



FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Especificación e Implementación de Transformaciones de Modelos de Software aplicando Estándares

Autores: Simoncini German, Garcia Luciano

Director: Prof. Dra. Claudia Fabiana Pons

Codirector:

Asesor profesional:

Carrera: Licenciatura en Informática - Plan 90

Resumen

La comunidad informática ha buscado mejorar siempre la calidad de los procesos de desarrollo de software. Con el paso del tiempo ha tomado fuerza el concepto de desarrollo dirigido por modelos (MDD) que propone mejorar la calidad de los procesos con la idea de que el punto de partida del desarrollo sean los modelos y las transformaciones entre los mismos. En respuesta a ello el Object Management Group, a través de su iniciativa denominada Model Driven Architecture (MDA), ha elaborado una serie de estándares para impulsar el desarrollo dirigido por modelos, parte de ello es la definición de un lenguaje de transformaciones estándar el cual denominaron QVT.

El objetivo de este trabajo es la realización de un repositorio de transformaciones de modelos de software el cual permitirá la difusión y expansión de dichas transformaciones que se encuentren publicadas en él. Las transformaciones que contendrá el repositorio se basaran en los estándares definidos por la OMG para tal fin. Las mismas estarán desarrolladas en lenguaje QVT y resolverán problemas recurrentes presentes en la ingeniería de software.

Palabras Claves

Desarrollo Dirigido por Modelos
MDD
Lenguajes de Transformación
QVT
Repositorio de Transformaciones
OMG
Plataforma Eclipse
Eclipse Modeling Project
Java
Frameworks

Trabajos Realizados

Analizamos y decidimos la implementación del lenguaje QVT para el desarrollo de transformaciones. Definimos el conjunto de transformaciones a desarrollar e incluir en el repositorio QVT. Definimos metamodelos en base a una simplificación del modelo UML. Diseñamos e implementamos el repositorio de transformaciones. Definimos un caso de estudio y le aplicamos un conjunto de transformaciones.

Conclusiones

Con el desarrollo del repositorio de transformaciones QVT le brindamos a la comunidad MDD un gran medio de difusión y expansión. Además dejamos gran precedente en el uso del lenguaje de transformación estándar QVT implementando un conjunto de transformaciones y demostrando su aplicabilidad a través de un caso de estudio simple y de fácil de entendimiento. En base a los resultados hemos comprobado que el paradigma MDD es un enfoque muy poderoso y puede llevarse a la realidad con muy buenos resultados.

Trabajos Futuros

Se proponen las siguientes líneas como trabajos futuros: implementar en lenguaje QVT el resto de las transformaciones existentes en el repositorio ATL. Extender las soluciones presentadas en esta tesis para el modelo UML completo. Plugins que permitan construir ecotes, sobre la plataforma eclipse, que usen el metamodelo SimpleUML como base para la construcción de diagramas simplificados. Implementación de un foro dentro del repositorio QVT donde discutir temas afines al desarrollo de transformaciones QVT.

Agradecimientos:

Germán

A mis viejos, Mary y Tano, y a mis hermanos, Tin, Flor y Vicky, porque son la razón de mi vida y porque siempre me apoyaron en esta etapa. A mis abuelos maternos, José y Silvia, y paternos, Adriano y Ana.

A Lucho, alias el cabecha, por ser la persona que me acompañó y motivó para terminar mi carrera, por ser el amigo y hermano del alma que me regaló la vida. Y a Nati por soportarnos en su casa cada día de estudio y por ayudarnos cada vez que lo necesitamos.

A mis amigos de estudio, Guille, Sabri, Pini, Sabino, Juancito, Facha, Mati y Bocha, con los cuales compartí momentos inolvidables y a los que le debo gran parte de lo que soy hoy como persona y como profesional.

Luciano

A mis padres Rubén y Mabel, a mi hermana Mariela y mis abuelos que impulsaron y apoyaron mi elección hacia este camino.

A mi compañero de tesis y amigo del alma Pato, el cabechon, que me regala día a día su incondicional amistad y que con su energía y tenacidad permitió llegar al final de este largo camino.

A Guille y Mike por su constante apoyo y colaboración en todo momento.

A mis suegros Ali y Riky por estar siempre presente y brindarme toso su amor y cariño.

Por último al amor de mi vida Nati, por toda su banca y constante empuje que hizo posible que hoy este aquí cerrando una de las etapas más importante en mi vida.

Germán y Luciano

A Claudia por ser la persona que nos orientó y marcó el camino hacia la recta final.

Índice:

INTRODUCCIÓN	8
1.1 OBJETIVO	8
1.2 APOORTE	8
1.3 ORGANIZACIÓN DE TESIS.....	8
2 CONCEPTOS GENERALES.....	9
2.1 PROBLEMAS DEL DESARROLLO TRADICIONAL DE SOFTWARE.....	9
2.1.1 <i>Productividad</i>	9
2.1.2 <i>Portabilidad</i>	11
2.1.3 <i>Interoperabilidad</i>	12
2.1.4 <i>Mantenimiento y documentación</i>	12
2.2 SOLUCIÓN: MDD	13
2.3 MDE Y LA TRANSFORMACIÓN DE MODELOS	13
2.4 BENEFICIOS DE MDE SOBRE EL DESARROLLO TRADICIONAL DE SOFTWARE	14
2.4.1 <i>El problema de la productividad</i>	14
2.4.2 <i>El problema de la portabilidad</i>	14
2.4.3 <i>El problema de la interoperabilidad</i>	15
2.4.4 <i>El problema del mantenimiento y la documentación</i>	15
2.5 MDA: LA ARQUITECTURA DIRIGIDA POR MODELOS	15
2.6 CONCEPTOS BÁSICOS SOBRE TRANSFORMACIONES.....	17
2.6.1 <i>Introducción</i>	17
2.6.2 <i>Arquitectura 4 capas</i>	19
2.6.3 <i>MOF (Infraestructura 2.0)</i>	21
3 LENGUAJES DE TRANSFORMACIÓN	24
3.1 QUE ES UNA TRANSFORMACIÓN	24
3.2 QUE ES UN LENGUAJE DE TRANSFORMACIÓN	26
3.3 QVT	26
3.3.1 <i>Descripción General de QVT</i>	26
3.3.2 <i>QVT Declarativo</i>	27
3.3.3 <i>QVT Operacional</i>	30
3.4 ATL	30
3.5 OCL.....	31
4 HERRAMIENTAS DE TRANSFORMACIÓN DE MODELOS MÁS USADAS.....	32
4.1 VIATRA (VISUAL AUTOMATED MODEL TRANSFORMATIONS)	32
4.2 TEFKAT	32
4.3 ATL (ATLAS TRANSFORMATION LANGUAGE)	33
4.4 EPSILON	33
4.5 ATOM (A TOOL FOR MULTI-FORMALISM AND META-MODELING).....	34

4.6	MOLA (MODEL TRANSFORMATION LANGUAGE)	34
4.7	MOFSSCRIPT	35
4.8	KERMETA	36
4.9	KENT MODEL TRANSFORMATION LANGUAGE.....	37
4.10	MEDINI BY IKV++ TECHNOLOGIES.....	37
4.11	SMARTQVT BY TELECOM FRANCE.....	37
4.12	TOGETHER ARCHITECTURE/QVT BY BORLAND (QVT/OPERATIONAL)	38
4.13	ECLIPSE MODELING PROJECT.....	39
4.13.1	EMF (Eclipse Modeling Framework)	39
4.13.2	Model To Model Transformation (M2M)	40
4.14	CONCLUSIONES.....	40
5	FRAMEWORKS INVESTIGADOS PARA LA IMPLEMENTACIÓN DEL REPOSITORIO	41
5.1	JSF	41
5.1.1	Las aplicaciones JSF típicamente contienen:	41
5.1.2	Existen diferentes implementaciones de JSF:.....	42
5.1.3	Los principales componentes de la tecnología Java Server Faces son:	42
5.1.4	Beneficios de la Tecnología JavaServer Faces.....	43
5.2	JPA.....	44
5.2.1	La API de persistencia Java	44
5.2.2	El lenguaje de consulta	44
5.2.3	Metadata para el mapeo Objeto/Relacional.....	45
5.3	SEAM	45
5.3.1	Características	46
5.4	MAVEN2.....	48
5.4.1	Objetivo	49
5.4.2	Características	49
6	CASO DE ESTUDIO – TRANSFORMACIÓN UML2RDBMS	50
6.1	LENGUAJE QVTR.....	50
6.2	LENGUAJE QVTO	51
6.3	CONCLUSIÓN.....	53
7	NUESTRA PROPUESTA (ESPECIFICACIÓN E IMPLEMENTACIÓN DE TRANSFORMACIONES DE MODELOS DE SOFTWARE APLICANDO ESTÁNDARES)	54
7.1	MOTIVACIÓN: REPOSITORIO DE TRANSFORMACIONES DE ATL	54
7.2	ELECCIÓN DE HERRAMIENTA Y LENGUAJE DE DESARROLLO.	54
7.3	VISIÓN DE LA HERRAMIENTA DE DESARROLLO ECLIPSE MODELING.....	56
7.4	TRANSFORMACIONES SELECCIONADAS PARA LA TESIS.....	59
7.4.1	JavaSource2Table	59
7.4.2	InroducePrimaryKey.....	64
7.4.3	ReplaceInheritanceByAssociation.....	68
7.4.4	ReplaceAssociationbyForeignKey	71
7.4.5	ReplaceManyToMany2Association.....	75
7.4.6	RemoveAssociationClass.....	79
7.4.7	RemoveRedundantClass	83

7.4.8	UMLObserver2JavaObserver	87
7.4.9	UML2TestUML	97
8	REPOSITORIO DE TRANSFORMACIONES	105
8.1	DESCRIPCIÓN GENERAL	105
8.2	ARQUITECTURA	105
8.2.1	<i>Visión de general de la arquitectura del repositorio de transformaciones</i>	<i>107</i>
8.3	DISEÑO	108
8.3.1	<i>Visión General de packages</i>	<i>108</i>
8.3.2	<i>Dependencia entre packages</i>	<i>109</i>
8.3.3	<i>Visión detallada de packages</i>	<i>109</i>
8.4	IMPLEMENTACIÓN	113
8.4.1	<i>Visión del IDE de desarrollo y Jboss Tools</i>	<i>113</i>
8.4.2	<i>Visión general de la funcionalidad del repositorio</i>	<i>115</i>
8.4.3	<i>Visión detallada de la funcionalidad del repositorio</i>	<i>118</i>
9	CASO DE ESTUDIO	123
9.1	DEFINICIÓN DEL MODELO UML	123
9.2	APLICACIÓN DE TRANSFORMACIONES	124
9.2.1	<i>Transformacion 1: RemoveAssociationClass</i>	<i>124</i>
9.2.2	<i>Transformacion 2: ReplaceManyToMany2Association</i>	<i>124</i>
9.2.3	<i>Transformacion 3: ReplaceInheritanceByAssociation</i>	<i>125</i>
9.2.4	<i>Transformacion 4: IntroducePrimaryKey</i>	<i>126</i>
9.2.5	<i>Transformacion 5: ReplaceAssociationbyForeignKey</i>	<i>127</i>
10	CONCLUSIÓN Y TRABAJOS FUTUROS	129
10.1	CONCLUSIÓN	129
10.2	TRABAJOS FUTUROS	130
11	REFERENCIAS	130

