestra el flujo en la construcción de una transformación. Una transformación se construye a partir de un archivo para el intercambio de metadatos (XMI) que es la salida de nuestro editor de transformaciones. Este archivo XMI es levantado en memoria, por el objeto **Loader**, como un grafo de objetos que luego es enviado al objeto **TransformationFactory,** en forma de lista, para que éste lo procese y retorne el objeto transformación correspondiente.

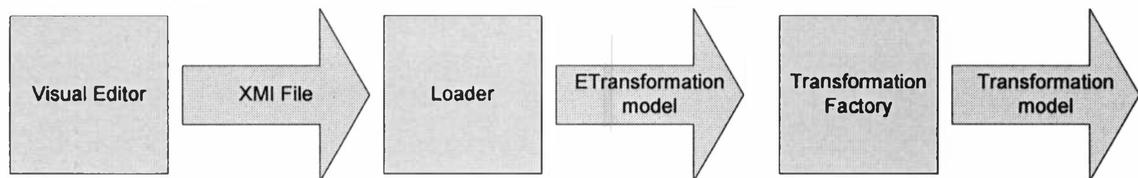


Figura 7.1 - Flujo en la construcción de una transformación

El Loader se encarga de recibir el XMI resultado del editor visual de transformaciones y parsearlo retornando un grafo de objetos o un listado de todos los objetos descriptos en tal archivo. Estos objetos implementan la interfaz **EObject[1]**, que es la interfaz de todos los objetos modelados. Entre el modelo de componentes del editor gráfico (ver creación del editor) y el modelo del evaluador hay una relación prácticamente uno a uno.

Modelo del Editor	Modelo del Evaluador	Responsabilidades
ETransformation	Transformation	Contiene un nombre y una colección de Relation que la definen.
ERelation	Relation	Representa una relación de una transformación que tiene un domain y un codomain (ambos objetos Domain) con sus respectivas declaraciones de variables (objetos VariableDeclaration), y dos predicados (objetos Predicate): when y where que contienen, a su vez, una expresión OCL que debe evaluarse.
EDomain	Domain	Representa un dominio de una relación que contiene una declaración de una variable. Un dominio tiene asociada una lista de clasificadores pertenecientes a un TypedModel. Existen dos dominios en una relación: domain y codomain. El primero contiene los clasificadores del metamodelo de entrada y el segundo los clasificadores del metamodelo de salida.
EInvariant	Predicate	Representa al predicado OCL que debe evaluarse. Un predicado puede ser **when**, que se aplica a una variable del dominio. **Where**, que se aplica a una variable del dominio y otra del codominio. Para poder evaluarse where, la previa evaluación de when debió haber sido verdadera.
EVariableDeclaration	VariableDeclaration	Representa la declaración de una

		variable. Contiene su nombre y su tipo (EClassifier). El tipo de una variable debe estar declarado en la lista de clasificadores del dominio al que pertenece. Las variables declaradas son utilizadas para las evaluaciones de los predicados definidos en una relación.

7.3 - Construcción del contexto de evaluación

El contexto de evaluación es un objeto **TransformationExecutionContext** que se asocia a una transformación y se obtiene pidiéndoselo al objeto **TransformationEnvironment**. Este último contiene una lista de todos los contextos asociados a todas las transformaciones.

Un contexto posee la información de los metamodelos utilizados para la creación y la evaluación de una transformación. Es un básicamente un contenedor de modelos como muestra la figura 7.2. Para poder evaluar es necesario tener los EObjects pertenecientes a las instancias de los metamodelos involucrados en una transformación. Éstos son tres: una instancia de ETransformation, una instancia de EDomain y una de ECodomain.

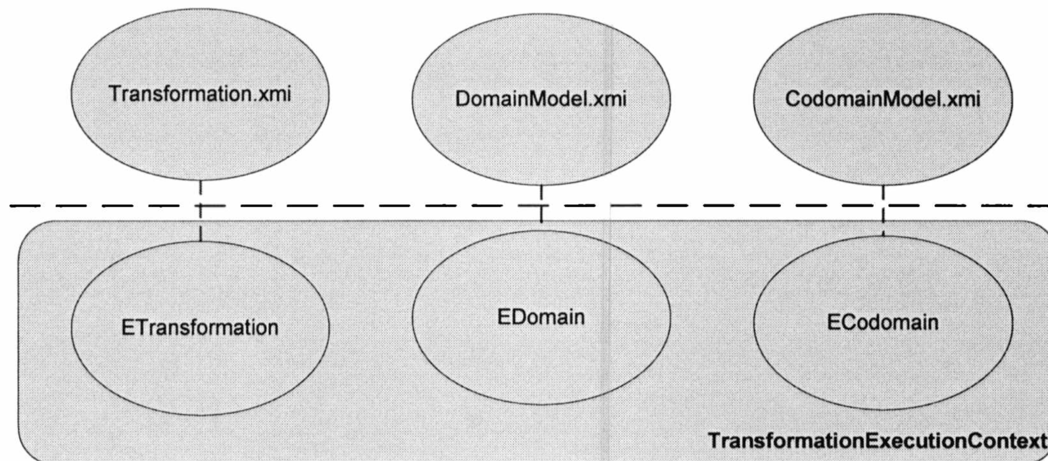


Figura 7.2 – TransformationExecutionContext como contenedor de objetos Ecore

Ejemplo instancia de ETransformation	Ejemplos instancia de EDomain	Ejemplos de instancia de ECodomain
ETransformation (name = UML2Rel)	EClass (name = Persona, persistent = true)	ETable (name = Persona)

ERelation (name = Class2Rel)	EAttribute (name = edad, type= int, multivalued = true)	EColumn (name = edad, type = int)

El contexto es referenciado en el proceso de evaluación cuando es necesario utilizar instancias a evaluar. Por ejemplo en la evaluación de Queries o en la evaluación de los predicados When y Where.

7.4 - Creación de un ambiente de evaluación de OCL

Este ambiente es necesario para poder evaluar cualquier expresión OCL. Al ambiente hay que enviarle un EClassifier que corresponde al contexto de la variable, una lista de variables, una lista de posibles valores para esas variables y la expresión OCL. El ambiente de ejecución OCL es creado pidiéndole una instancia al objeto EcoreEnvironmentFactory, esto es porque se va a evaluar OCL sobre instancias Ecore. Primero se le configura el contexto al ambiente. El contexto de la variable es el clasificador (EClassifer) del objeto sobre el que se va a evaluar la expresión OCL. Luego se agregan al contexto de ejecución del ambiente la lista de variables correspondientes a las variables referenciadas en la expresión. Cada variable debe tener el nombre y el tipo. Una vez cargadas las variables se crea una consulta OCL enviándole al ambiente la expresión OCL. Para poder ejecutar la consulta es necesario hacer el *bind* entre las variables declaradas y los valores que se le asignarán. Luego de realizar esta ligadura se ejecuta la consulta retornando un resultado. El resultado de cada evaluación es un objeto **OCLEvalResult** que tiene las variables que fueron utilizadas para evaluar y el resultado de la ejecución de la consulta OCL. El diagrama de la figura 7.3 representa los flujos aquí explicados:

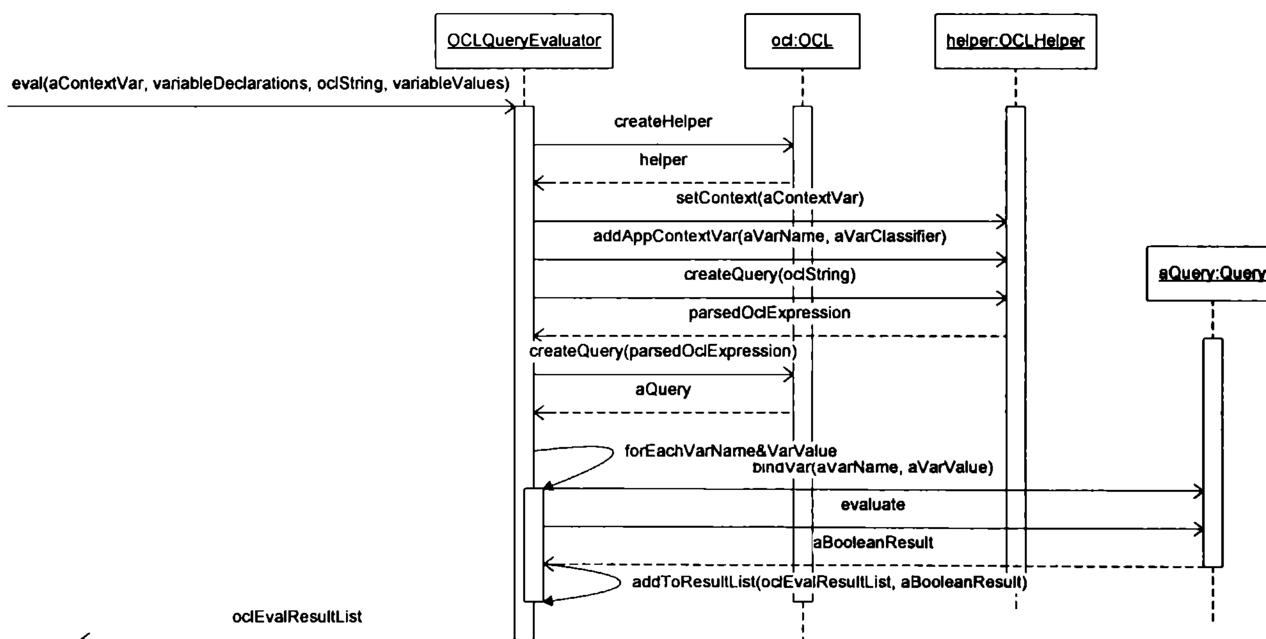


Figura 7.3 – Diagrama de Secuencia de la evaluación de OCL.

7.5 - El proceso de evaluación

Evaluar una transformación significa evaluar cada una de sus relaciones. Evaluar una relación significa evaluar primero el predicado when y luego, si esta evaluación es verdadera, evaluar el predicado where.

Evaluar el predicado when consiste en crear un ambiente de evaluación OCL al que se le envía, como contexto de variable, el clasificador de la variable declarada en el Dominio de la relación. Esto es porque el predicado when siempre hace referencia a una variable del dominio de una relación. Como segundo parámetro se le envía la declaración de variable del dominio. Los posibles valores para esta variable son todos aquellos EObject, que tienen el mismo tipo que la declaración de variable, que pertenezcan al modelo de entrada. Esta lista se le pide al contexto de ejecución (*TransformationExecutionContext*).

Por último al proceso que crea el ambiente de evaluación OCL se le envía la expresión OCL que tiene el predicado when de la relación. En la figura 7.4 se puede observar esta primera etapa de la evaluación de una relación.

El resultado de esta evaluación retorna una lista de resultados (objetos *OCLEvalResult*) que es filtrada para generar parte de la entrada para la próxima evaluación: la del predicado where de la relación.