Índices e Imágenes

Imagen 1 - Ciclo de vida tradicional del desarrollo de software	10
Imagen 2 - Pasos y Niveles en el desarrollo MDA	17
Imagen 3 - MetaModelo y MetaMetaModelo según las OMG	18
Imagen 4 - Proceso completo de transformación de modelos	18
Imagen 5 - Modelos, Lenguajes, Metamodelos y Metalenguajes	19
Imagen 6 - Modelo de 4 capas	21
Imagen 7 - Sintaxis Abstracta MoF	22
Imagen 8 - Estructura de la herramienta de transformación "abierta"	25
Imagen 9 - Transformación UML a Java	25
Imagen 10 - Relaciones entre metamodelos QVT	27
Imagen 11 - Dependencias de paquetes en la especificación QVT	27
Imagen 12 - Modelo de EMF	40
Imagen 13 - Ciclo JSF	43
Imagen 14 - Instanciación de una transformación	51
Imagen 15 - Código Fuente Java	59
Imagen 16 - Invocaciones de métodos totalizadas para FirstClass y SecondClass.	60
Imagen 17 - Metamodelo básico, JavaSource	61
Imagen 18 - Definición Abstracta Table	61
Imagen 19 - Modelo IntroducePrimaryKey	65
Imagen 20 - Metamodelo Para IntroducePrimaryKey	66
Imagen 21 - Modelo ReplaceInheritanceByAssociation	68
Imagen 22 - Metamodelo ReplaceInheritanceByAssociation	69
Imagen 23 - Modelo ReplaceAssociationbyForeignKey	72
Imagen 24 - Metamodelo ReplaceAssociationbyForeignKey	73
Imagen 25 - Modelo ReplaceManyToMany2Association	76
Imagen 26 - Metamodelo ReplaceManyToMany2Association	77
Imagen 27 - Modelo RemoveAssociationClass	80
Imagen 28 - Metamodelo RemoveAssociationClass	80
Imagen 29 - Modelo RemoveRedundantClass	83
Imagen 30 - Motamodelo RemoveRedundantClass	84
Imagen 31 - Metamodelo UMLObserver2JavaObserver	88
Imagen 32 - Metamodelo1 UML2TestUML	98
Imagen 33 - Metamodelo2 UML2TestUML	99
Imagen 34 - Visión de general de la arquitectura del repositorio	107
Imagen 35 - Visión General de packages	108
Imagen 37 - Visión detallada de packages - Controler	109
Imagen 36 - Dependencia entre packages	109
Imagen 38 - Visión detallada de packages - Model	110
Imagen 39 - Visión detallada de packages –Manager	111
Imagen 41 - Visión detallada de packages - Servlet	111

Imagen 40 - Visión detallada de packages - Session	111
Imagen 42 - Visión detallada de packages - Util	112
Imagen 43 - Modelo De Datos	113
Imagen 44 - Visión del ide + Jboss.....	114
Imagen 45 - Visión del Ide vista código Java	114
Imagen 46 - Visión general del repositorio - Tranformaciones Publicadas	115
Imagen 47 - Visión general del repositorio - Subir Transformaciones.....	116
Imagen 48 - Visión general del repositorio - Login	116
Imagen 49 - Visión general del repositorio - Administración de Transformaciones.....	117
Imagen 50 - Visión detallada del repositorio - Transformaciones Publicadas	119
Imagen 51 - Visión detallada del repositorio - Subir Transformaciones 2	120
Imagen 52 - Visión detallada del repositorio - Subir Transformaciones 1	120
Imagen 53 - Administracion de Transformaciones - Principal	121
Imagen 54 - Admistración de Transformaciones - DownLoad	122
Imagen 55 - Admistración de Transformaciones - Barra Lateral.....	122
Imagen 56 - Modelo UML Caso de Estudio	123
Imagen 57 - Modelo resultante de aplicar RemoveAssociationClass	124
Imagen 58 - Modelo resultante de aplicar ReplaceManyToMany2Association	125
Imagen 59 - Modelo resultante de aplicar ReplaceInheritanceByAssociation	126
Imagen 60 - Modelo resultante de aplicar IntroducePrimaryKey	127
Imagen 61 - Resultado de aplicar ReplaceAssociationbyForeignKey.....	128

Introducción

1.1 Objetivo

El objetivo de este trabajo es la realización de un repositorio de transformaciones de modelos de software el cual permitirá la difusión y expansión de dichas transformaciones que se encuentren publicadas en él.

Las transformaciones que contendrá el repositorio se basarán en los estándares definidos por la OMG para tal fin. Las mismas estarán desarrolladas en lenguaje QVT y resolverán problemas recurrentes presentes en la ingeniería de software.

Para la difusión y expansión hacia la comunidad MDD de las transformaciones QVT se desarrollará un repositorio, cuya arquitectura y diseño facilitará el objetivo propuesto.

Lo descripto anteriormente se logra mediante los siguientes sub-objetivos:

- Investigación del nuevo proceso de desarrollo de software (MDD) y los principales lenguajes de transformación.
- Investigación de los modelos de transformaciones presentados en el repositorio ATL para identificar y seleccionar el subconjunto de transformaciones que se implementarán en la tesis.
- El estudio del estado de madurez de las herramientas que implementan transformaciones de modelos y cumplen con la especificación estándar de QVT.
- El desarrollo de un repositorio de transformaciones QVT.

1.2 Aporte

Este trabajo presentará un conjunto de transformaciones QVT que será un gran aporte para la motivación de futuros desarrolladores de nuevas transformaciones.

Otro aporte interesante es el repositorio el cual actuará como vía de comunicación y propagación de todo su contenido.

1.3 Organización de tesis

Como explicamos en el capítulo anterior, este trabajo de tesis tiene como objetivo principal la realización de un repositorio de transformaciones de modelos de software el cual permitirá la difusión y expansión de dichas transformaciones que se encuentren publicadas en él.

Nuestro trabajo está dividido en tres grandes partes:

- La primera de ellas tiene como objetivo situar a nuestro trabajo en un marco teórico-conceptual. Los capítulos 2 y 3 explican conceptos sobre MDD y lenguajes de transformaciones que serán aplicados en el resto de la tesis.

- La segunda parte tiene como objetivo la investigación de las herramientas involucradas en el desarrollo del trabajo. El capítulo 4 aborda la investigación de las herramientas para la implementación de las transformaciones en QVT que fueron evaluadas. El capítulo 5 hace hincapié en el análisis de los frameworks contemplados para la realización del repositorio de transformaciones.
- La Tercera parte tiene como objetivo abordar los aspectos prácticos del diseño y construcción, tanto de las transformaciones como del repositorio. El capítulo 6 cubre un caso de estudio simple para el entendimiento y comprensión del lenguaje de transformación QVT en sus dos implementaciones, la operacional y la declarativa. El capítulo 7 detalla las transformaciones desarrolladas, el capítulo 8 describe el repositorio de transformaciones, y el capítulo 9 presenta un caso de estudio en el cual se aplica un conjunto de transformaciones desarrolladas en la tesis.

2 Conceptos Generales

Este capítulo describe el contexto en el que se enmarca esta tesis; se describen los problemas principales encontrados en los desarrollos actuales de sistemas de software, y se presenta un nuevo paradigma que promete solucionarlos. A continuación se mencionan los principales aportes a la Ingeniería de Software Conducida por Modelos y finalmente, se introducen los conceptos básicos de transformaciones.

2.1 Problemas del desarrollo tradicional de software

El desarrollo del software se encuentra actualmente en una etapa en la que debe superar un importante número de problemas. Hoy en día, debido a la gran cantidad de tecnologías existentes, escribir software es una tarea difícil. Con cada nueva tecnología se requiere mucho trabajo para poder emplearla teniendo en cuenta el tiempo de aprendizaje, el tiempo de desarrollo, etc. Este trabajo debe ser hecho una y otra vez por cada una de estas nuevas tecnologías.

Por otro lado, la mayoría de los sistemas actuales usan más de una tecnología, y necesitan comunicarse con otros sistemas. También hay que tener en cuenta que los requisitos cambian continuamente. A continuación, se analizan algunos de los problemas más importantes encontrados durante el desarrollo del software y se intenta descubrir su causa. En el siguiente punto se presentará una propuesta (MDE/MDD) que intenta solucionar los problemas planteados.

2.1.1 Productividad

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

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

No importa si el proceso es iterativo e incremental, o es el tradicional en cascada. Los documentos y los diagramas se producen sólo en las primeras fases (de la uno a la tres). Estos documentos incluyen descripciones de los requisitos escritos en lenguaje natural, dibujos, y a menudo incluyen diagramas expresados en UML [1][2][3][4]. Entre estos últimos, están los diagramas de casos de uso, diagramas de clase, diagramas de interacción y diagramas de actividad.

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

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

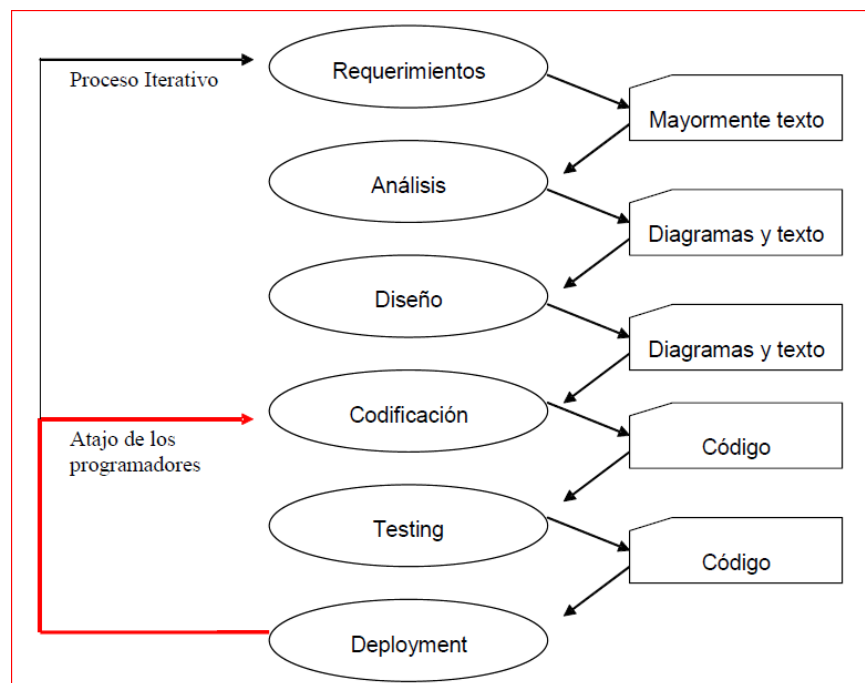


Imagen 1 - Ciclo de vida tradicional del desarrollo de software

De estos problemas surgen varias soluciones. Una de ellas es la idea de Extreme Programming [5] (XP) que se volvió popular en muy poco tiempo.

Una de las razones de esto es que reconoce al código como la fuerza impulsora del desarrollo del software. Afirma que las únicas fases en el proceso del desarrollo que son realmente productivas son la codificación y testing.

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

Planteado así, quedan dos alternativas: o utilizamos nuestro tiempo en las primeras fases del desarrollo del software construyendo la documentación y diagramas de alto nivel, o utilizamos nuestro tiempo en la fase de mantenimiento descubriendo lo que hace realmente el software. Muchas veces se considera la tarea de la documentación como tareas adicionales.

Se considera que escribir código es productivo, pero hacer modelos y documentación no lo es. No obstante, en un proyecto maduro de software estas tareas deben realizarse.

2.1.2 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.). Muchas compañías necesitan seguir estas nuevas tecnologías por buenas razones:

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

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

cambiar utilizando la vieja tecnología, pero que ahora necesite comunicarse con sistemas nuevos, los cuales serán construidos usando nuevas tecnologías.

2.1.3 Interoperabilidad

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

2.1.4 Mantenimiento y documentación

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

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

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

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

2.2 Solución: MDD

MDD es el acrónimo de Desarrollo Dirigido por Modelos (Model Driven Development). MDD concibe al desarrollo de software bajo la idea central de que los artefactos fundamentales del desarrollo son los modelos (y no los programas). Implica la generación automática de programas a partir de modelos. En MDD “El modelo es la implementación” MDA no es más que un MDD particular, con nombre propio (y estándar definido por la OMG).

MDE es el acrónimo de Ingeniería Dirigida por Modelos (Model Driven Engineering) y refiere al uso sistemático de modelos, como los elementos principales en la ingeniería de software, durante el ciclo de vida completo del proyecto.

MDD vs MDE: Por ser MDD una marca registrada de la OMG, la comunidad científica utiliza el término MDE para referirse a ideas y conceptos relacionados con la ingeniería basada en modelos, sin centrarse, exclusivamente, en los estándares de la OMG.

2.3 MDE y la Transformación de Modelos

A lo largo de esta década, la Ingeniería de Software Conducida por Modelos (MDE, acrónimo de Model Driven Engineering) [7] 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 el 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.

El paradigma MDE tiene dos ejes principales: por un lado hace énfasis en la separación entre la especificación de la funcionalidad esencial del sistema y la implementación de dicha funcionalidad usando plataformas tecnológicas específicas. 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); por otro lado, los modelos son considerados los conductores primarios en todos los aspectos del desarrollo de software.

Un PIM es transformado en uno o más PSMs, es decir que para cada plataforma tecnológica específica se genera un PSM específico. La transformación entre modelos constituye el motor del MDE y de esta manera los modelos pasan de ser entidades meramente contemplativas a ser entidades productivas.

En los próximos apartados de esta sección, se enumeran los beneficios principales que aporta MDE al proceso de desarrollo de software y a continuación, presentamos un enfoque concreto (MDA) para esta metodología de desarrollo.

2.4 Beneficios de MDE sobre el desarrollo tradicional de software

Específicamente, enunciaremos los problemas más representativos que aparecen en el desarrollo tradicional de software, analizaremos la causa de estos problemas y veremos cómo MDE los trata, enfatizando fuertemente en la necesidad de contar con herramientas sólidas que automaticen el proceso de transformación de modelos.

2.4.1 El problema de la productividad

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 solo necesita ser definida sólo una vez, y puede ser aplicada en el desarrollo de muchos sistemas. Ya que los desarrolladores necesitan enfocarse solo en los PIM, pueden trabajar independientemente de los detalles de las plataformas. Esos detalles técnicos serán agregados automáticamente por la transformación del PIM al PSM. Esto mejora la productividad de dos maneras:

- a. En primer lugar, los desarrolladores del PIM tienen menos trabajo que hacer ya que los detalles específicos de la plataforma no necesitan ser diseñados, sino que ya están en la definición de la transformación. Pensando en el código, habrá mucho menos código que escribir ya que una gran cantidad de código será generado a partir del PIM.
- b. En segundo lugar, los desarrolladores pueden enfocarse en el PIM en vez de en el código. Esto hace que el sistema desarrollado se acerque más a las necesidades del usuario final, el cual tendrá mejor funcionalidad en menor tiempo.

Estas mejoras en la productividad se pueden alcanzar con el uso de herramientas que automaticen completamente la generación de un PSM a partir de un PIM; es decir, herramientas que hagan automático el proceso de transformación de modelos.

2.4.2 El problema de la portabilidad

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.

Como ejemplo, supongamos que transformamos un PIM (diagrama de clases UML) a dos PSMs, el primero a un modelo Java y el segundo a un modelo de base de datos relacional. Para la clase “Cliente” en el PIM, sabemos que clase Java y que tabla genera. Construir un puente entre ambos elementos es posible. Para retornar un objeto de la base de datos, se deberían leer los datos de la tabla Cliente e instanciar un objeto de la clase Cliente.

La interoperabilidad entre plataformas puede ser realizada por herramientas para transformación de modelos que generen, además de PSMs, puentes entre ellos y posiblemente entre otras plataformas.

2.4.3 El problema de la interoperabilidad

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. Como ejemplo, supongamos que transformamos un PIM (diagrama de clases UML) a dos PSMs, el primero a un modelo Java y el segundo a un modelo de base de datos relacional. Para el concepto “Cliente” definido en el PIM, sabemos qué clase Java genera y qué tabla genera. Construir un puente entre ambos elementos es posible. Para retornar un objeto de la base de datos, se deberían leer los datos de la tabla Cliente e instanciar un objeto de la clase Cliente. La interoperabilidad entre plataformas puede ser realizada por herramientas para transformación de modelos que generen, además de PSMs, puentes entre ellos y posiblemente entre otras plataformas.

2.4.4 El problema del mantenimiento y la documentación

Con MDE los desarrolladores pueden enfocarse solo en el PIM, el cual a su vez tiene un nivel más abstracto que el del código. El PIM es utilizado entonces para generar PSMs, los cuales posteriormente serán utilizados para generar código. Por lo tanto, el modelo será una exacta representación del código. Así el PIM cumplirá la función de documentación de alto nivel necesaria en cualquier sistema de software.

La gran diferencia es que el PIM no se deja de lado luego de ser escrito. Los cambios hechos en el sistema serán efectuados sobre el PIM y regenerando PSMs y código. Esta tarea requerirá su automatización mediante herramientas adecuadas para ello.

2.5 MDA: La 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[8][9][10][36], presentada por el consorcio OMG [11] en noviembre de 2000 con el objetivo de abordar los desafíos de integración de aplicaciones y los continuos cambios tecnológicos.

MDA propone el uso de un conjunto de estándares como Meta Object Facility MOF [12], UML [13], JMI [14] o XMI [15]. Su objetivo es separar la especificación de la funcionalidad del sistema de su implementación sobre una plataforma concreta, por lo que se hace una distinción entre modelos PIM y modelos PSM. En general, el código de una aplicación se puede generar a partir de un modelo PIM, mediante sucesivas transformaciones de modelos hasta llegar al código fuente. La idea es también poder implementar el modelo PIM de un sistema en diferentes plataformas (es decir generar varios modelos PSM), lo que los llevaría a disponer de sistemas más robustos a los cambios tecnológicos, y por otro lado poder realizar las labores de modelado abstrayéndonos totalmente de los detalles de la tecnología que usemos como soporte.

Las transformaciones modelo-modelo y modelo-código juegan un papel crucial en MDA y son necesarios lenguajes de transformación especiales para escribir cada una de las definiciones de transformación que establecen las correspondencias (relaciones, mappings) entre dos lenguajes de modelado (o entre un lenguaje de modelado y un lenguaje de programación).

En la actualidad existen un buen número de herramientas y lenguajes que soportan MDA, entre las que destacan OptimalJ [16], ArcStyler [17] y AndroMDA [18].

Cada herramienta incorpora un mecanismo propio para el manejo de transformaciones. Una iniciativa de OMG es la definición de un lenguaje de transformaciones estándar denominado QVT [15], aún en proceso de estandarización.

Para llevar a cabo todo este proceso MDA define fases que a su vez marcan tres niveles de abstracción del sistema a desarrollar, como muestra la Figura 1-1:

Fase I. Creación de un Modelo Independiente de la Plataforma (Platform Independent Model o PIM), es el modelo con el mayor nivel de abstracción del sistema, describe la funcionalidad del sistema pero omitiendo detalles de cómo y dónde va a ser implementado.

Fase II. Transformación del PIM a uno o varios Modelos Específicos de Plataforma (Platform Dependent Model o PSM). Un PSM es el mismo modelo que el PIM pero describiendo el sistema de acuerdo a una plataforma dada, como .NET o J2EE.

Fase III. Generación del código a partir del PSM. Consiste en hacer un mapping (transformación) del modelo PSM a la plataforma en la que está especificado. Esto se puede realizar de forma automática ya que el PSM se encuentra muy ligado a dicha plataforma.

Sin embargo, pueden observarse en la práctica, situaciones donde existan más de tres niveles de abstracción. Pueden existir transformaciones intermedias entre modelos que repitan la Fase II (PIM/PSM), donde el modelo de salida de una transformación es la entrada para la siguiente, generando así una cadena hasta llegar a la generación de código.

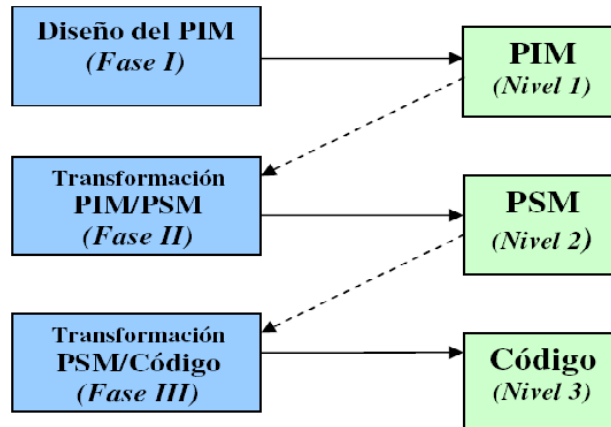


Imagen 2 - Pasos y Niveles en el desarrollo MDA

2.6 Conceptos Básicos sobre Transformaciones

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

2.6.1 Introducción

La transformación de modelos consiste en obtener un nuevo modelo mediante la transformación de un modelo existente. En dicho contexto, los modelos son elementos de primer orden, apareciendo entonces, la noción de metamodelos y transformaciones.

Un modelo debe ser definido de acuerdo a la semántica de su metamodelo: un modelo se dice que conforma a su metamodelo. De la misma manera, un metamodelo debe conformar a su metamodelo. En esta arquitectura de tres capas (modelos, metamodelos y metamodelos), el metamodelo usualmente conforma a su propia semántica, es decir, éste puede ser definido utilizando sus propios conceptos.

Concepto de metamodelo y Metamodelo según la OMG:

- 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.

La siguiente imagen permite visualizar dicha arquitectura:

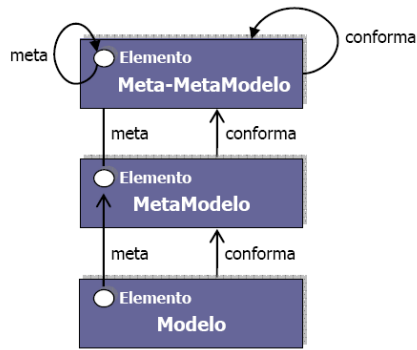


Imagen 3 - MetaModelo y MetaMetaModelo según las OMG

El considerar los modelos como entidades de primer orden requiere contar con un conjunto de herramientas para definir operaciones sobre los mismos. En este sentido, la transformación de modelos parece ser la de mayor utilidad. Una transformación de modelos entonces provee facilidades para generar un modelo Mb, conforme a un metamodelo MMB, desde un modelo Ma conforme a un metamodelo MMA.

La característica más importante es lograr que todos los elementos involucrados sean considerados como modelos. Luego, la transformación de modelos puede ser definida como un modelo, debiendo conformar a un metamodelo que define la semántica del modelo de transformación. Y como otro metamodelo, el metamodelo de la transformación conforma a un metametamodelo.

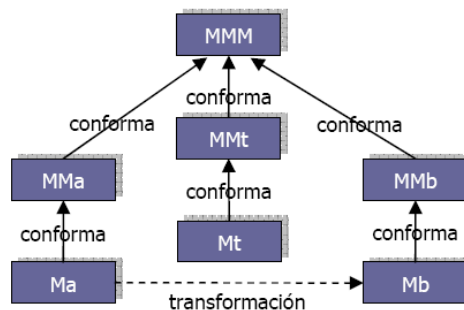
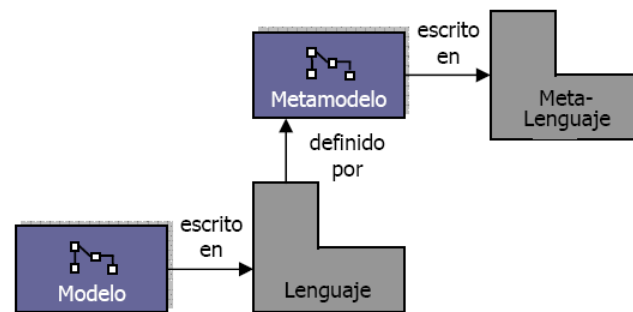


Imagen 4 - Proceso completo de transformación de modelos

La imagen anterior resume el proceso completo de transformación de modelos. Un modelo Ma, conforme a un metamodelo MMA, es transformado al modelo Mb conforme a un metamodelo MMB. La transformación es definida por el modelo de transformación de modelos

Mt el cual a su vez conforma al metamodelo de transformación MMt. Éste último metamodelo, junto con los metamodelos MMA y MMB, deben conformar a un metametamodelo.