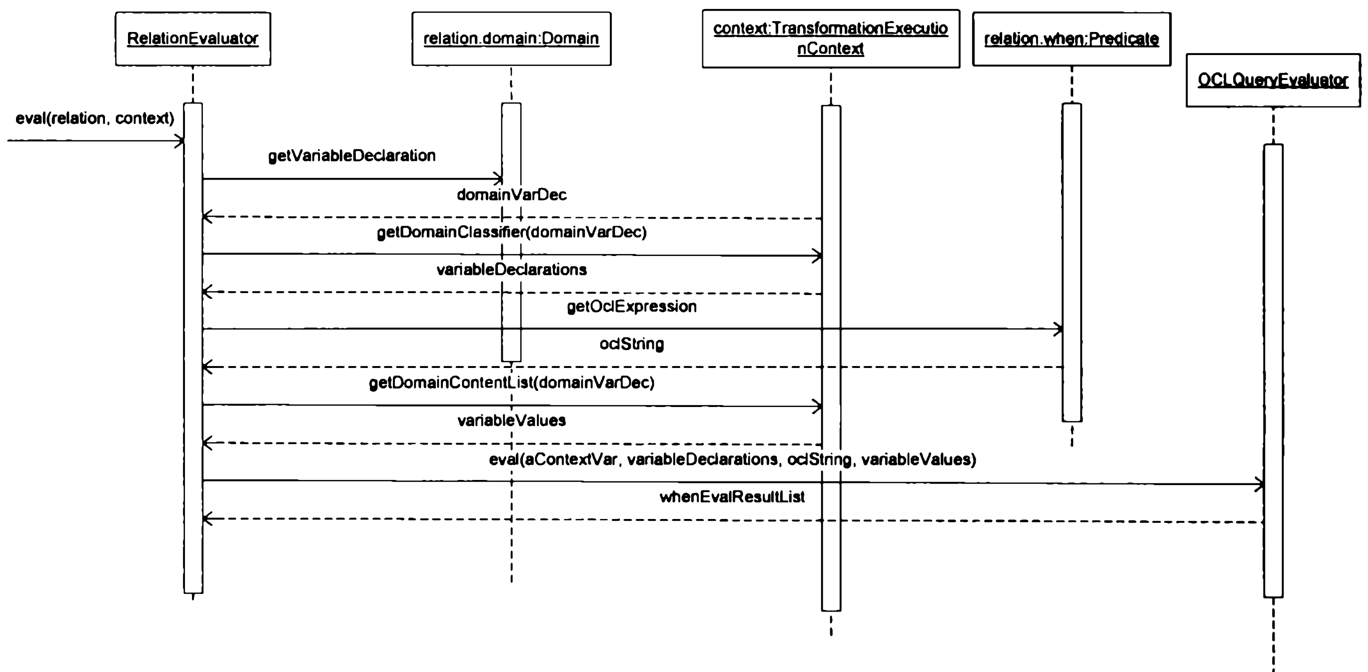


Figura 7.4 - Diagrama de Secuencia de la evaluación del predicado "when" de una relación.

Para la segunda evaluación (ver figura 7.5) se debe crear otro ambiente de evaluación OCL con otros parámetros. El contexto de variable se mantiene, es decir se utiliza el tipo de la variable declarada en el dominio. También se le envían las declaraciones de las variables del dominio y del codominio.

Los posibles valores para la variable del dominio son todos aquellos que en la evaluación del when dieron verdadero; es por este motivo que se filtra la lista de resultados obtenida en la evaluación del predicado when.
Los posibles valores para la ligar a la variable del codominio son los EObject que pertenecen al modelo de salida y son del mismo tipo que dicha variable posee. Al igual que para el caso de la evaluación de when, esta lista se le pide al contexto de ejecución de la transformación.
La expresión OCL que se le envía al nuevo ambiente de evaluación OCL es la que contiene el predicado where de la relación que está siendo evaluada. Con todos estos parámetros cargados en el ambiente se evalúa el predicado where retornando una lista con resultados.
Los resultados obtenidos en ambas evaluaciones se agregan a un resultado más general: un objeto EvalResult. Éste representa el resultado de evaluar una relación, y es responsable de determinar si la evaluación tiene resultado verdadero o falso.

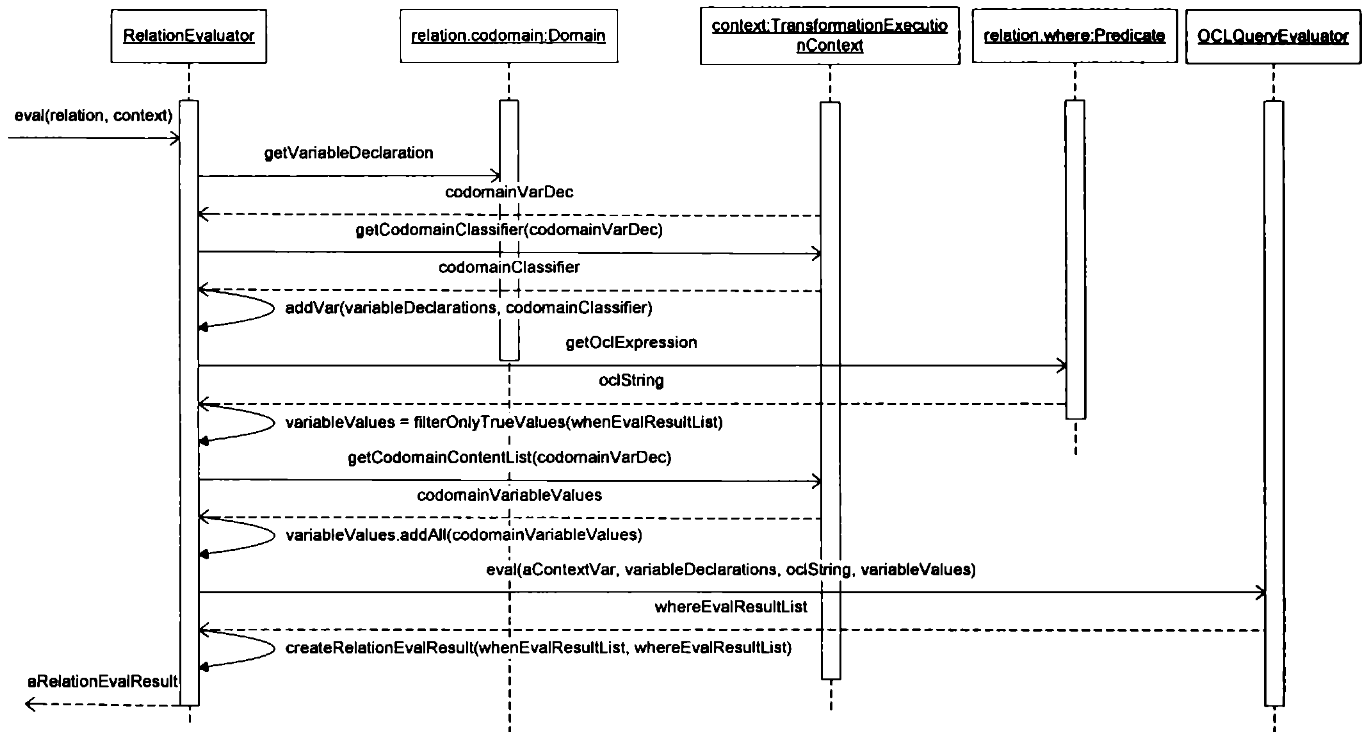


Figura 7.5 - Diagrama de Secuencia de la evaluación del predicado "where" de una relación

7.6 - Interpretación de los resultados

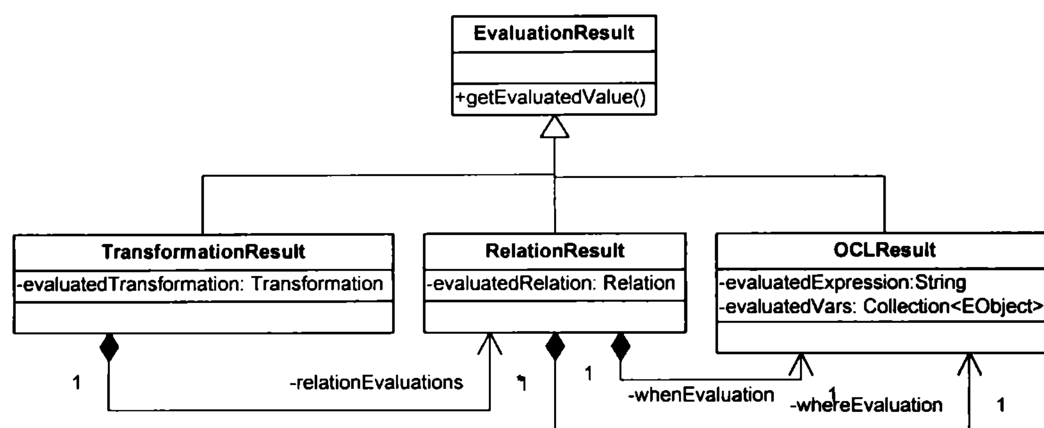


Figura 7.6 – Resultados modelados: la jerarquía EvaluationResult

El resultado de una evaluación de transformación (figura 7.6) se modeló pensando en delegar en él la responsabilidad de saber si es verdadero o falso. Del proceso de evaluación se desprenden tres tipos de resultados posibles, según las "etapas" de éste: El resultado de una evaluación OCL, el resultado de una evaluación de relación y el resultado de la transformación en sí.
El resultado de una evaluación OCL contiene la expresión OCL evaluada, los nombres de las variables utilizadas en la expresión, los valores que fueron utilizados para ligar a dichas variables en la evaluación. Un resultado OCL es verdadero cuando la evaluación de su expresión OCL que resulta de ligar las variables allí utilizadas con los valores que él posee, es verdadera.
El resultado de una relación contiene una lista de resultados OCL para la evaluación del predicado when y otra lista para la evaluación del predicado where. Un resultado relación es verdadero cuando todos los tiene resultados when y where y todos son verdaderos.
Por último el resultado de una transformación contiene una lista de resultados relación. Este resultado es verdadero cuando todos sus resultados relación son verdaderos.

7.7 - Referencias a otras relaciones. Las relaciones como funciones

El evaluador soporta la modificación de OCL para agregar como operaciones el llamado a otras relaciones declaradas en la definición de la transformación. La referencia se realiza nombrándola y pasándole dos parámetros, que son las variables (la del dominio y la del codominio) que deberán ser ligadas en tiempo de ejecución para que el llamado funcione correctamente.
Para brindar este soporte se pensó, en una primera instancia, en modificar el parseador de OCL provisto en el plugin de OCL. Para ello se investigó el funcionamiento del mismo y el desglose de cada expresión OCL que ingresa a parsearse. La primera idea era la de parsear la expresión recibida y convertirla en una EOperation (que son las operaciones modeladas por el plugin EMF).
Esta solución quedó descartada por la elevada complejidad al no poder encontrar bibliografía respecto al tema.