La OMG [11] ha definido una arquitectura de cuatro capas de modelado mediante la cual es posible definir los lenguajes de modelado y sus respectivos modelos. Mediante un lenguaje de modelado es posible crear modelos; un modelo especifica los elementos que pueden tener lugar en un modelo real. Por ejemplo, al definirse la clase Persona en un modelo, luego se podrán tener instancias de Persona como Juan, Pedro, etc. Por otro lado, la definición de un lenguaje de modelado establece que elementos pueden existir en un modelo. Por ejemplo, el lenguaje UML establece que dentro de un modelo se pueden usar los conceptos Clase, Atributo, Asociación, Paquete, etc. Debido a esta similitud, se puede describir un lenguaje por medio de un modelo, usualmente llamado “metamodelo”. El metamodelo describe que elementos pueden ser usados en el lenguaje. Como un metamodelo es también un modelo, el metamodelo en sí mismo debe estar escrito en un lenguaje bien definido. Este lenguaje se llama metalenguaje.



Modelos, Lenguajes, Metamodelos y Metalenguajes

Imagen 5 - Modelos, Lenguajes, Metamodelos y Metalenguajes

El metamodelo describe la sintaxis abstracta del lenguaje. Esta sintaxis es la base para el procesamiento automatizado (basado en herramientas) de los modelos. Por otra parte, la sintaxis concreta es definida mediante otros mecanismos y no es relevante para las herramientas de transformación de modelos. La sintaxis concreta es la interfaz para el modelador e influye fuertemente en el grado de legibilidad de los modelos. Como consecuencia de este desacoplamiento, el metamodelo y la sintaxis concreta de un lenguaje pueden mantener una relación 1:n, es decir que la misma sintaxis abstracta (definida por el metamodelo) puede ser visualizada a través de diferentes sintaxis concretas. Inclusive un mismo lenguaje puede tener una sintaxis concreta gráfica y otra textual.

2.6.2 Arquitectura 4 capas

El metamodelado es entonces un mecanismo que permite definir formalmente lenguajes de modelado, como por ejemplo UML. La Arquitectura de cuatro capas de modelado es la

propuesta del OMG orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos.

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

- **Nivel M0: Instancias**

En el nivel M0 se encuentran todas las instancias “reales” del sistema, es decir, los objetos de la aplicación.

- **Nivel M1: Modelo del sistema**

Por encima de la capa M0 se sitúa la capa M1, que representa el modelo de un sistema de software. Los conceptos del nivel M1 representan categorías de las instancias de M0. Es decir, cada elemento de M0 es una instancia de un elemento de M1. Sus elementos son modelos de los datos. El modelo del usuario contiene los elementos del modelo y los snapshots de instancias de dichos elementos.

- **Nivel M2: Metamodelo**

Análogamente a lo que ocurre con las capas M0 y M1, los elementos del nivel M1 son a su vez instancias del nivel M2. Esta capa recibe el nombre de metamodelo. UML y OCL son ejemplos de metamodelos.

- **Nivel M3: Meta-Metamodelo**

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

M3 es el nivel más abstracto, que permite definir metamodelos concretos. Dentro del OMG, MOF es el lenguaje estándar de la capa M3. Esto supone que todos los metamodelos de la capa M2, son instancias de MOF. No existe otro meta nivel por encima de MOF. Básicamente, el MOF se define a sí mismo.

Razones por las que el metamodelado es importante:

- Permite contar con un mecanismo para definir lenguajes de modelado sin ambigüedades y que una herramienta de transformación pueda leer, escribir y entender los modelos;
- Las reglas de transformación que constituyen una definición de una transformación describen como un modelo en un lenguaje fuente puede ser transformado a un modelo en un lenguaje destino. Estas reglas usan los metamodelos de los lenguajes fuente y destino para definir la transformación;

- La sintaxis de los lenguajes en los cuales se expresan las reglas de transformación también debe estar formalmente definida para permitir su automatización. En este caso también se utilizará la técnica de metamodelado para especificar su sintaxis.

2.6.3 MOF (Infraestructura 2.0)

El lenguaje MOF, acrónimo de Meta-Object Facility, es un estándar del OMG para el desarrollo conducido por modelos.

Como se vio anteriormente, MOF se encuentra en la capa superior de la arquitectura de 4 capas. Provee un metamodelenguaje que permite definir metamodelos en la capa M2. El ejemplo más popular de un elemento en la capa M2 es el metamodelo UML, que describe al lenguaje UML.

Esta es una arquitectura de metamodelado cerrada y estricta. Es cerrada porque el metamodelo de MOF se define en términos de sí mismo. Y es estricta porque cada elemento de un modelo en cualquiera de las capas tiene una correspondencia estricta con un elemento del modelo de la capa superior.

Tal como su nombre lo indica, MOF se basa en el paradigma de Orientación a Objetos. Por este motivo usa los mismos conceptos y la misma sintaxis concreta que los diagramas de clases de UML.

Actualmente, la definición de MOF está separada en dos partes fundamentales, EMOF (Essential MOF) y CMOF (Complete MOF), y se espera que en el futuro se agregue SMOF (Semantic MOF). Ambos paquetes importan los elementos de un paquete en común, del cual utilizan los constructores básicos y lo extienden con los elementos necesarios para definir metamodelos simples, en el caso de EMOF y metamodelos más sofisticados, en el caso de CMOF.

Ejemplo de modelo de 4 capas

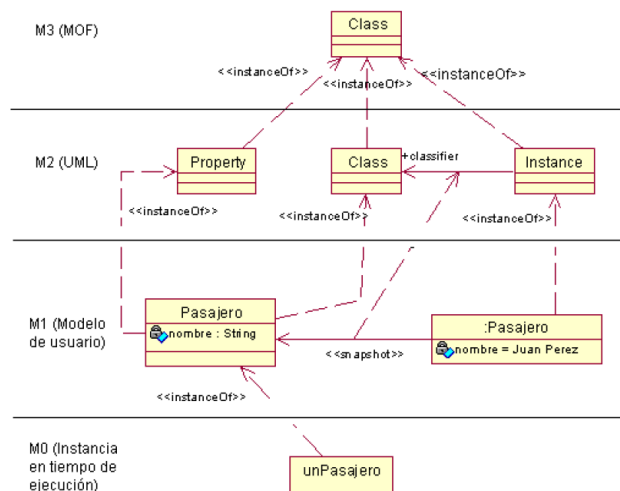


Imagen 6 - Modelo de 4 capas

Sintaxis Abstracta MoF

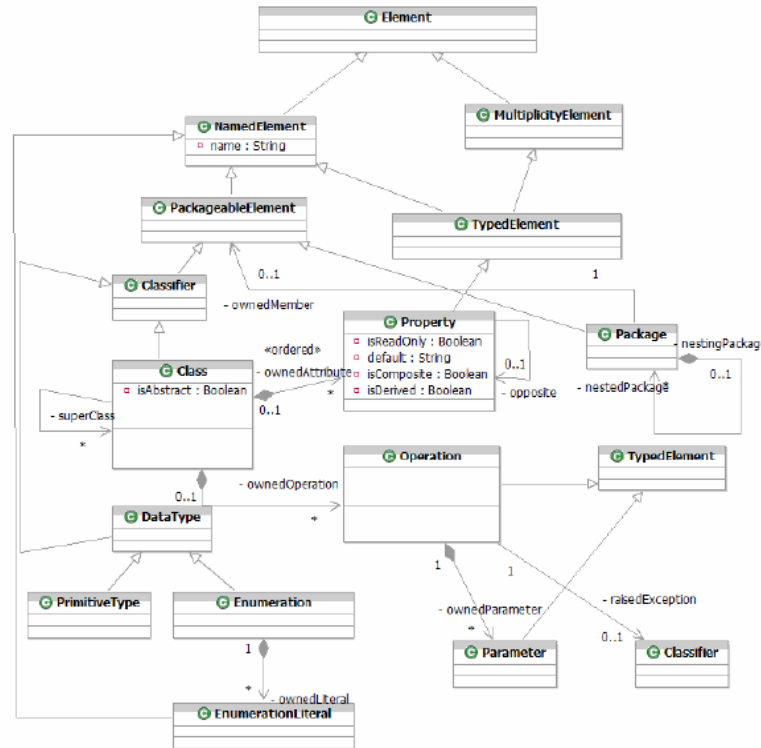


Imagen 7 - Sintaxis Abstracta MoF

- **Implementación en eclipse(ECORE/EMF)**

Eclipse es principalmente una comunidad que fomenta el código abierto. Los proyectos en los que trabaja se enfocan principalmente en construir una plataforma de desarrollo abierta. Esta plataforma puede ejecutarse en múltiples sistemas operativos, lo que convierte a Eclipse en una multi-plataforma. Está compuesta por frameworks extensibles, y otras herramientas que permiten construir y administrar software. Permite que los usuarios colaboren para mejorar la plataforma existente y extiendan su funcionalidad a través de la creación de plugins.

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas VisualAge. Inicialmente fue un entorno de desarrollo para JAVA, escrito en JAVA. Pero luego fue extendiendo el soporte a otros lenguajes de programación. Eclipse ganó popularidad debido a que la plataforma Eclipse proporciona el código fuente de la plataforma y esta transparencia generó confianza en los usuarios.

Hace algunos años se creó la Fundación Eclipse, una organización independiente sin fines de lucro que fomenta una comunidad de código abierto. La misma trabaja sobre un conjunto de

productos complementarios, capacidades y servicios que respalda y se encarga del desarrollo continuo de la plataforma. La comunidad Eclipse trabaja actualmente en 60 proyectos. En este capítulo solamente pondremos foco en el proyecto dedicado a promover tecnologías de desarrollo basadas en modelos llamado Eclipse Modeling Project[28] (<http://www.eclipse.org/modeling/>). Este proyecto está compuesto por otros subproyectos relacionados.

Subproyectos de Eclipse Modeling Project:

- Abstract Syntax development (EMF)
- Concrete Syntax development (GMF)
- Model Transformation (QVT, JET, MOF2Text)
- Standards Implementations (UML2, OCL, OMD, XSD)
- Technology & Research (GMT, MDDi, Query, Transaction, etc)

○ **ECORE**

El metamodelo MOF está implementado mediante un plugin para Eclipse llamado Ecore [19]. Este plugin respeta las metaclases definidas por MOF. Todas las metaclases mantienen el nombre del elemento que implementan y agregan como prefijo la letra “E”, indicando que pertenecen al metamodelo Ecore. Por ejemplo, la metaclase EClass implementa a la metaclase Class de MOF. La primera implementación de Ecore fue terminada en Junio del 2002. Esta primera versión usaba como meta-metamodelo la definición estándar de MOF (v1.4). Gracias a las sucesivas implementaciones de Ecore y basado en la experiencia ganada con el uso de esta implementación en varias herramientas, el metalenguaje evolucionó. Como conclusión, se logró una implementación de Ecore realizada en JAVA, más eficiente y con sólo un subconjunto de MOF, y no con todos los elementos como manipulaban las implementaciones hechas hasta ese momento. A partir de este conocimiento, el grupo de desarrollo de Ecore colaboró más activamente con la nueva definición de MOF, y se estableció un subconjunto de elementos como esenciales, llegando a la definición de EMOF, que fue incluida como parte del MOF (v2.0).

MOF y Ecore son conceptualmente muy similares, ambos basados en el concepto de clases con atributos tipados y operaciones con parámetros y excepciones. Además ambos soportan herencia múltiple y utilizan paquetes como mecanismo de agrupamiento. MOF puede ser usado para definir metamodelos de lenguajes orientados a objetos, como es el caso de UML, y también para otros lenguajes no orientados a objetos, como es el caso de las redes de Petri o los lenguajes para servicios web.

○ **EMF**

El proyecto EMF es un framework para modelado que permite la generación automática de código para construir herramientas y otras aplicaciones a partir de modelos de datos estructurados.

EMF comenzó como una implementación del metalenguaje MOF. En los últimos años se usó para implementar una gran cantidad de herramientas lo que permitió mejorar la eficiencia del código generado. Actualmente el uso de EMF para desarrollar herramientas está muy extendido. Se usa, por ejemplo, para implementar XML Schema Infoset Modelo (XSD), Servicio de Data Objects (SDO), UML2, y Web Tools Platform (WTP) para los proyectos Eclipse. Además EMF se utiliza en productos comerciales, como Omondo, EclipseUML, IBM Rational y productos WebSphere.