Luego se pensó en atacar el problema a un nivel más abstracto que consiste en reemplazar el llamado a la relación por la conjunción de los predicados when y where de la relación invocada. Este reemplazo se debe realizar antes que la expresión OCL que hace la invocación se evalúe.
Si se respeta el patrón de nombres al llamar a una relación, puede identificarse definiendo expresiones regulares para poder encontrarlo dentro de la descripción de la expresión OCL. Esto es posible porque el editor de transformaciones permite ingresar un String, o cadena, para describir dicha expresión. Este dato es parseado utilizando los patrones de las expresiones regulares. Si encuentra un posible llamado, se chequea la transformación iterando por cada una de sus relaciones y comparando el nombre de cada una de estas con el nombre encontrado. Si en esta comparación las cadenas coinciden se reemplaza sobre la expresión OCL original la subcadena donde está el llamado a la relación por la conjunción de los predicados when y where de la relación encontrada. Haciendo este reemplazo puede surgir que los nombres de las variables declaradas en la relación encontrada se solape con algún nombre de la expresión a modificada, haciendo que los resultados de la evaluación no sean los esperados. A este problema se lo solucionó reemplazando en la expresión que representa la conjunción de los predicados, los nombres de las variables allí referenciadas por los nombres de los parámetros utilizados en la invocación de la relación.

7.8 – Asunciones

Hay algunas asunciones que debimos tomar para poder implementar SQVT:
Las relaciones tienen en su When expression la referencia a una variable del dominio.
En su Where expression se referencian dos variables, que tienen un orden implícito. La primera pertenece al dominio y la segunda al codominio. Se pueden hacer referencias a relaciones, invocándolas como operaciones OCL con 2 parámetros. El primero es ligado a la variable del dominio de la relación, y el segundo parámetro es ligado a la variable declarada perteneciente al codominio. La relación nombrada es reemplazada por la conjunción de las expresiones de when y where.

CAPÍTULO 8
Conclusiones

En este capítulo se van a desarrollar las conclusiones sobre el trabajo de tesis, como así también los trabajos que quedaron pendientes para su posterior investigación y desarrollo.

8.1 – Conclusiones

A lo largo de esta década la *Ingeniería de Software Conducida por Modelos* (MDE) se ha convertido en un nuevo paradigma de software que propone mejorar la construcción de software a través de un proceso guiado por modelos y soportado por potentes herramientas que generan código a partir de modelos. MDE promete una mejora de la productividad y de la calidad del software generado debido a que se 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.
Sobre la investigación que se realizó sobre el trabajo efectuado por Giandini referente a la formalización de un Lenguaje de Transformación de Modelos se desarrolló un plugin para la plataforma Eclipse, en el cual se puede diseñar gráficamente una transformación con sus correspondientes metamodelos de entrada y de salida. Dicho desarrollo se dividió en dos partes, por un lado el desarrollo del editor gráfico de transformaciones, y por el otro el evaluador OCL para dichas transformaciones.
Con respecto al desarrollo del editor, se investigaron con detalle los plugins EMF, GEF y GMF, los cuales permitieron y facilitaron su creación. Uno de los grandes inconvenientes que surgieron fue que el plugin GMF no estaba totalmente desarrollado al momento de necesitarlo, por lo que se tuvo que modificar el código fuente del plugin para que funcione correctamente; más precisamente se tuvieron que extender algunas clases del plugin para proveerlas de mayor funcionalidad.
Aunque se resaltó el inconveniente anterior, cabe destacar que luego de una investigación profunda de los plugins anteriormente nombrados, resultó fácil la creación de editores gráficos con la herramienta GMF.
Otro punto en contra que se encontró fue la incompatibilidad que existe entre las diferentes versiones de los plugins de Eclipse, es decir, si se está trabajando con una versión de un plugin, y se quiere actualizar a una más reciente hay que tener mucho cuidado que ésta nueva versión no genere conflicto con los demás plugins que se están utilizando.
Un punto rescatable es la fácil y rápida conexión que se logra entre los diferentes plugins de Eclipse que corresponden a la misma versión. Fue por esta ventaja que el diseño e implementación del editor de transformaciones pudo ser dividido en dos partes, desarrolladas en forma paralela: por un lado la parte del editor visual y por otra la del evaluador OCL. Cada parte fue desarrollada por uno de los integrantes de este trabajo y luego ensambladas, sin mayores inconvenientes, para terminar en el producto completo.

Al implementar SQVT (la propuesta de Giandini) y no QVT (la propuesta de la OMG) se nos facilitó la tarea pues pudimos utilizar el plugin de OCL, que ya existe para Eclipse. Esto es porque SQVT utiliza mas estándares facilitando no solo su entendimiento sino también su desarrollo. Específicamente, no fue necesario crear parseadores, analizadores léxicos y demás herramientas necesarias para realizar las evaluaciones de las transformaciones.
Creemos enfáticamente que el desarrollo de herramientas visuales, como lo es nuestra implementación, donde los conceptos están representados mediante íconos facilita ampliamente la labor de los programadores. MDA tiende a que la futura tarea del programador sea la de diseñar y el código sea derivable automáticamente. Para que esto sea posible es necesaria la existencia de herramientas como las que nosotros hemos desarrollado.

8.2 – Trabajos Futuros

Aunque el plugin funciona de manera correcta, quedan algunas posibles extensiones para realizar a futuro, las cuales son:

- Generar modelo de salida: El plugin solo soporta el chequeo de los dominios. Es decir que las relaciones no pueden ser forzadas como propone QVT (atributos de Domain, isCheckeable e isEnforceable) o SQVT que propone modificar o generar un modelo de salida. Con nuestro plugin sólo se pueden validar si dos modelos (de entrada y salida) satisfacen en forma consistente la definición de la transformación.

- Relaciones de alto nivel: El plugin no hace distinción entre las categorías de las relaciones. Si bien en el editor se puede configurar si una relación es de alto nivel (atributo isTopLevel de una Relation), a la hora de editar dicha relación, esta propiedad no es utilizada ya que el plugin no deriva código. Queda para trabajo futuro, realizar el chequeo y que la ejecución de la transformación tenga arranque en las relaciones que tengan en valor verdadero el atributo *isTopLevel*.

- Evaluación de los Querys: Los componentes querys pueden ser creados en el editor (también se puede escribir la expresión OCL dentro de ellos). No está implementado el evaluador de los mismos.

- Mejoras de visualización: Se pueden optimizar las figuras del editor, como así también la visualización de sus propiedades (por ej. Invariant).

- Integración con e-Platero: Aunque la integración entre plugins de Eclipse es de forma independiente, es engorroso trabajar con distintas versiones entre los plugins. Es decir que un plugin está pensado para trabajar con una versión determinada de otro plugin. Esta desventaja hace que la forma de acoplar nuestro plugin a e-Platero no sea directa, ya que ambos utilizan los mismos plugins pero con diferente versión. Por ésto queda como trabajo futuro investigar la integración con e-Platero.

Referencias Bibliográficas

1- A Definition of MDA - http://www.omg.org/docs/ormsc/04-08-02.pdf.

2- MDA Guide, v1. 0. 1, omg/03-06-01, June 2003. http://www.omg.org.

3- Fabricia R. Frantz,Ana Llona - Estudio de QVT con Smart-QVT y MOMENT http://www.lsi.us.es/docencia/get.php?id=2289

4- Object Management Group (OMG) http://www.omg.org

5- Meta-Object Facility (MOF) – http://www.service-architcture.com/web-services/articles/meta-object_facility_mof.html

6- Carlos Diego García - Implementación de técnicas de evaluación y refinamiento para OCL 2.0 sobre múltiples lenguajes basados en MOF – Tesis de Magíster – Facultad de Informática - Universidad Nacional de La Plata – Julio, 2006.

7- 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). Lima, Perú, Febrero, 2007.

8- Frédéric Fondement, Raul Silaghi - Defining Model Driven Engineering Processes, Software Engineering Laboratory Swiss Federal Institute of Technology in Lausanne CH-1015 Lausanne EPFL, Switzerland - WiSME@UML 2004 Monday October 11th 2004 Lisbon, Portugal - 3rd Workshop in Software Model Engineering http://www.metamodel.com/wisme-2004/ - Workshop WS5 at the 7th International Conference on the UML October 11-15 2004, Lisbon, Portugal http://www.umlconference.org/

9- Jean-Marie Favre - Towards a Basic Theory to Model Model Driven Engineering - http://www-adele.imag.fr/users/Jean-Marie.Favre/papers/TowardsABasicTheoryToModelModelDrivenEngineering.pdf - 2004

10- Valerio Adrián Anacleto - Introducción a UML 2.0 - La evolución de la programación hacia la ejecución y validación automática de modelos

11-The Unified Modeling Language Infrastructure version 2.0, OMG Final Adopted Specification. Marzo,2005. http://www.omg.org

12-The Unified Modeling Language Superstructure. Version 2.0., OMG Final Adopted Specification. April 2004. http://www.omg.org.

13-Eclipse Platform Technical Overview – http://www.eclipse.org/whitepapers/eclipse-overview.pdf

14-Ed Burnette - Rich Client Tutorial – http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.ui.tutorials.rcp.part1/html/tutorial1.html?view=co

15-Rich Client Platform - http://wiki.eclipse.org/index.php/Rich_Client_Platform

16- Lars Vogel - Eclipse Rich Client Platform (RCP) with Eclipse Europa – Abril, 2008.

17-Jeff McAffer, Jean, Michel Lemieux - Designing, Coding, and Packaging Java™ Applications - Addison Wesley Professional – Octubre, 2005.

18-GefDescription - http://eclipsewiki.editme.com/GefDescription

19-GMF- http://www.eclipse.org/gmf/

20-Eclipse Modeling Framework Project (EMF) - http://www.eclipse.org/modeling/emf/

21-The Graphical Editing Framework (GEF) - http://www.eclipse.org/gef/

22- Eric Clayberg, Dan Rubel - Building Commercial Quality Eclipse Plug-ins - Addison Wesley Professional - Marzo,2006

23- Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, Philippe Vanderheyden - Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework - IBM RedBooks - Febrero 2004.

24-Lidia Fuentes y Antonio Vallecillo - Una Introducción a los Perfiles UML - Depto. de Lenguajes y Ciencias de la Computación - Universidad de Málaga - Campus de Teatinos. E29071- Málaga (SPAIN) - Novática - Abril 2004

25-Jos Warmer, Anneke Kleppe - The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition - Addison Wesley Professional; 2 edition - Agosto, 2003

26-Lars Vogel - Eclipse Modeling Framework (EMF) and Java Emitter Template (JET) - Tutorial - Version 0.4 - Copyright © 2007 Lars Vogel - Agosto, 2007

27- Desarrollo de Software Dirigido por Modelos: teorías, metodologías y herramientas - http://sol.info.unlp.edu.ar/eclipse/

28-Roxana Giandini, Claudia Pons - “Un lenguaje para Transformación de Modelos basado en MOF y OCL”. - Proceedings of the 32th Latin-American Conference on Informatics - CLEI 2006. Santiago, Chile. Agosto, 2006

29-UML 2.0 OCL Specification - http://www.omg.org/docs/ptc/03-10-14.pdf

30-OCL 2.0 Specification – Versión 2.0 - http://www.omg.org/docs/ptc/05-06-06.pdf

31-MOF QVT Final Adopted Specification - OMG Specification - www.omg.org/docs/ptc/05-11-01.pdf

32-e-Platero - http://sol.info.unlp.edu.ar/eclipse.

33-Pérez J., García M. - XMI: XML Metadata Interchange – Facultad de Informática – Universidad Politécnica de Valencia

34-Roxana Silvia Giandini - Un Marco Formal para Transformaciones en la Ingeniería de Software Conducida por Modelos – Tesis Doctorado en Ciencias Informáticas – Facultad de Informática – Universidad Nacional de La Plata – Noviembre, 2007.

35- Pons Claudia, Garcia Diego. "An OCL-based Technique for Specifying and Verifying Refinement - oriented Transformations in MDE". Proceedings MoDELS/UML 2006 "- Model Driven Engineering Languages and Systems, 9th International Conference, Genoa, Italy, October 2006. Lecture Notes in Computer Science (LNCS).

36-Demuth, Birgit and Heinrich Hussmann, and Ansgar Konermann. Generation of an OCL 2.0 Parser. Workshop co-located with MoDELS'05: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Jamaica. October 4, 2005.

10 - ANEXO I
Utilización del Editor Gráfico

En este capítulo se explicará cómo se tiene que utilizar el editor gráfico para realizar una transformación y luego validar el código OCL.

10.1 - Creación de los archivos *transformation* y *transformation_diagram*

En primera instancia se deben crear dos archivos, uno con extensión *transformation* y otro con extensión *transformation_diagram;* el primero es el que contendrá toda la información referida a la transformación (nombre, relaciones, dominios, etc.) y el segundo estará compuesto de la información gráfica de la transformación (ubicación dentro del editor de los elementos, tamaño de los mismos, etc.). Ver figura 10.1:

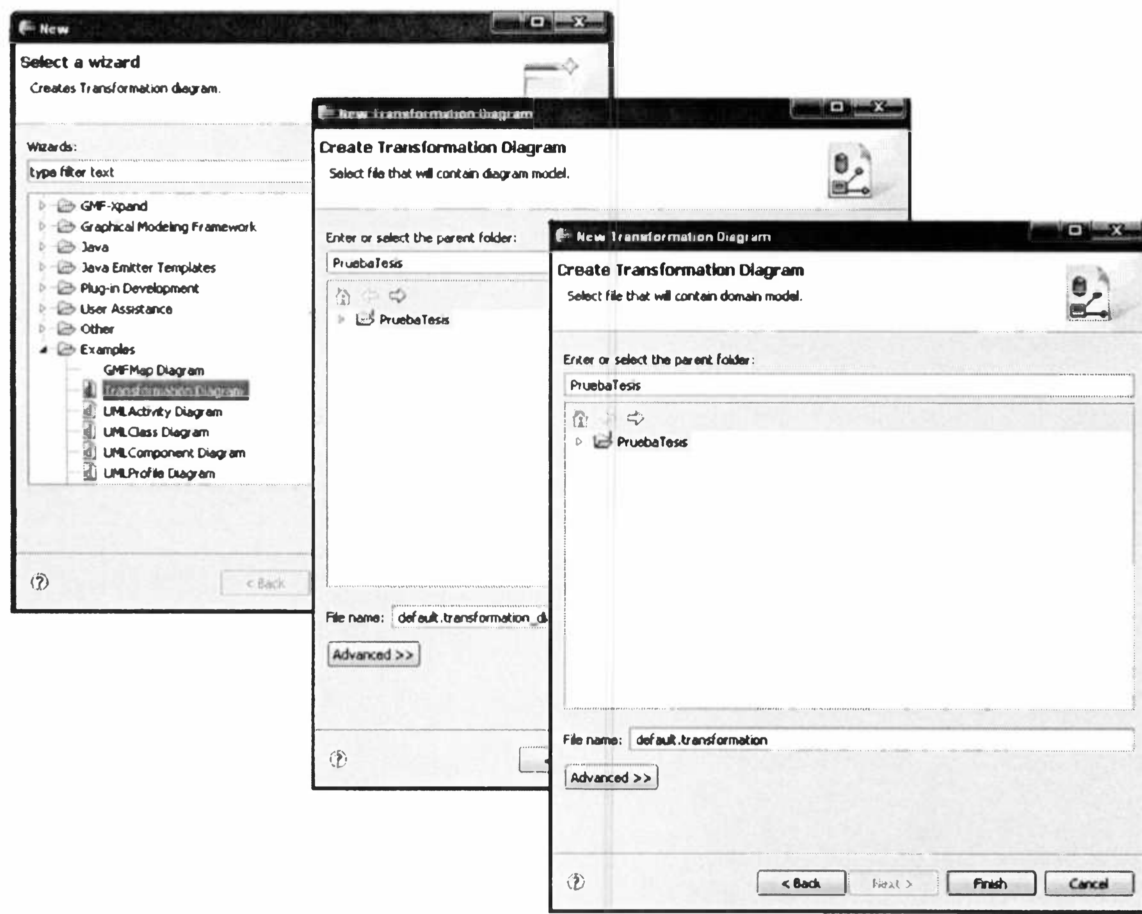


Figura 10.1 – Creación de la transformación

Para la creación de estos dos archivos se debe estar ubicado dentro de un proyecto y ahí presionar en el menú la secuencia File -> New -> Other. Una vez accedido se debe abrir la carpeta examples y en esa carpeta elegir la opción *Transformation Diagram.* Cuando se seleccionó el tipo de archivo a crear se debe presionar siguiente y escribir el nombre del archivo para el diagrama. Luego se presiona siguiente nuevamente y se ingresa el nombre para el

archivo de la transformación. Al terminar con esto se debe presionar sobre el botón *Finish*.

10.2 - Edición de la transformación

Una vez que se crearon los archivos, se debe realizar doble clic sobre el archivo transformation_diagram para comenzar a editar la transformación. El editor posee una barra de herramientas y un visor (que es el sector donde se va a ir graficando la transformación), en la primera se encuentran los íconos para ir agregando las figuras al visor. Los íconos de figuras que se encuentran son *Relation*, *Domain*, *Query*, *Invariant* y *Variable Declaration*; y los íconos de conexiones son *domain*, *codomain*, *when*, *where* y *variableDeclaration*.

10.2.1 – Creación de una figura Relation

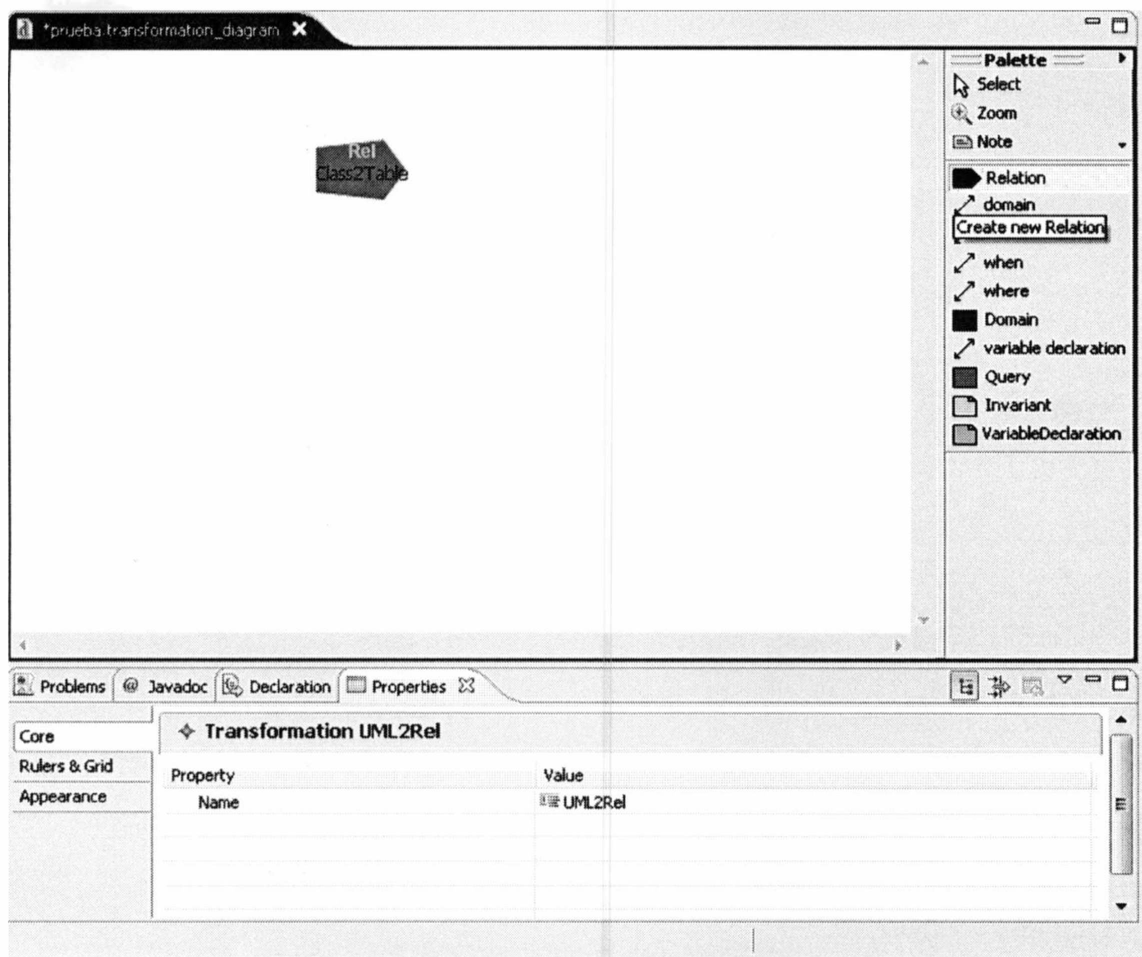
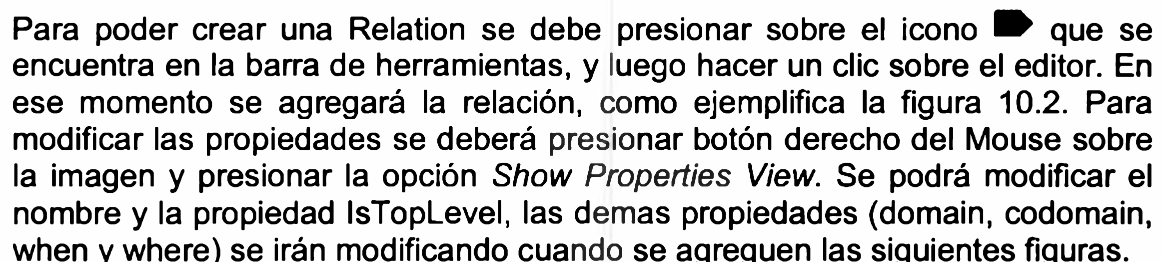


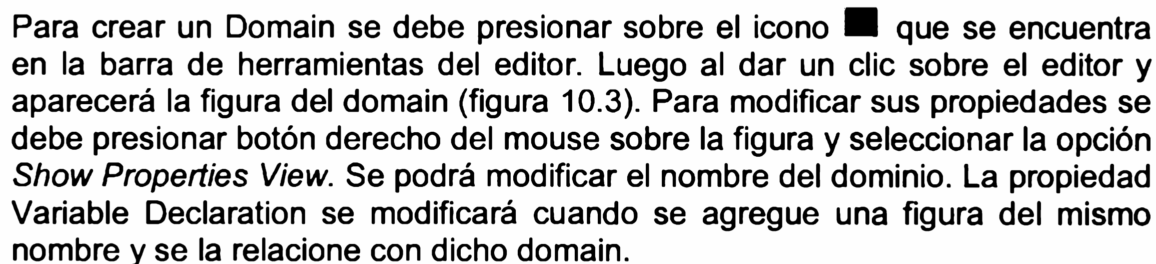
Figura 10.2 – Creación de una figura Relation



Para poder crear una Relation se debe presionar sobre el icono que se encuentra en la barra de herramientas, y luego hacer un clic sobre el editor. En ese momento se agregará la relación, como ejemplifica la figura 10.2. Para modificar las propiedades se deberá presionar botón derecho del Mouse sobre la imagen y presionar la opción *Show Properties View*. Se podrá modificar el nombre y la propiedad IsTopLevel, las demas propiedades (domain, codomain, when y where) se irán modificando cuando se agreguen las siguientes figuras.