EMF permite usar un modelo como el punto de partida para la generación de código, e iterativamente refinar el modelo y regenerar el código, hasta obtener el código requerido. Aunque también prevé la posibilidad de que el programador necesite modificar ese código, es decir, se contempla la posibilidad de que el usuario edite las clases generadas, para agregar o editar métodos y variables de instancia. Siempre se puede regenerar desde el modelo cuando se necesite, y las partes agregadas serán preservadas durante la regeneración.

3 Lenguajes de Transformación

En este capítulo se detallará la definición de transformación de modelos y los conceptos relacionados a esta definición, a través de la descripción de los elementos principales del lenguaje para transformaciones QVT. Además se menciona una breve descripción del lenguaje de transformación ATL y por último se detalla el lenguaje de programación, complementario a los lenguajes de transformaciones, OCL.

3.1 Que es una transformación

Las herramientas de transformación toman un modelo como entrada y producen un modelo como salida.

Cuando miramos dentro de una herramienta de transformación, podemos ver qué elementos están involucrados para realizar la transformación. En algún lugar dentro de la herramienta hay una definición que describe cómo un modelo debería ser transformado. A esta definición la llamamos definición de la transformación.

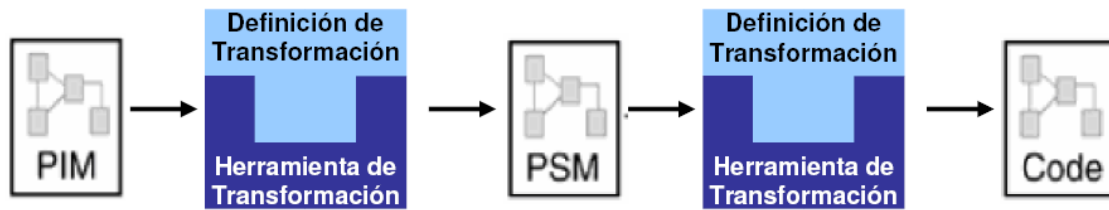


Imagen 8 - Estructura de la herramienta de transformación "abierta".

Aquí vemos que hay una diferencia entre la transformación en sí misma, es decir el proceso de generación de un nuevo modelo desde otro, y la definición de la transformación.

La herramienta de transformación usa la misma definición de transformación para cada transformación de cualquier modelo de entrada correspondiente al lenguaje fuente sobre el que se aplique, construyendo un modelo correspondiente al lenguaje destino. Podemos, por ejemplo, especificar una definición de transformación desde UML a Java, la cual describe como debería ser generado Java para un (o cualquier) modelo UML.

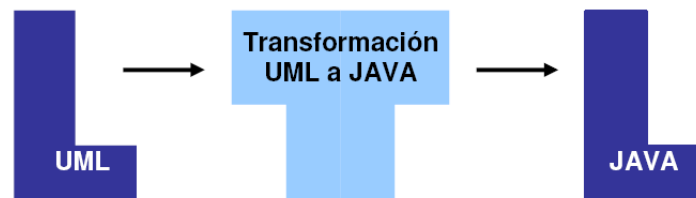


Imagen 9 - Transformación UML a Java

En general podemos decir que una definición de transformación consiste de una colección de reglas de transformación, que son especificaciones no-ambiguas de la manera que un modelo (o parte de él) puede ser usado para crear otro modelo (o parte de él). Basándonos en estas observaciones, podemos definir, según Kleppe [8]:

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

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

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

3.2 Que es un lenguaje de transformación

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

3.3 QVT

El lenguaje QVT (Query/View/Transformation) es el lenguaje estándar de OMG para transformaciones. Los conceptos, definiciones y ejemplos que siguen están extraídos del manual de especificación MOF 2.0 Query/View/Transformation [20]

Los requerimientos impuestos por la OMG para un lenguaje estándar para transformaciones, justifican el acrónimo Q/V/T:

- Query: el lenguaje debe facilitar consultas ad-hoc para selección y filtrado de elementos de modelos. Generalmente se seleccionan elementos de modelos que son la fuente de entrada (inputs) de una transformación.
- View: el lenguaje debe permitir la creación de vistas de metamodelos MOF, sobre los que se define la transformación.
- Transformation: el lenguaje debe permitir la definición de transformaciones, a través de relaciones entre un metamodelo MOF fuente F, y un metamodelo MOF destino D, el cual es usado para generar un modelo destino, instancia que conforma a D, desde un modelo fuente, instancia que conforma a F.

Las siguientes sub-secciones describen una visión general de QVT y sus componentes principales.

3.3.1 Descripción General de QVT

La especificación de QVT tiene una naturaleza híbrida, declarativa/imperativa, con la parte declarativa dividida en una arquitectura de dos niveles. Esta especificación define tres paquetes principales, uno por cada lenguaje definido: QVTCORE, QVTRelation y QVTOperational, como puede observarse en la imagen 10. Estos paquetes principales se comunican entre sí y comparten otros paquetes intermedios (imagen 11):

El paquete QVTBase define estructuras comunes para transformaciones. El paquete QVTRelation usa expresiones de patrones template definidas en el paquete QVTTemplateExp. El paquete QVTOperational extiende al QVTRelation, dado que usa el mismo framework para trazas. Usa también las expresiones imperativas definidas en el paquete ImperativeOCL. Todos los paquetes QVT dependen del paquete EssentialOCL de OCL 2.0 [21], y todos los paquetes del lenguaje dependen de EMOF (EssentialMOF) [12].

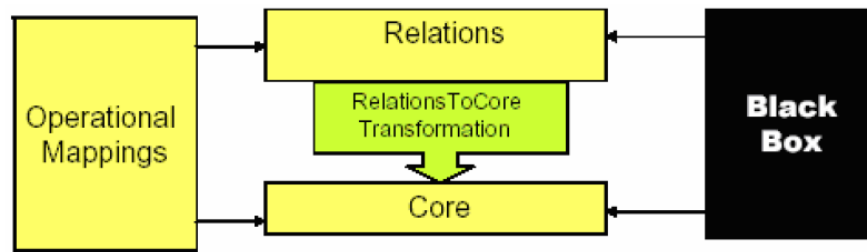


Imagen 10 - Relaciones entre metamodelos QVT

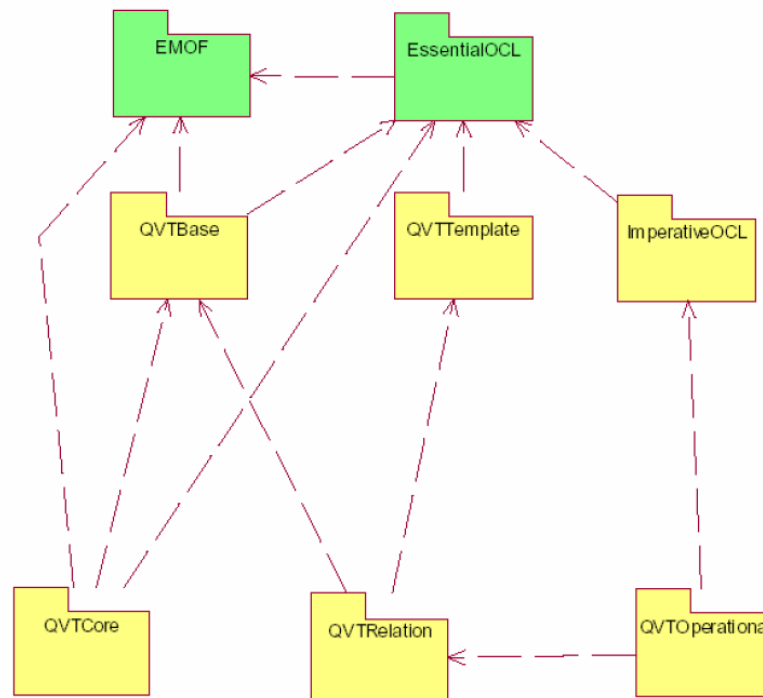


Imagen 11 - Dependencias de paquetes en la especificación QVT

En las siguientes secciones continuamos explicando la arquitectura de la parte declarativa y operacional del lenguaje.

3.3.2 QVT Declarativo

La parte declarativa de QVT está dividida en una arquitectura de dos niveles. Las capas son:

- Un metamodelo y lenguaje Relations amigable para el usuario, que soporta pattern matching complejo de objetos y creación de template para objetos. Las

trazas entre elementos del modelo involucrados en una transformación se crean implícitamente.

- Un metamodelo y lenguaje Core definido usando extensiones de EMOF y OCL. Las clases traza se definen explícitamente como modelos MOF, y pueden crearse y borrarse como cualquier otro objeto.

○ **Lenguaje Relations**

Es una especificación declarativa de las relaciones entre modelos MOF. Las relaciones pueden pedir que otras relaciones también se establezcan entre elementos particulares del modelo, que cumplen con los pattern matching. En este lenguaje, una transformación entre modelos se especifica como un conjunto de relaciones que deben establecerse para que la transformación sea exitosa.

Los modelos tienen nombre, y los elementos que contienen deben ser de tipos correspondientes al metamodelo que referencian.

Un ejemplo:

transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) { ... En esta declaración llamada "umlRdbms," hay dos modelos tipados: "uml" y "rdbms". El modelo llamado "uml" declara al paquete SimpleUML como su metamodelo y el modelo "rdbms" declara al paquete SimpleRDBMS como su metamodelo. Una transformación puede ser invocada para chequear consistencia entre dos modelos o para modificar un modelo forzando consistencia. Cuando se fuerza consistencia, se elige el modelo destino; puede estar vacío o contener elementos a ser relacionados por la transformación. Los elementos que no existan serán creados para forzar el cumplimiento de la relación.

○ **Relaciones, Dominios y Pattern Matching**

Las relaciones en una transformación declaran restricciones que deben satisfacer los elementos de los modelos. Una relación se define por dos o más dominios y un par de predicados when y where, que deben cumplirse entre los elementos de los modelos. Un dominio es una variable con tipo que puede tener su correspondencia en un modelo de un tipo dado. Un dominio tiene un patrón, que se puede ver como grafo de nodos de objetos. Un patrón se puede ver alternativamente como un conjunto de variables y un conjunto de restricciones que los elementos del modelo asocian a aquellas variables que lo satisfacen (pattern matching). Un patrón del dominio se puede considerar un template para los objetos y sus propiedades que deben ser seteadas, modificadas o creadas en el modelo para satisfacer la relación. En el ejemplo que se muestra debajo, se declaran dos dominios que harán correspondencia entre elementos de los modelos "uml" y "rdbms" respectivamente. Cada dominio especifica un patrón simple: un paquete con un nombre, y un esquema con un nombre, en ambos la propiedad "name" asociada a la misma variable "pn" implica que deben tener el mismo valor:

```

relation PackageToSchema /* map each package to a schema */
{
    domain uml p:Package {name=pn}
    domain rdbms s:Schema {name=pn}
}

relation ClassToTable /* map each persistent class to a table */
{
    domain uml c:Class { namespace = p:Package {}, kind='Persistent', name=cn}
    domain rdbms t:Table { schema = s:Schema {}, name=cn, column = cl:Column {
        name=cn+'_tid', type='NUMBER'}, primaryKey = k:PrimaryKey { name=cn+'_pk',
        column=cl }}
    when { PackageToSchema(p, s);}
    where { AttributeToColumn(c, t); }
}

```

○ Relaciones top-Level

Una transformación contiene dos tipos de relaciones: `topLevel` y `no topLevel`. La ejecución de una transformación requiere que se puedan aplicar todas sus relaciones `topLevel`, mientras que las `no topLevel` se deben cumplir solamente cuando son invocadas, directamente o a través de una cláusula `where` de otra relación:

```

transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS)
{
    top relation PackageToSchema {...}
    top relation ClassToTable {...}
    relation AttributeToColumn {...}
}

```

Una relación `topLevel` tiene la palabra `top` para distinguirla sintácticamente. En el ejemplo de arriba, `PackageToSchema` y `ClassToTable` son relaciones `topLevel`, mientras que `AttributeToColumn` es una relación `no topLevel`.