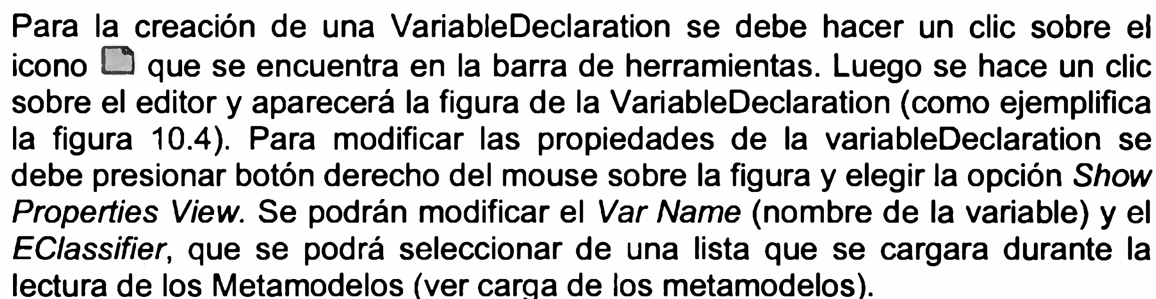
10.2.2 – Creación de una figura Domain

Figura 10.3 - Creación de una figura Domain



Para crear un Domain se debe presionar sobre el icono ■ que se encuentra en la barra de herramientas del editor. Luego al dar un clic sobre el editor y aparecerá la figura del domain (figura 10.3). Para modificar sus propiedades se debe presionar botón derecho del mouse sobre la figura y seleccionar la opción *Show Properties View*. Se podrá modificar el nombre del dominio. La propiedad Variable Declaration se modificará cuando se agregue una figura del mismo nombre y se la relacione con dicho domain.

10.2.3 – Creación de una figura VariableDeclaration



Para la creación de una VariableDeclaration se debe hacer un clic sobre el icono 🗋 que se encuentra en la barra de herramientas. Luego se hace un clic sobre el editor y aparecerá la figura de la VariableDeclaration (como ejemplifica la figura 10.4). Para modificar las propiedades de la variableDeclaration se debe presionar botón derecho del mouse sobre la figura y elegir la opción *Show Properties View*. Se podrán modificar el *Var Name* (nombre de la variable) y el *EClassifier*, que se podrá seleccionar de una lista que se cargara durante la lectura de los Metamodelos (ver carga de los metamodelos).

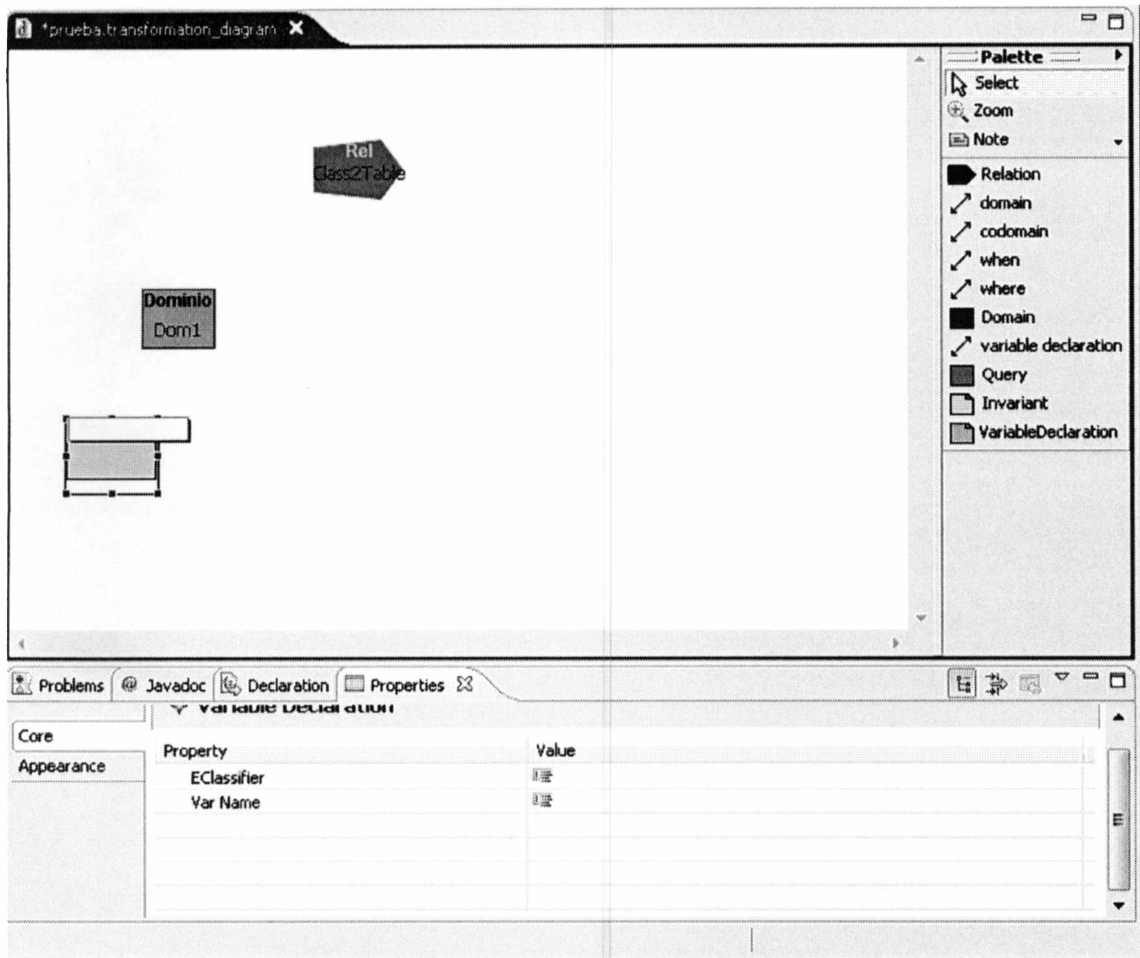


Figura 10.4 - Creación de una figura VariableDeclaration

10.2.4 – Creación de una relación de Dominio

Para unir una *Relation* con un *Domain* a través de un link de *domain* se debe presionar el icono "↗ domain" de la barra de herramientas del editor. Luego se debe hacer un clic sobre la Relation que se quiere unir y se debe arrastrar el mouse, sin soltar el botón, hasta el Domain interesado. Esta acción hará que aparezca un link entre la *Relation* y *Domain* con la palabra "domain" como muestra la figura 10.5. Este link no posee propiedades. Si se quiere verificar dicha composición se pueden visualizar las propiedades de la *Relation* comprendida y en la propiedad *domain* aparecerá el nombre del *Domain(*en nuestro ejemplo "dom1"*).*

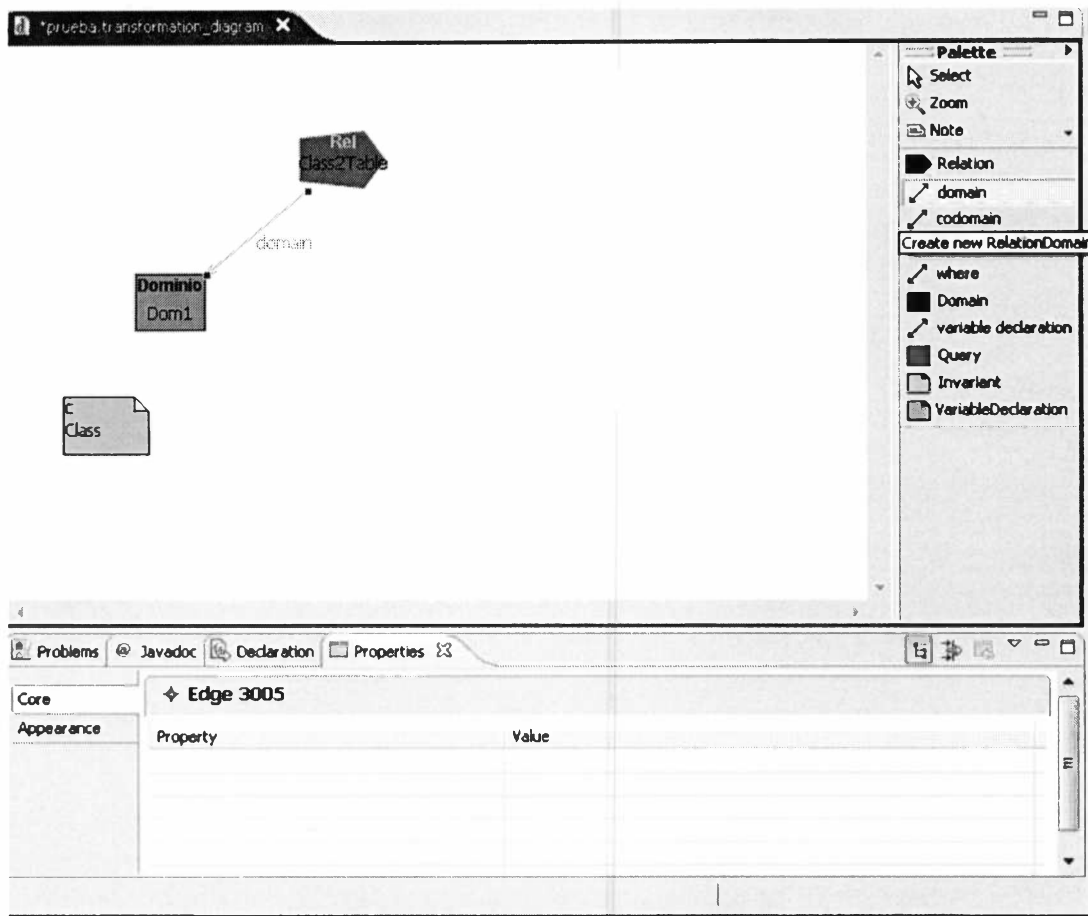


Figura 10.5 - Creación de una relación de dominio

10.2.5 – Creación de una relación de Codominio

Para unir una *Relation* con un *Domain* a través de un link de *codomain* se debe presionar el icono "codomain" de la barra de herramientas del editor. Luego se debe hacer un clic sobre la Relation que se quiere unir y se debe arrastrar el mouse, sin soltar el botón, hasta el Domain interesado. Esta acción hará que aparezca un link entre la *Relation* y *Domain* con la palabra "codomain" como muestra la figura 10.6. Este link no posee propiedades. Si se quiere verificar dicha composición se pueden visualizar las propiedades de la *Relation* comprendida y en la propiedad *codomain* aparecerá el nombre del *Domain*(en nuestro ejemplo "dom2").