○ Claves y Creación de Objetos usando Patrones

Como ya mencionamos, una expresión `object template` provee un `template` para crear un objeto en el modelo destino. Cuando para un patrón válido en el dominio de entrada no existe el correspondiente elemento en la salida, entonces la expresión `object template` es usada como `template` para crear objetos en el modelo destino.

Por ejemplo, cuando `ClassToTable` se ejecuta con el modelo fuente “`rdbms`”, la siguiente expresión `object template` sirve como `template` para crear objetos en el modelo destino “`rdbms`”:

```

t:Table {

```

```

schema = s:Schema {},
name = cn,
column = cl:Column {name=cn+'_tid', type='NUMBER'},
primaryKey = k:PrimaryKey {name=cn+'_pk', column=cl}
}

```

El template asociado con Table especifica que un objeto tabla debe ser creado con las propiedades “schema”, “name”, “column” y “primaryKey” con los valores especificados en la expresión template. Similarmente, los templates asociados con Column, PrimaryKey, etc, especifican cómo sus respectivos objetos deben ser creados.

3.3.3 QVT Operacional

Además de su lenguajes declarativos, QVT proporciona dos mecanismos más para implementaciones de transformaciones: un lenguaje estándar, Operational Mappings, e implementaciones no-estándar Black-box. El lenguaje Operational Mappings se detallará en el apartado siguiente.

El mecanismo caja negra (black-box), permite implementaciones opacas (escritas en otro lenguaje existente) de parte de transformaciones. Esto puede ser peligroso si la implementación tiene acceso a referencias de objetos en los modelos, pudiendo manipularlos arbitrariamente. Una implementación caja negra no tiene relación implícita con el lenguaje Relations, pero cada caja negra debería implementar una Relation, la cual es responsable de mantener las trazas entre los elementos relacionados.

○ Lenguaje Operacional

Este lenguaje se especificó como una forma estándar para proveer implementaciones imperativas. Proporciona extensiones OCL con efectos laterales que permiten un estilo más procedural y una sintaxis que resulta familiar a los programadores.

Las operaciones de mapeo (Mappings) pueden ser usadas para implementar una o más relaciones de una especificación del lenguaje de relaciones. Una transformación escrita usando solamente operaciones de mapeo es llamada transformación.

3.4 ATL

ATL (ATLAS Transformation Language) [22] es un lenguaje de transformación de modelos y un conjunto de herramientas desarrolladas por el Grupo ATLAS (INRIA & LINA). En el campo de Model-Driven Engineering (MDE), ATL provee formas de producir un conjunto de modelos destino, desde un conjunto de modelos fuente.

Desarrollado sobre la plataforma Eclipse, el ATLIntegrated Enviroment (IDE) provee un número de herramientas estándar de desarrollo (syntax highlighting, debugger, etc.) que facilita el

desarrollo de transformaciones en ATL. El proyecto ATL incluye también una librería de transformaciones ATL.

ATL soporta tres tipos de unidades: Módulos (Module), Consultas (Query) y Librerías (Library). Los mismos podrán contener funciones auxiliares (helpers), atributos y reglas (matched and called).

3.5 OCL

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

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

Un diagrama UML, como por ejemplo un diagrama de clases, en general no está lo suficientemente refinado para proveer todos los aspectos relevantes de una especificación. Existe, entre otras cosas, una necesidad de describir restricciones adicionales sobre objetos del modelo. Esas restricciones son a veces descriptas en lenguaje natural, pero en la práctica esto resulta siempre en ambigüedades.

Como consecuencia, para escribir restricciones que no resulten ambiguas se han desarrollado los lenguajes formales, los cuales tienen la desventaja de tener que ser utilizados por personas con un gran conocimiento matemático, lo cual los hace difíciles de usar por los modeladores de sistemas.

OCL fue desarrollado para solucionar este problema. Es un lenguaje formal y al mismo tiempo es fácil de leer y escribir. Es un lenguaje puro de especificación, el cual garantiza estar libre de efectos laterales. Cuando se evalúa una expresión OCL, simplemente retorna un valor, sin hacer modificaciones en el modelo. OCL no es un lenguaje de programación, es decir, no es posible escribir la lógica de un programa o flujos de control en OCL. No se pueden invocar procesos o activar operaciones que no sean de consulta con OCL.

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

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

4 Herramientas de transformación de modelos más usadas

La versión 1.0 de la especificación de la OMG[11] fue publicada en Abril del año 2008, por lo que las soluciones en base a la especificación se encuentran en un estado naciente.

Existen diversos enfoques que contribuyen brindando soluciones para implementar transformaciones de modelos: lenguajes, frameworks, etc. Pero que en su mayoría no son soluciones compatibles con la especificación de QVT. Uno de los proyectos más representativos en éste campo es ATL [22]. Actualmente, este lenguaje quedó fuera de la especificación.

A continuación se detallan las principales herramientas de transformación de modelos.

4.1 Viatra (Visual Automated model Transformations)

Viatra es una herramienta para la transformación de modelos que actualmente forma parte del framework VIATRA2 [43], implementado en lenguaje Java y se encuentra integrado en Eclipse.

Provee un lenguaje textual para describir modelos y metamodelos, y transformaciones llamados VTML y VTCL respectivamente. La naturaleza del lenguaje es declarativa y está basada en técnicas de descripción de patrones, sin embargo es posible utilizar secciones de código imperativo. Se apoya en métodos formales como la transformación de grafos (GT) y la máquina de estados abstractos (ASM) para ser capaz de manipular modelos y realizar tareas de verificación, validación y seguridad, así como una temprana evaluación de características no funcionales como fiabilidad, disponibilidad y productividad del sistema bajo diseño.

Como puntos débiles podemos resaltar que Viatra no se basa en los estándares MOF ni QVT. No obstante, pretende soportarlos en un futuro mediante mecanismos de importación y exportación integrados en el framework.

4.2 Tefkat

Tefkat [45] es un lenguaje de transformación de modelos y un motor para la transformación de modelos. El lenguaje está basado en F-logic y la teoría de programas estratificados de la lógica. El motor es un plugin para Eclipse.

Tefkat fue uno de los subproyectos del proyecto Pegamento, desarrollado en Distributed Systems Technology Centre DSTC) de Australia.

Esta implementado como un plugin de Eclipse y usa EMF para manejar los modelos basados en MOF, UML2 y esquemas XML.

Tefkat define un mapeo entre un conjunto de metamodelos origen en un conjunto de metamodelos destino. Una transformación tefkat consiste de reglas, patterns y templates. Las

reglas contienen un término origen y un término destino. Los patterns son términos origen compuestos agrupados bajo un nombre, y los templates son términos destino compuestos agrupados bajo un nombre. Estos elementos están basados en la F-logic y la programación pura de la lógica, sin embargo, la ausencia de símbolos de función significa una reducción importante en la complejidad.

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

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

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

4.3 ATL (Atlas Transformation Language)

ATL [39] es un proyecto creado por gente de la Université de Nantes, Faculté des Sciences et Techniques, en Francia. ATL forma parte del framework de gestión de modelos AMMA que se encuentra integrado en Eclipse y EMF. Utiliza un lenguaje propietario llamado ATLAS [41] para definir transformaciones que se ejecuta sobre JAVA para el cual proporciona un editor.

La naturaleza de ATLAS es declarativa, aunque lleva mucho tiempo utilizando también construcciones imperativas, y las transformaciones son expresadas como reglas de transformación ya que ATL cumple el estándar MOF.

ATL posee un algoritmo de ejecución preciso y determinista, sin embargo no se apoya sobre ningún método formal. No proporciona mecanismos para la validación de las transformaciones. Para llevar a cabo transformaciones compuestas se necesita ejecutar cada una de las transformaciones participantes una a una.

Aunque la sintaxis de ATL es muy similar a la de QVT, no es interoperable con este último.

4.4 Epsilon

Epsilon [40] es una plataforma desarrollada como un conjunto de plug-ins (editores, asistentes, pantallas de configuración, etc.) sobre Eclipse. Presenta el lenguaje metamodelo independiente Epsilon Object Language que se basa en OCL. Puede ser utilizado como lenguaje de gestión de modelos o como infraestructura a extender con nuevos lenguajes específicos de dominio. Tres son los lenguajes definidos en la actualidad:

Epsilon Comparison Language (ECL), Epsilon Merging Language (EML), Epsilon Transformation Language (ETL), para comparación, composición y transformación de modelos

respectivamente. Se da soporte completo al estándar MOF mediante modelos EMF y documentos XML a través de JDOM.

La finalidad pretendida con la extensión del lenguaje OCL es dar soporte al acceso de múltiple modelos, ofrecer constructores de programación convencional adicionales (agrupación y secuencia de sentencias), permitir modificación de modelos, proveer depuración e informe de errores, así como conseguir una mayor uniformidad en la invocación.

Soporta mecanismos de herencia, traceability e introduce mecanismos de comprobación automática del resultado en composición y transformación de modelos.

4.5 AToM (A Tool for Multi-formalism and Meta-Modeling)

AToM [46] es una herramienta para modelado en muchos paradigmas bajo el desarrollo de MSDL (Modelling Simulation and Design Lab) en la escuela de ciencias de la computación de la universidad de McGill.

Las dos tareas principales de AToM3 son metamodelado y transformación de modelos. El metamodelado se refiere a la descripción o modelado de diferentes clases de formalismos usados para modelar sistemas (aunque se enfoca en formalismos de simulación y sistemas dinámicos, las capacidades de AToM3 no se restringen a solo esos). Transformación de modelos se refiere al proceso automático de convertir, traducir o modificar un modelo en un formalismo dado en otro modelo que puede o no estar descrito en el mismo formalismo.

En AToM3 los formalismos y modelos están descriptos como grafos, para realizar una descripción precisa y operativa de las transformaciones entre modelos.

Desde una meta especificación (en un formalismo de entidad relación), AToM3 genera una herramienta para manipular visualmente (crear y editar) modelos descriptos en el formalismo especificado. Las transformaciones de modelos están ejecutadas por la reescritura de grafos. Las transformaciones pueden entonces ser expresadas declarativamente como modelos basados en grafos.

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

4.6 MOLA (Model Transformation Language)

El proyecto Mola [47] consiste de un lenguaje de transformación de modelos y de una herramienta para la definición y ejecución de transformaciones.

El objetivo del proyecto Mola es proveer un lenguaje gráfico para definir las transformaciones entre modelos que sean simple y fácilmente entendibles.

El lenguaje para la definición de la transformación Mola es un lenguaje gráfico, basado en conceptos tradicionales como pattern matching y reglas que definen como los elementos deben ser transformados. El orden en el cual se aplican las reglas es el orden tradicional de los constructores de programación (secuencia, loop y branching).

Los procedimientos Mola definen la parte ejecutable de la transformación.

La unidad principal ejecutable es la regla que contiene un pattern y acciones. Un procedimiento está construido con reglas usando constructores de la programación estructural tradicional, es decir, loops, branchings y llamadas a procedimientos, todos definidos de una manera grafica.

La parte ejecutable es similar a los diagramas de actividad de UML, que comienzan en un nodo inicio y contiene

- **loop symbols (Foreach and While)**
- **rules**
- **text statements**
- **procedure calls (including external ones)**
- **end nodes**

Mola usa una manera simple para definir metamodelos: Diagramas de clases UML, los cuales consisten solo en un subconjunto de los elementos de UML, es decir, clases, asociaciones, generalizaciones y enumerativos.

Solamente soporta herencia simple. Actualmente el metamodelo completo (el metamodelo fuente y el metamodelo destino) deben estar en el mismo diagrama de clases. Adicionalmente se le agregan asociaciones para mapear los elementos del metamodelo fuente en el destino.

4.7 MOFScript

La herramienta MOFScript [44] permite la transformación de cualquier modelo MOF a texto. Es decir, permite la generación de código Java, EJB, JSP, C#, SQL Scripts, HTML o documentación a partir de los modelos. La herramienta está desarrollada como un plugin de Eclipse, el cual soporta el parseo, chequeo y ejecución de scripts escritos en MOFScript (es el lenguaje de transformación basado en QVT)

Algunas características de MOFScript son:

- Generación de texto a partir de modelos basados en MOF. La habilidad de generar texto a partir de cualquier modelo basado en un metamodelo definido en MOF, como por ejemplo, modelos UML o cualquier otro metamodelo.