Figura 10.6 - Creación de una relación de codominio

10.2.6 – Creación de una relación de Declaración de Variable

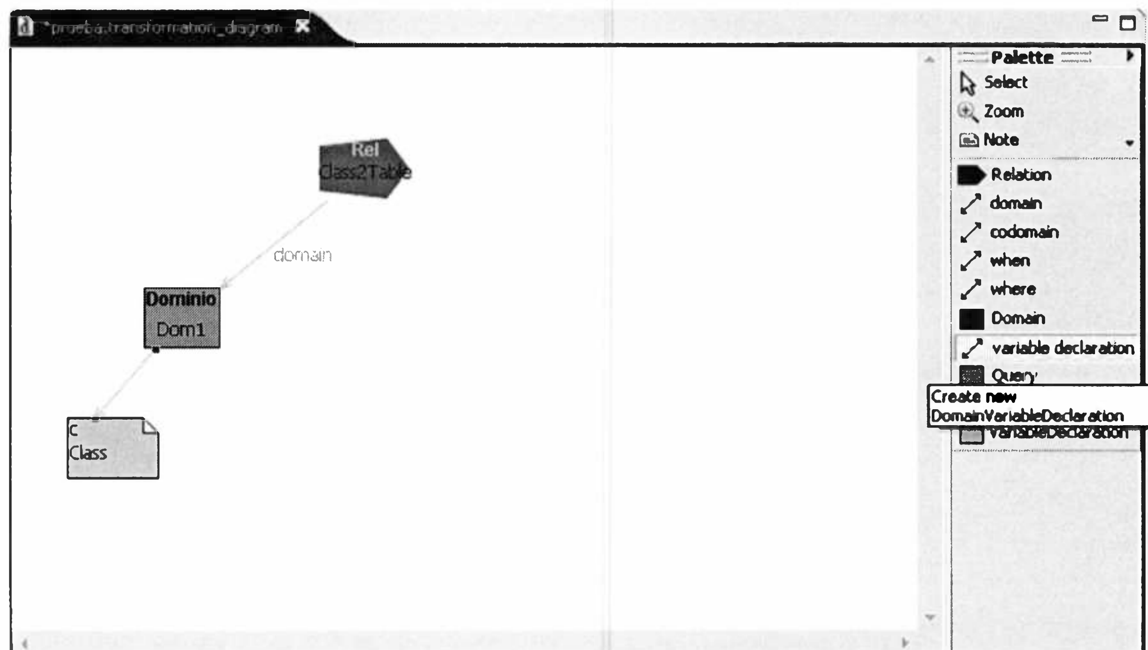
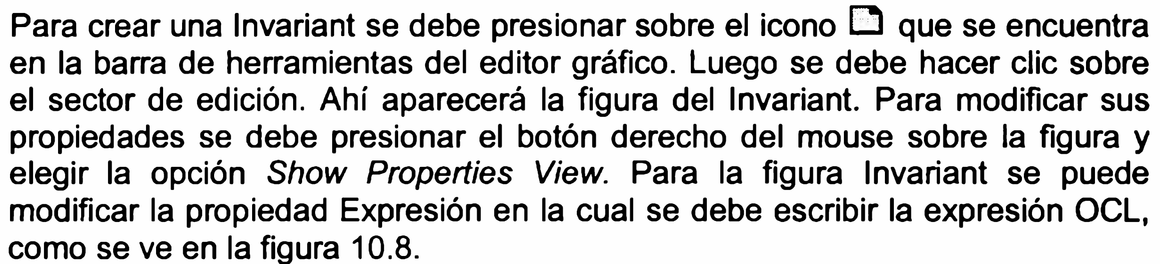


Figura 10.7 - Creación de una relación de Declaración de Variable

Para unir un Domain con una VariableDeclaration se debe presionar en el icono "↗ variable declaration" del menú de herramientas del editor. Luego se debe hacer un clic en el Domain elegido y se debe arrastrar el Mouse sin soltar el botón hasta la VariableDeclaration que se quiere involucrar. Este link no posee ninguna propiedad, pero se puede ver, si la composición quedó correcta, en las propiedades del Domain, la opción Variable Declaration (figura 10.7).

10.2.7 – Creación de una figura Invariant



Para crear una Invariant se debe presionar sobre el icono 🗋 que se encuentra en la barra de herramientas del editor gráfico. Luego se debe hacer clic sobre el sector de edición. Ahí aparecerá la figura del Invariant. Para modificar sus propiedades se debe presionar el botón derecho del mouse sobre la figura y elegir la opción *Show Properties View.* Para la figura Invariant se puede modificar la propiedad Expresión en la cual se debe escribir la expresión OCL, como se ve en la figura 10.8.

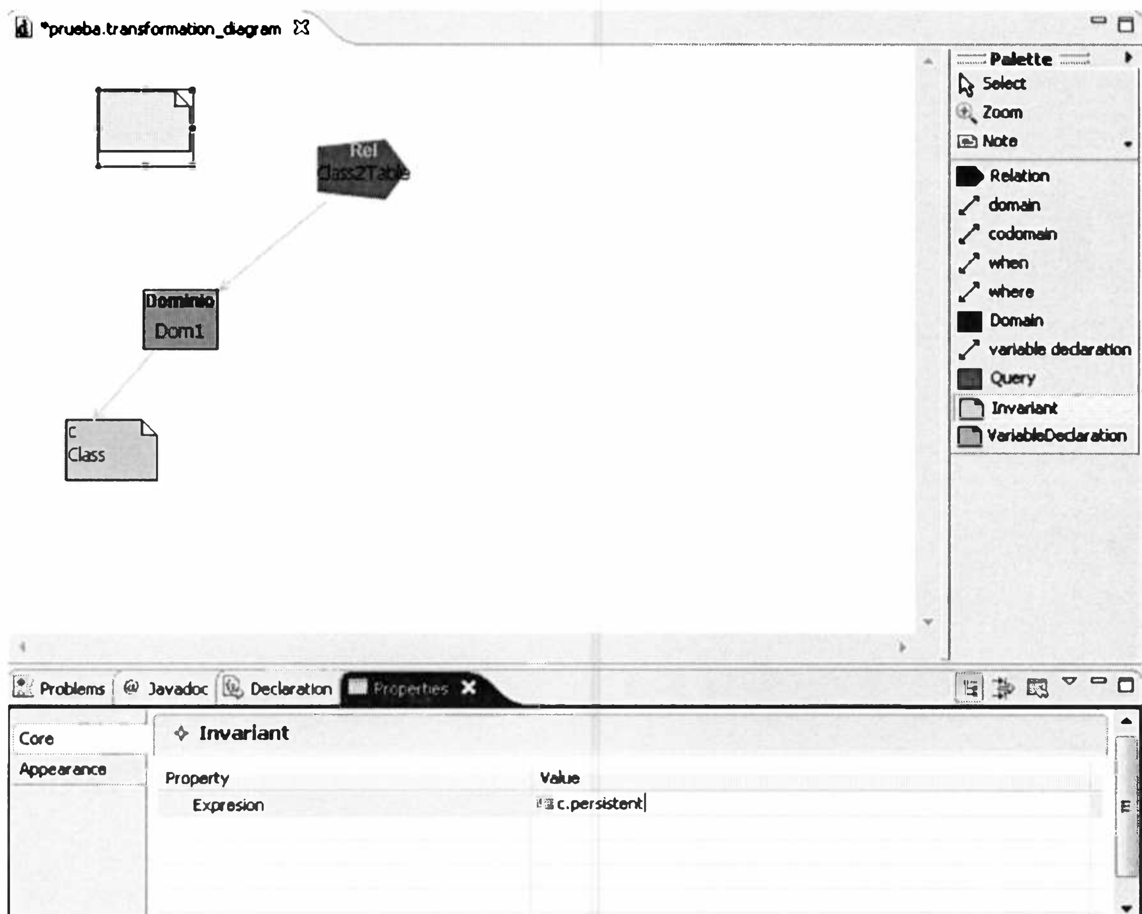


Figura 10.8 - Creación de una figura Invariant

10.2.8 – Creación de una relación de When

Para unir una Relation con un Invariant a través de una relación When se debe presionar en el icono "↗ when". Luego se debe realizar un clic sobre la figura Relation que se quiere comprender y se debe arrastrar el mouse sin soltar el botón hasta el Invariant que se quiere alcanzar. Al soltar dicho botón tiene que aparecer un link que une la Relation con el Invariant con la palabra *when.* Esta relación no posee propiedades. Para ver si la composición quedó correctamente se debe visualizar las propiedades de la figura Relation en la opción *when*(ver figura 10.9).

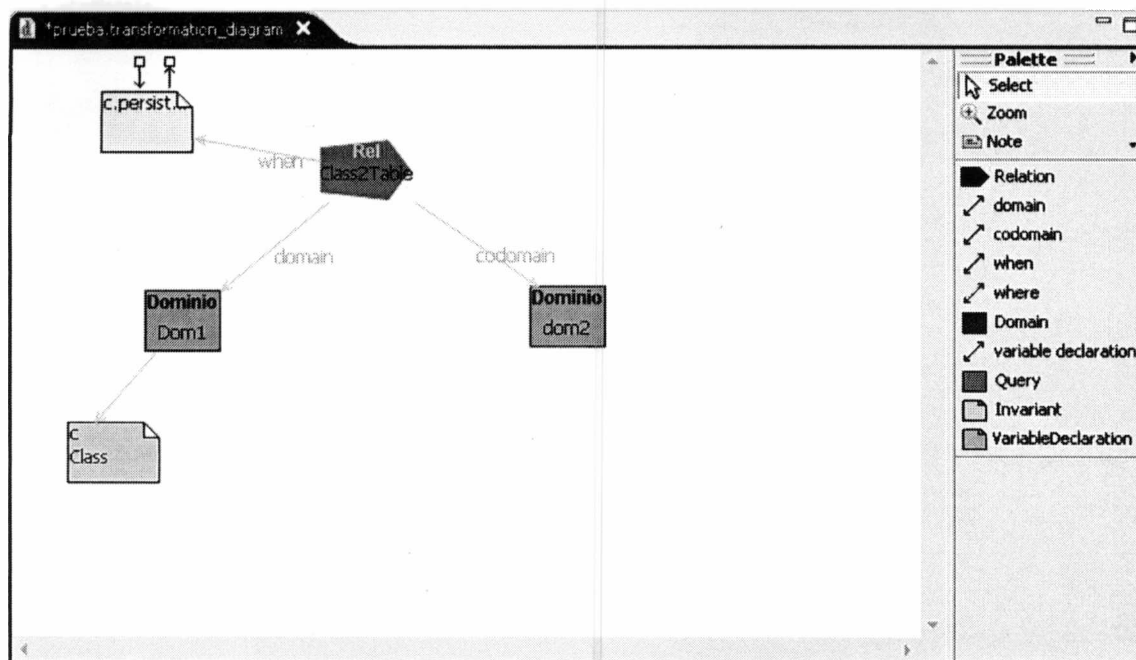


Figura 10.9 - Creación de una relación when

10.2.9 – Creación de una relación de Where

Para unir una Relation con un Invariant a través de una relación Where se debe presionar en el icono "↗ where". Luego se debe realizar un clic sobre la figura Relation que se quiere comprender y se debe arrastrar el mouse sin soltar el botón hasta el Invariant que se quiere alcanzar. Al soltar dicho botón tiene que aparecer un link que une la Relation con el Invariant con la palabra *where*(ver figura 10.10). Esta relación no posee propiedades. Para ver si la composición quedo correctamente se debe visualizar las propiedades de la figura Relation en la opción *where.*

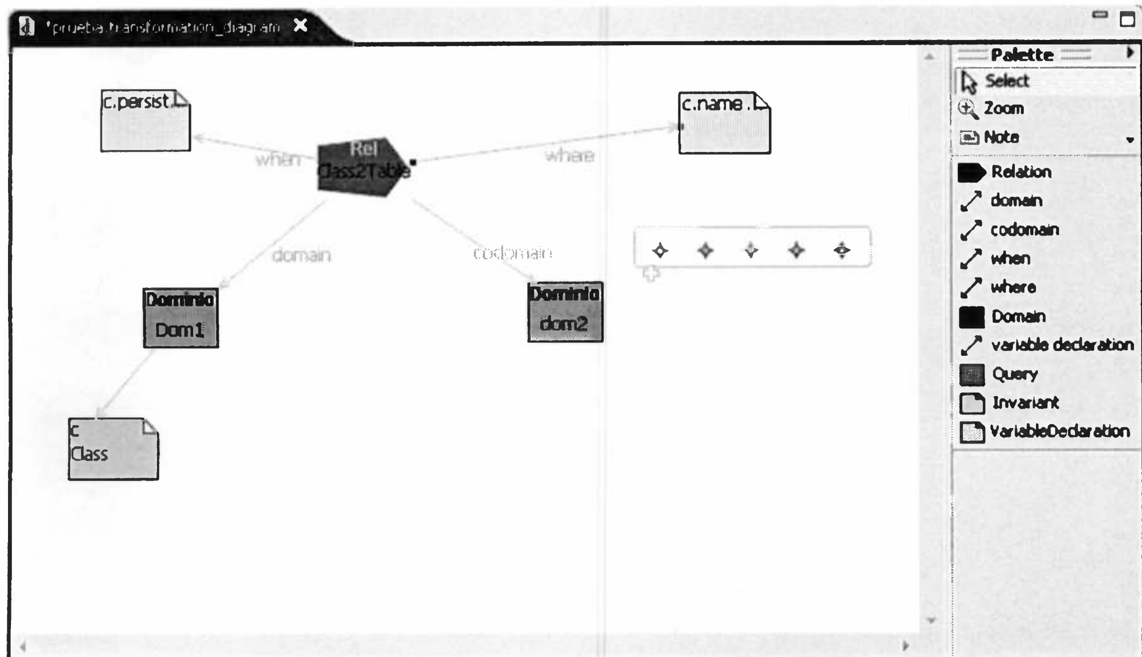
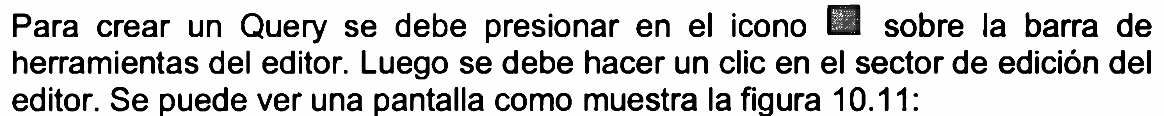


Figura 10.10 - Creación de una relación where

10.2.10 – Creación de una figura Query

Para crear un Query se debe presionar en el icono  sobre la barra de herramientas del editor. Luego se debe hacer un clic en el sector de edición del editor. Se puede ver una pantalla como muestra la figura 10.11:

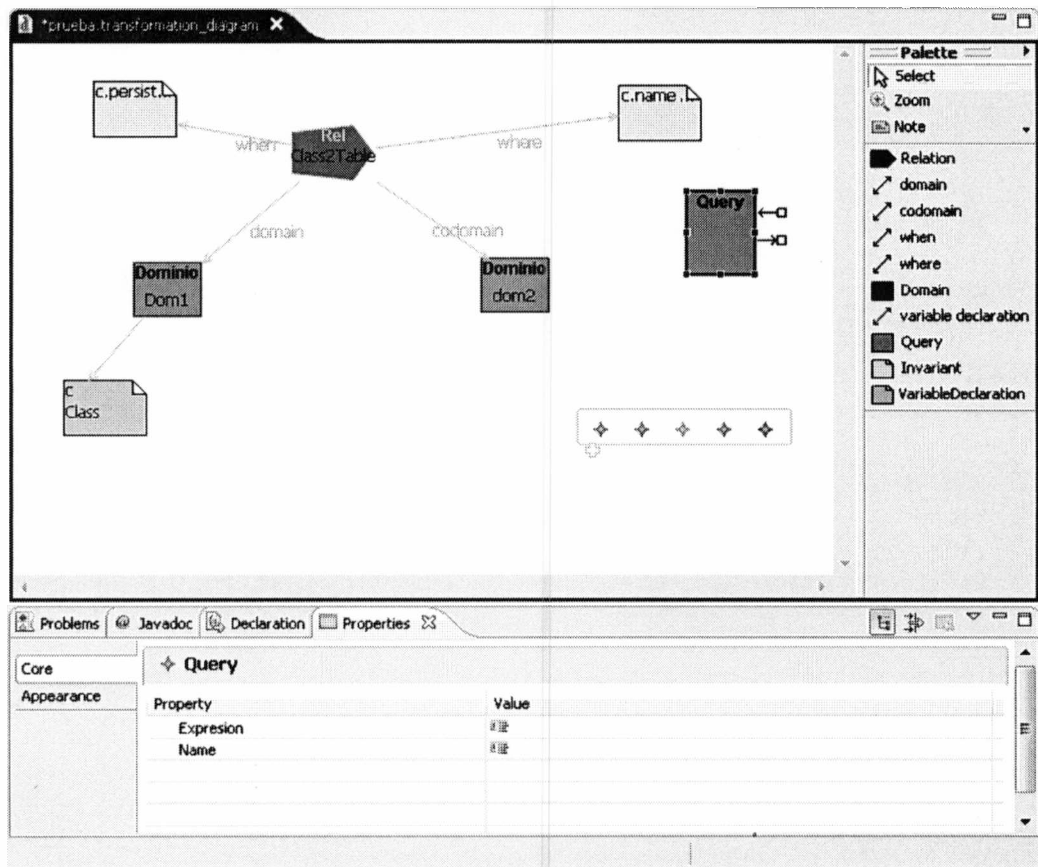


Figura 10.11 Creación de una figura Query

Para modificar sus propiedades se debe presionar botón derecho sobre la figura y elegir la opción *Show Properties View.* Al seleccionar esta opción se podrá modificar el nombre del Query y la propiedad Expresión (es una consulta OCL que se realiza sobre la transformación).

10.2.11 – Finalizando la edición de la Transformación

Una vez que se van agregando uno a uno todos los componentes de la Transformación se le puede modificar el nombre a la transformación, sin seleccionar ninguna figura y presionando botón derecho, se puede elegir la opción *Show Properties View.* Ahí se puede modificar el nombre de la transformación que se está editando. La imagen de la figura 10.12 muestra una transformación finalizada, lista para ser evaluada.

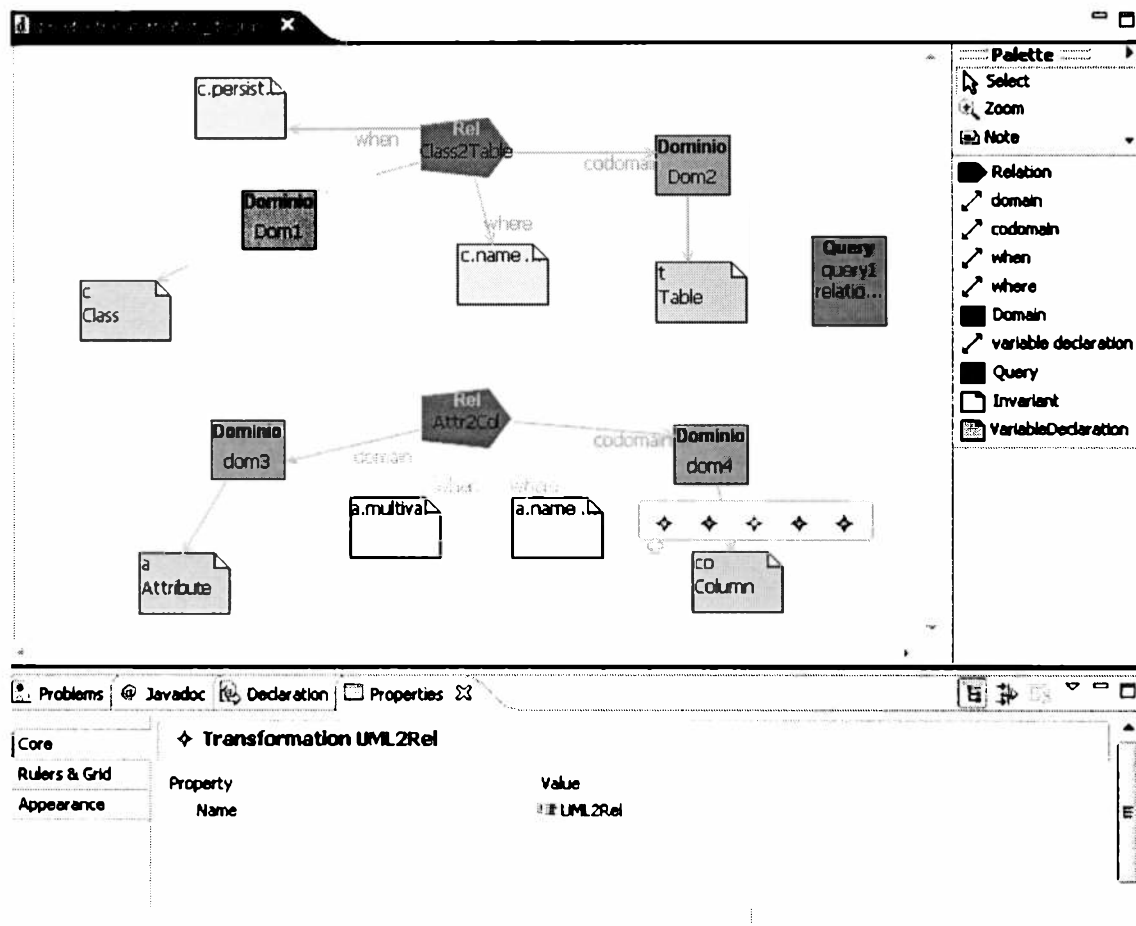


Figura 10.12 – Vista de la transformación finalizada

10.3 - Carga de los metamodelos

Para poder seleccionar la información en la propiedad *EClassifier* de las *Variables Declaration* se deberá primero realizar la carga de los metamodelos a la transformación. Para realizar esto se debe presionar botón derecho del mouse sobre el archivo *transformation_diagram* y aparecerá una pantalla como la que muestra la figura 10.13. Luego se debe seleccionar la opción Cargar Metamodelos. Una vez seleccionada dicha opción se abrirá una ventana que

contiene dos sectores, Transformación y Modelos (ver figura 10.13). En el primero aparece el path junto con el nombre de la transformación que se esta editando. En el segundo sector aparecen dos entradas de texto en los que se deberán seleccionar los archivos correspondientes a los metamodelos de entrada y de salida. Para especificar cualquiera de los metamodelos se deberá presionar el botón *Cargar*. Ahí se abrirá otra ventana que posee una entrada de texto y dos botones, uno para buscar el archivo en el *File System* y otro para buscarlo en el *Workspace*. Una vez se seleccionó el archivo se debe presionar el botón Ok para cerrar la ventana.
Cuando se especificaron los dos metamodelos se debe presionar el botón OK para cerrar la ventana principal. Luego que ocurre esto el plugin procesa todos los objetos de los dos metamodelos y los almacena en el contexto para luego poder ser seleccionados en la vista de las propiedades.