- Mecanismos de control: la habilidad de especificar mecanismos de control básicos como loops y sentencias condicionales.
- Manipulación de strings con operaciones básicas.
- Permite producir recursos (archivos) como salida. Permite especificar el archivo salida para la generación de texto.
- Permite hacer traceability entre los modelos y el texto generado. Esta propiedad permite hacer una regeneración.
- El editor de scripts posee ayuda para completar el código.
- La ingeniería inversa todavía no es parte de la herramienta.

Los archivos que definen la transformación tienen como extensión “.m2t”. El lenguaje de transformación MOFScript está basado en el lenguaje QVT. MOFScript es un refinamiento del lenguaje operacional QVT. Es un lenguaje textual, basado en objetos y usa OCL para la navegación de los atributos. Además, presenta algunas características avanzadas, como la jerarquía de transformaciones, traceabilidad, etc.

4.8 Kermeta

El lenguaje Kermeta [23] fue desarrollado por un equipo de investigación de IRISA (investigadores de INRIA, CNRS, INSA y la Universidad Rennes).

El nombre Kermeta es una abreviación para "Kernel Metamodeling" y refleja el hecho que el lenguaje fue pensado como una parte fundamental para el metamodelado. La herramienta que ejecuta las transformaciones Kermeta está desarrollada en Eclipse, bajo licencia EPL.

Kermeta es un lenguaje de modelado y de programación. Su metamodelo conforma el estándar EMOF. Fue diseñado para escribir modelos, para escribir transformaciones entre modelos y para escribir restricciones sobre estos modelos y ejecutarlos.

El objetivo de esta propuesta es brindar un nivel de abstracción adicional sobre el nivel de objetos, y de esta manera ver un sistema dado como un conjunto de conceptos (e instancias de conceptos) que forman un todo coherente que se puede llamar modelo.

Kermeta ofrece todos los conceptos de EMOF usados para la especificación de un modelo. Un concepto real de modelo, más precisamente de tipo de modelo, y una sintaxis concreta que encaja bien con la notación de modelo y metamodelo.

- Kermeta ofrece dos posibilidades para escribir un metamodelo:

Escribir el metamodelo con Omondo2, e importarlo

- Escribir el metamodelo en Kermeta y traducirlo a un archivo.ecore usando la función “kermeta2ecore”

4.9 Kent Model Transformation language

El lenguaje de transformación de modelos Kent [48][49] es la tercera generación de propuestas de transformación de modelos desarrolladas en la Universidad de Kent. El lenguaje intenta ser un lenguaje de especificación declarativa, con la opción de proveer partes constructivas si es requerido. La semántica del lenguaje va desde una base formal de relaciones

La semántica del lenguaje de transformación está basada en el lenguaje Relations de QVT.

Los metamodelos son importados desde Poseidon, un editor de UML, donde fueron creados como diagramas de clases.

4.10 Medini by ikv++ Technologies

Medini [24] es una herramienta Open Source construida por ikv++ technologies sobre las bases del proyecto OSLO [25]. OSLO (Open Source Libraries for OCL) es un proyecto basado en la implementación OCL de la Universidad de Kent que provee las clases del metamodelo de OCL.

Medini es un conjunto de herramientas construidas para el diseño y ejecución de transformaciones declarativas de modelos. Define una estructura de clases que extienden a las librerías de OSLO para modelar transformaciones QVT. Pero además, Medini implementa un motor capaz de ejecutar código QVT declarativo, siendo una especial excepción entre las herramientas de transformación de modelos.

4.11 SmartQVT by Telecom France

SmartQVT [42] es un proyecto que nació en el departamento de I+D de France Telecom.

Es un compilador Open Source de transformaciones de modelos escritas en QVT operacional. En esencia, toma como input una transformación escrita en lenguaje QVT operacional y obtiene a partir de ella, una clase Java que implementa comportamiento descrito en lenguaje QVT. Esta última, como toda clase Java, puede ser integrada en cualquier aplicación y ejecutada desde cualquier máquina virtual.

Para realizar su trabajo, la herramienta implementa un parser y un compiler. El parser lee código QVT y obtiene a partir de él, su metamodelo. El compiler toma el metamodelo y escribe el texto del código fuente de la clase Java.

Pero este parser tiene una característica que lo hace muy particular:

La instanciación del metamodelo es un proceso que se realiza en dos partes: primero se leen los tokens recibidos por el scanner y se generan las clases necesarias para construir el árbol sintáctico que representa al código leído. Segundo, es necesario usar los elementos y la estructura del mencionado árbol para instanciar los objetos del metamodelo de QVT. Esta última tarea es vista como una transformación de modelos, donde el modelo de entrada es el árbol sintáctico producido por el parser y el modelo de salida es el metamodelo QVT. El equipo de France Telecom escribió entonces una transformación utilizando el lenguaje QVT declarativo que se describe en la especificación. Dicha transformación toma el modelo de un árbol sintáctico generado a partir de un código QVT (que respete la gramática de QVT operacional) y genera un modelo de salida que no es otra cosa que el metamodelo del código representado por dicho árbol sintáctico.

Dicha transformación fue compilada a Java (con versiones anteriores de SmartQVT que utilizaban parsers construidos en otros lenguajes, como Python) pasando así, a ser parte de la misma herramienta, demostrando de esta manera el poder de expresividad del lenguaje QVT (y de la propia herramienta).

A continuación se muestra un fragmento del código QVT que implementa esta herramienta:

```
#####
--Transformation
#####
mapping TransformationAst::toOperationalTransformation() :
OperationalTransformation {
    name :=
        self.p_transformation_header.p_identifier.value;
    isBlackbox:=
        self.p_transformation_header.m_qualifier[BlackboxQualifierAst]->size()>0;
    ownedType +=
        self.p_transformation_header.p_transformation_signature.p_simple_sig
        nature.m_param[ParamAst]->map toModelType()->asOrderedSet();
    modelParameter +=
        self.p_transformation_header.p_transformation_signature.p_simple_
        signature.m_param[ParamAst]->map toModelParameter()-> asOrderedSet();
    ownedVariable += result.map createThisVariable();
}
```

4.12 Together Architecture/QVT by Borland (QVT/Operational)

Together [26] es un producto de Borland que integra la IDE de Java con la herramienta de modelado UML del mismo nombre (tenía sus orígenes en JBuilder). Técnicamente, Together es un conjunto de plug-ins para Eclipse. Permite a los arquitectos de software trabajar focalizados en el diseño UML de manera gráfica; luego la herramienta se encarga de los procesos de análisis del diseño, y de generar un modelo de plataforma específica (PSM) a partir de modelos plataformas independientes (PIM) usando el QVT estándar [27].

Esta herramienta también es propietaria y requiere licencia para su uso.

4.13 Eclipse Modeling Project

El proyecto Eclipse Modeling [28] se enfoca en la evolución y promoción de tecnologías de desarrollo orientadas a modelos dentro de la comunidad de Eclipse proveyendo un conjunto de Frameworks de modelado, herramientas e implementaciones estándares.

Los dos frameworks más importantes que pertenecen a este proyecto y que pueden ser usados para implementar transformaciones en el lenguaje QVT son: EMF y M2M.

4.13.1 EMF (Eclipse Modeling Framework)

El proyecto EMF es un framework de modelado y generación de código para construir herramientas y otras aplicaciones basadas en un modelo de datos estructurados. A partir de una especificación de modelo descripta en XMI [15], 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), un nuevo 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-Core: 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 [15] (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:
 - Clases proveedoras de contenido y etiquetas, y otras clases que permiten a los modelos EMF ser mostrados usando visualizadores de escritorio estándar.
 - 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 necesario para construir un editor completo para un modelo EMF. Esto incluye un GUI desde el

cual pueden ser especificadas las opciones de generación, y los generadores a ser invocados. La generación se basa en JDT (Java Development Tooling), un componente de Eclipse. Hay tres niveles de generación de código que son soportadas:

- 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).
- Adaptadores: Genera clases de implementación (llamadas ItemsProviders) que adaptan las clases del modelo para ser editadas y mostradas.
- 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.

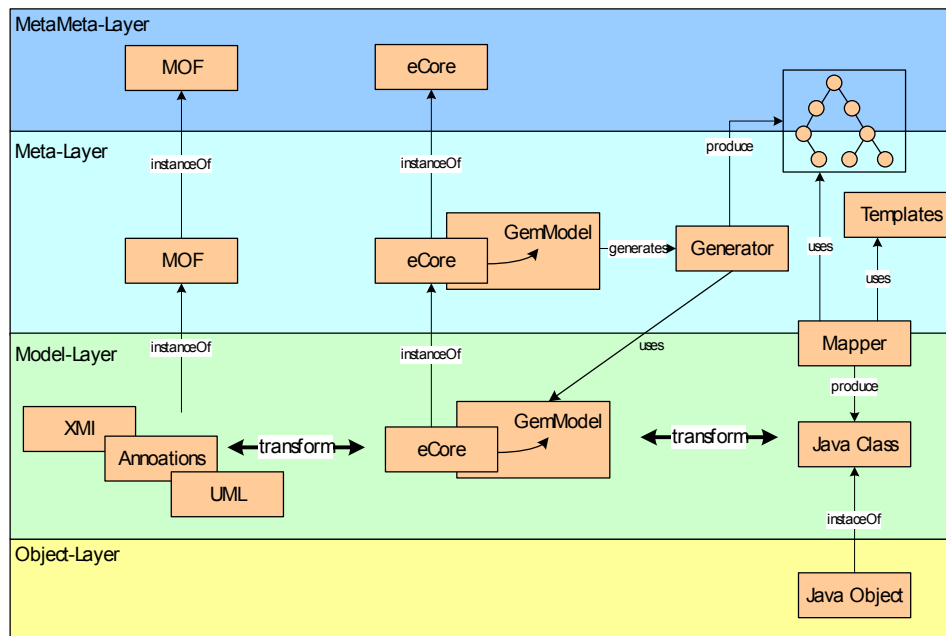


Imagen 12 - Modelo de EMF

4.13.2 Model To Model Transformation (M2M)

Es un framework extensible para lenguajes de transformación de modelo a modelo que ya viene con una implementación de ejemplo del lenguaje QVT.

4.14 Conclusiones

En la actualidad los lenguajes de transformación más usados y estables son QVT y ATL. Dado que uno de los propósitos de la tesis es difundir el lenguaje estándar de la OMG, la herramienta que se utilizara es Eclipse Modeling Project, el cual se detallara y fundamentara en el punto 7.2.

5 Frameworks investigados para la implementación del repositorio

En el siguiente punto se abordarán las posibles herramientas a ser utilizadas en la construcción del repositorio de transformaciones. El conjunto de las mismas, presentadas debajo, se basan en la plataforma Java y se adaptan a las necesidades tecnológicas del repositorio a desarrollar.

5.1 JSF

JSF (Java Server Faces) [29] es una especificación que busca el desarrollo de aplicaciones J2EE orientada a componentes con un modelo basado en eventos.

JSF es un framework de desarrollo basado en el patrón MVC (Modelo Vista Controlador), el cual pretende normalizar y estandarizar el desarrollo de aplicaciones web. El mismo es parte del estándar Java EE.

Una aplicación JSF se compone de páginas web con componentes JSF de interfaz de usuario (UI). Requiere de configuración las cuales se realizan en los archivos “faces-config.xml y web.xml”. El primero contiene la configuración de componentes JSF como, administrador de beans con sus alcances (Sesión, Pedido (Request), Aplicación y None), convertidores, validadores y la navegación de páginas. En el segundo se configuran los archivos generales de la aplicación web, esto es similar a aplicaciones web que están basadas en servlets.

En el “web.xml” se debe especificar el Faces Servlet que es responsable de manejar las aplicaciones JSF. Faces Servlet es el controlador central de una aplicación JSF, es el receptor de todos los pedidos de la aplicación JSF e inicializa los componentes JSF antes de mostrar una JSP. Una aplicación JSF corren en un contenedor web estándar como Tomcat o Jetty.