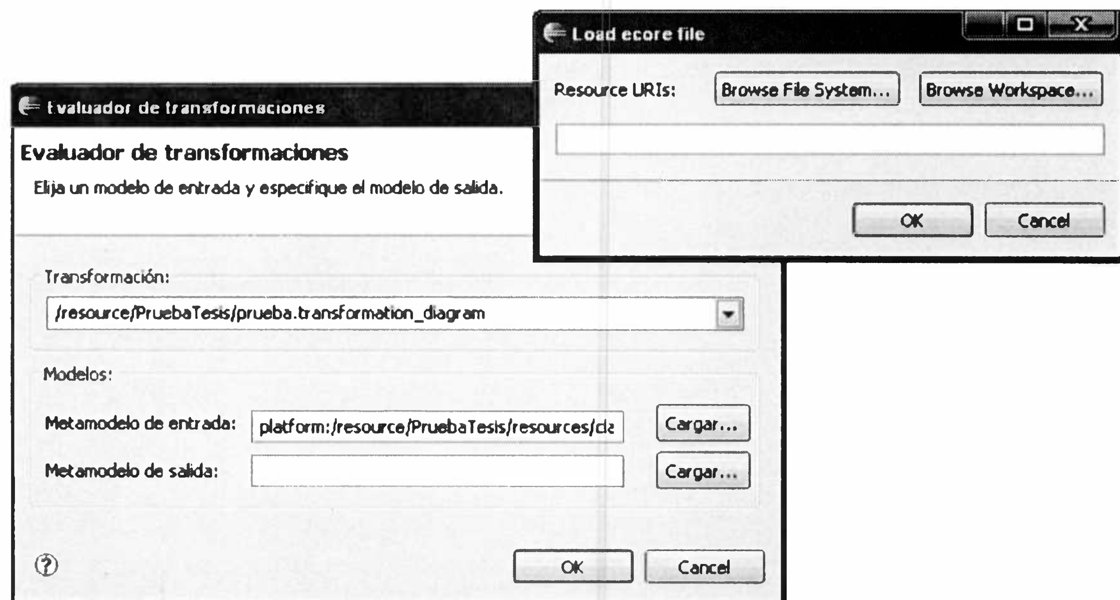


Figura 10.13 – Vista de las ventanas para cargar los metamodelos

10.4 - Validación de la transformación

Una vez que creada la transformación debe ser validada utilizando dos instancias de metamodelos (también llamados modelos). Para realizar esta validación se debe presionar botón derecho del mouse sobre el archivo con extensión "*transformation*". Se debe elegir la opción Validar Transformación, esto hará que se abra una nueva ventana, como la que muestra la figura 10.14.

Dicha ventana esta formada por tres sectores:

- Transformation: Posee el nombre de la transformación que se esta validando.

- Models: En este sector se encuentran dos entradas de texto en los que se deberán seleccionar los archivos correspondientes a los modelos de entrada y de salida. Para especificar cualquiera de los modelos se

deberá presionar el botón *Load*. Ahí se abrirá otra ventana que posee una entrada de texto y dos botones, uno para buscar el archivo en el *File System* y otro para buscarlo en el *Workspace*. Una vez se seleccionó el archivo se debe presionar el botón Ok para cerrar la ventana. Cuando fue cargado el segundo modelo se podrá visualizar los resultados de las validaciones en el tercer sector.

- Result List: En este último sector es donde se pueden visualizar los resultados de las validaciones. En primer lugar se van a visualizar la lista de todas las relaciones de la transformación. Al seleccionar alguna de estas se mostrará el resultado de la misma. Estas relaciones se pueden expandir para poder ver las invariantes *when* y *where*; al seleccionar alguna de estas dos se mostrara en el visor OCL la definición de la misma, como así también el resultado de haber aplicado dicha invariante a todas las instancias. Las invariantes anteriormente nombradas serán aplicadas, como se dijo, a cada una de las instancias dentro de los modelos (por ejemplo, si se debe aplicar al elemento Class, y el modelo posee 10 instancias de Class, esta invariante será aplicada 10 veces), por lo que se podrán ver todos los resultados expandiendo alguna de las invariantes. Cada uno de estos resultados podrán ser seleccionados para poder ver los valores que toman las variables (en el visor Variables), como así también el resultado obtenido al haber aplicado la invariable a esa instancia.

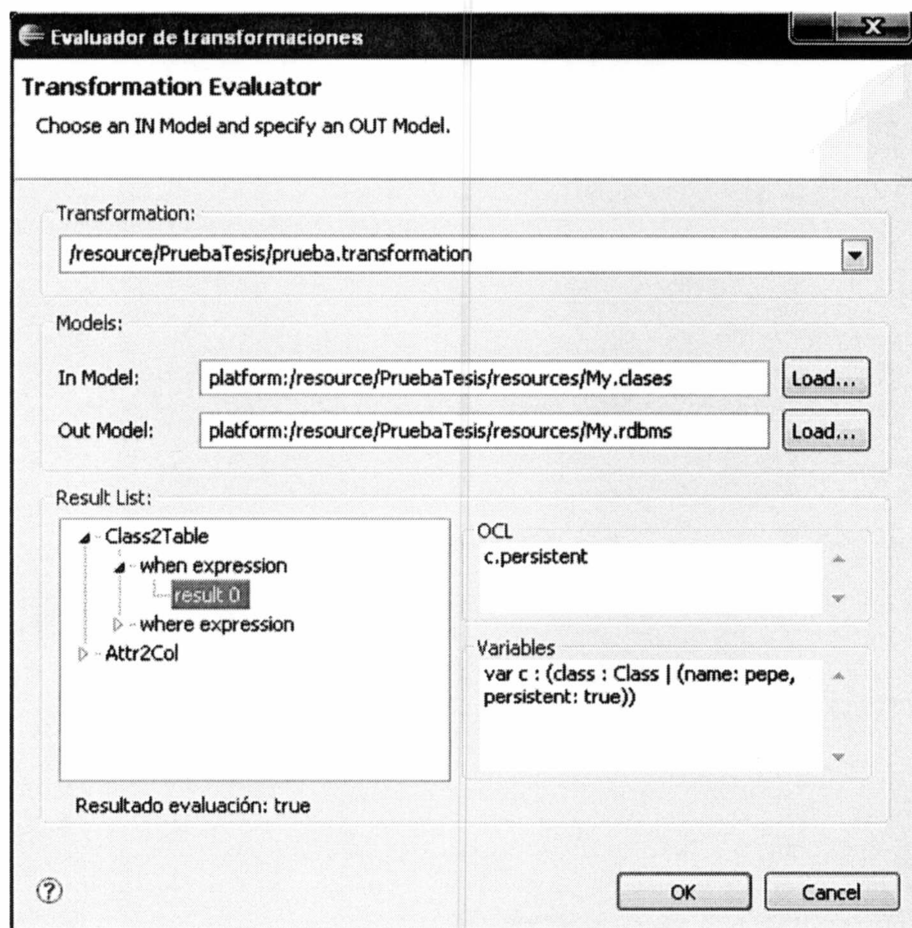


Figura 10.14 – Vista del evaluador de transformación

GLOSARIO

- *Sistema*: Los conceptos MDA se presentan en términos de algún sistema existente o planteado. El sistema puede incluir:
    i. Un programa.
    ii. Un sistema de computadora simple.
    iii. Alguna combinación de partes de diferentes sistemas, personas, una empresa, etc.
- *Modelo*: Un modelo de un sistema es una descripción o especificación de ese sistema y su ambiente para cierto propósito. Un modelo es a menudo presentado como una combinación de dibujos y texto. El texto puede ser un lenguaje de modelado o natural.
- *Orientado a Modelo*: MDA es una aproximación a un desarrollo de sistema, el cual incrementa la potencia de los modelos en ese trabajo. Orientado a modelo es porque provee medios para usar modelos para dirigir el curso de interpretación, diseño, construcción, desarrollo, operación, mantenimiento y modificación
- *Arquitectura*: La arquitectura de un sistema es la especificación de las partes y conectores del sistema y las reglas para las interacciones de las partes usando los conectores. La Arquitectura orientada a Modelo prescribe ciertos tipos de modelos a ser usados, como esos modelos pueden ser preparados y las relaciones de los diferentes tipos de modelos.
- Punto de vista: Un punto de vista en un sistema es una técnica de abstracción usando un conjunto selecto de conceptos arquitecturales y reglas estructuradas, para enfocarse en concerns particulares dentro de ese sistema. Aquí "abstracción" es usado con el significado de proceso de supresión de detalles seleccionados para establecer un modelo simplificado. Los conceptos y reglas pueden ser considerados para formar un lenguaje de "puntos de vista".

  El MDA especifica tres puntos de vista en un sistema:
    I. independiente a la computación.
    II. independiente de la plataforma.
    III. especifico de la plataforma.
- *Vista*: Una vista de un sistema es una representación de ese sistema desde la perspectiva del punto de vista elegido.

- *Plataforma*: Una plataforma es un conjunto de subsistemas y tecnologías que proveen un conjunto de funcionalidades a través de interfaces y patrones de uso especifico, que alguna aplicación soportada por esa plataforma puede usar sin preocuparse por los detalles de como esa funcionalidad provista por la plataforma es implementada.

- *Aplicación*: Se refiere a la funcionalidad a ser desarrollada. Un sistema esta descrito en términos de una o más aplicaciones soportadas por una o más plataformas.

- *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.

- *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 unciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de software reutilizables.

- Diagrama de Clases: En UML, un diagrama de clases es un tipo de diagrama estático que describe la estructura de un sistema mostrando sus clases, atributos y las relaciones entre ellos. Los diagramas de clases son utilizados durante el proceso de análisis y diseño de los sistemas informáticos, donde se crea el diseño conceptual de la información que se manejará en el sistema, y los componentes que se encargaran del funcionamiento y la relación entre uno y otro.

- *API*: Una API (del inglés Application Programming Interface - Interfaz de Programación de Aplicaciones) es el conjunto de funciones y procedimientos (o métodos si se refiere a programación orientada a objetos) que ofrece cierta librería para ser utilizado por otro software como una capa de abstracción.

- 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*: Es el nombre que recibe el estándar para el intercambio de metamodelos usando XML. Su principal objetivo es permitir un intercambio de meta-información entre herramientas de modelado basadas en UML y repositorios de meta-información basados en MOF en heterogéneos entornos distribuidos. El hecho de incluir tres estándares como XML, UML y MOF, permite a los desarrolladores de sistemas distribuidos compartir modelos de objetos y otra información sobre Internet.