5.1.1 Las aplicaciones JSF típicamente contienen:

- Componentes Java Beans conteniendo datos y funcionalidades específicas de la aplicación.
- Oyentes de Eventos.
- Páginas, como JSP.
- Clases de utilidad del lado del servidor, como beans para acceder a las bases de datos.
- Una librería de etiquetas personalizadas para dibujar componentes UI en una página.
- Una librería de etiquetas personalizadas para representar manejadores de eventos, validadores, y otras acciones.

- Componentes UI representados como objetos con estado en el servidor.
- Validadores, manejadores de eventos y manejadores de navegación.

La librería de etiquetas de componentes puede ser la librería incluida con la implementación de Java Server Faces, o podemos definir nuestra propia librería de etiquetas que dibuje componentes personalizados o que dibuje una salida distinta a HTML.

Otra ventaja importante de las aplicaciones Java Server Faces es que los componentes UI de la página están representados en el servidor como objetos con estado. Esto permite a la aplicación manipular el estado del componente y conectar los eventos generados por el cliente a código en el lado del servidor.

En contrapartida a lo dicho anteriormente, JSF [29] nos permite convertir y validar datos sobre componentes individuales y reportar cualquier error antes de que se actualicen los datos del lado del servidor.

5.1.2 Existen diferentes implementaciones de JSF:

- Sun: JSF RI (Oracle)
- Oracle: ADF
- Apache: MyFaces
- JBoss: RichFaces
- ICESoft: IceFaces

5.1.3 Los principales componentes de la tecnología Java Server Faces son:

- Un API y una implementación de referencia para representar componentes UI y manejar su estado, manejo de eventos, validación del lado del servidor y conversión de datos, definir la navegación entre páginas, soportar internacionalización y proporcionar extensibilidad para todas estas características.
- Una librería de etiquetas Java Server Pages (JSP) personalizadas para dibujar componentes UI dentro de una página JSP.
- Un modelo de programación bien definido facilitando la tarea de la construcción y mantenimiento de aplicaciones Web.
- Conexión de eventos generados en el cliente con el código de la aplicación existente en el servidor.

- Mapear datos de componentes UI de una página con el servidor.
- Construir una UI con componentes reutilizables y extensibles.
- Grabar y restaurar el estado de la UI más allá de la vida de los pedidos al servidor.

Como se puede apreciar en la siguiente figura, la interface de usuario llamada myUI se ejecuta en el servidor y se renderiza en el cliente.

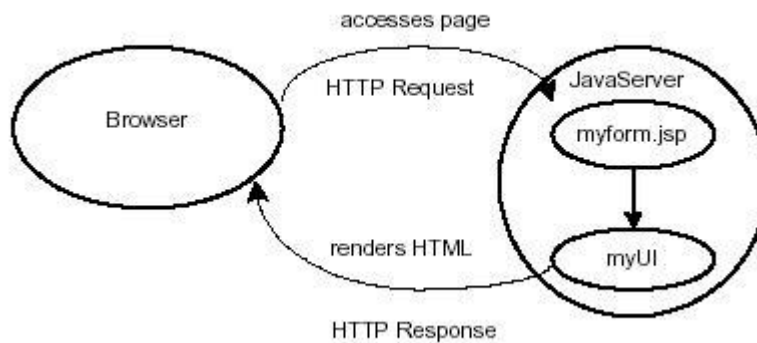


Imagen 13 - Ciclo JSF

La página JSP, myform.jsp, dibuja los componentes de interface de usuario con etiquetas personalizadas definidas por la tecnología JavaServer Faces. La UI de la aplicación Web (myUI) maneja los objetos referenciados por la página JSP.

5.1.4 Beneficios de la Tecnología JavaServer Faces

Una de las grandes ventajas de la tecnología JavaServer Faces es que ofrece una clara separación entre el comportamiento y la presentación. Esta separación entre la lógica y la presentación permite a cada miembro del equipo de desarrollo de una aplicación Web enfocarse en su parte del proceso de desarrollo. Proporcionando un sencillo modelo de programación para enlazar todas las piezas. Por ejemplo, los creadores de páginas sin experiencia en programación pueden usar las etiquetas de componentes UI de la tecnología JavaServer Faces para enlazar código de la aplicación desde dentro de la página Web sin escribir ningún script.

Otro objetivo importante de la tecnología JavaServer Faces es mejorar los conceptos de familias de componente-UI y capa-Web, sin limitarnos a una tecnología de script particular o un lenguaje de marcas particular. Aunque la tecnología JavaServer Faces incluye una librería de etiquetas JSP personalizadas para representar componentes en una página JSP, las APIs de la tecnología JavaServer Faces se han creado directamente sobre el API Java Servlet. Esto nos permite usar otra tecnología de presentación junto a JSP, crear nuestros propios componentes

personalizados directamente desde las clases de componentes, y generar salida para diferentes dispositivos cliente.

Por último y resumiendo en pocas líneas la tecnología JavaServer Faces nos proporciona una rica arquitectura para manejar el estado de los componentes, procesar los datos, validar la entrada del usuario y manejar eventos.

5.2 JPA

JPA (Java Persistence Api) [30], provee un mapeo Objeto/Relacional para facilitar a los desarrolladores el manejo de datos y relaciones en aplicaciones java. La persistencia en Java consiste de 3 áreas:

5.2.1 La API de persistencia Java

Esta API es la encargada de administrar la manera en que los datos son mapeados a objetos java ("entidades persistentes"), y la forma en que estos objetos son almacenados en una base de datos relacional para luego poder ser accedidos. Mantiene el estado de la entidad aún después de que la aplicación que la usa finalice. Adicionalmente, para simplificar el modelo de persistencia de entidades, la API de persistencia java estandariza el mapeo Objeto/Relacional.

Se basa en los frameworks de persistencia y APIs más populares, tales como, Hibernate, Oracle Toplink o JDO.

JPA simplifica el modelo de programación para persistir entidades y agrega capacidades que no estaban en el EJB 2.1. A continuación se detalla una lista de sus simplificaciones y agregados:

- Requiere menos clases e interfaces.
- Elimina gran parte de los descriptores a través de las anotaciones (annotations).
- Direcciona las especificaciones más típicas mediante anotaciones por default.
- Provee un mapeo Objeto/Relacional más claro, simple y estandarizado.
- Elimina la necesidad de código de búsqueda (lookup code).
- Agrega soporte para herencia, polimorfismo, y consultas polimórficas.
- Agrega soporte para consultas dinámicas y estáticas.
- Provee un lenguaje de consulta para persistencia Java (EJB QL enriquecido).
- Hace más fácil testear entidades fuera del contenedor EJB.
- Puede ser usado con proveedores de persistencia de terceras partes.

5.2.2 El lenguaje de consulta

EJB QL ha sido una faceta muy popular de la tecnología EJB. Sin embargo, a pesar de su popularidad, EJB QL ha carecido de algunas de las características de un lenguaje de consulta

completamente estructurado, como las operaciones masivas de actualización o borrado (bulk), operaciones outer joins, proyecciones y subconsultas. El lenguaje de consulta para persistencia en java agrega todas estas características. Todas las características de este lenguaje pueden ser usadas tanto en consultas estáticas como dinámicas.

5.2.3 Metadata para el mapeo Objeto/Relacional

La metadata en forma de Anotación @Entity marca una clase como una entidad. EJB 3.0 [55] y JPA se basan fuertemente en el uso de metadata en forma de anotaciones, una característica que fue introducida en J2SE 5.0 [54]. Una anotación consiste del @ precediendo un tipo de anotación, a veces seguido por una lista de pares elemento-valor. La especificación EJB 3.0 define una gran variedad de tipo de anotaciones. Algunos ejemplos son aquellos que especifican el tipo de un bean, como es @Stateless; aquellos que especifican cuando un bean es accesible remotamente o localmente, como los son @Remote y @Local; atributos de transacciones tales como @TransactionAttribute; seguridad y permisos sobre métodos tales como @MethodPermission, @Unchecked y @SecurityRoles. La API JPA agrega anotaciones, tales como @Entity, las cuales son específicas para entidades.

5.3 Seam

Seam es un framework liviano (lightweight) para Java EE. Provee un modelo de programación consistente y sencillo de entender para todos los componentes en una aplicación web empresarial. En otras palabras, Seam se basa en la productividad de los desarrolladores y en la escalabilidad de las aplicaciones.

Seam [31] es una poderosa plataforma de desarrollo de código abierto para construir aplicaciones web en Java. Seam integra tecnologías tales como, Asynchronous JavaScript sobre XML (AJAX), JavaServer Faces (JSF), Java Persistence (JPA), Enterprise Java Beans (EJB 3.0) y Business Process Management (BPM), en un sistema unificado completo con sofisticadas herramientas (JBoos Tools).

Seam ha sido diseñado desde el comienzo para eliminar la complejidad en la arquitectura y los niveles de API. Permitiendo a los desarrolladores ensamblar complejas aplicaciones web usando simples anotaciones en clases Java, un rico conjunto de componentes interfaz de usuario (UI) y muy pocos XML. El soporte único de Seam para conversaciones y administración de estados declarativos puede requerir usuarios más experimentados, mientras que al mismo tiempo elimina errores comunes encontrados en aplicaciones web tradicionales.

Seam está bajo licencia LGPL. Las herramientas de desarrollo para aplicaciones Seam son basadas en eclipse [35] y son provistas por JBoos Tools [34]. El framework está basado sobre la plataforma Java EE.

5.3.1 Características

5.3.1.1 Se integra y enriquece Frameworks Java EE

Los Frameworks centrales en Java EE 5.0 [53] son EJB (Enterprise JavaBeans) 3.0 y JSF (javaServer Faces) 1.2. EJB 3.0 es una framework liviano basado en objetos POJO para la capa de negocio y para la persistencia en base de datos. JSF es un framework basado en MVC para aplicaciones web.

La mayoría de las aplicaciones web Java EE 5.0 tienen módulos EJB3 para la lógica de negocio y módulos JSF para la presentación web de la aplicación. Sin embargo, a pesar de que EJB3 y JSF son complementarios entre sí, están diseñados como Frameworks separados cada uno con su propia filosofía. Por ejemplo, EJB3 usa anotaciones para configurar los servicios, mientras que JSF usa archivos XML. Más aun, los componentes EJB3 y JSF no están acoplados, sino que para poder trabar juntos es necesario objetos facade para enlazar componentes de negocio a las páginas web, y código repetitivo para hacer llamados a métodos a través de los límites del framework. Juntar estas tecnologías es parte de las responsabilidades de Seam.

Seam derrumba la capa artificial que había que crear entre EJB3 y JSF, para ello, provee un mecanismo consistente, basado en anotaciones, para integrar EJB3 y JSF. Con unas pocas anotaciones, los componentes de negocio EJB3 puede ser ahora directamente usados para retornar a los formularios web JSF o para manejar eventos UI webs.

Seam provee un gran número de tags JSF y anotaciones para incrementar el aspecto amigable de las páginas y la eficiencia de las mismas en aplicaciones JSF.

Al mismo tiempo, Seam extiende el modelo de componentes EJB3 para POJOs y brinda un contexto con estado (stateful) desde la capa web a los componentes de negocio. Más aún, Seam integra otros Frameworks open source, como jBPM, JBoss Rules, iText, y Spring. Seam, no sólo los une, sino que también mejora los Frameworks de la misma manera que lo hace con la combinación JSF + EJB3.

A pesar de que Seam tiene sus raíces en Java EE 5.0, su aplicación no está limitada a servidores Java EE 5.0 [53], en particular las aplicaciones Seam pueden ser desplegadas en servidores de aplicaciones J2EE 1.4 así como también en servidores Tomcat [50].

5.3.1.2 Un framework web que entiende ORM

Seam fue inventado por Gavin King, el inventor de Hibernate, la solución ORM más popular del mundo. Con Seam, no hay más necesidad de escribir DTOs, la carga lazy simplemente funciona, la eficiencia del ORM puede ser mejorada dado que el contexto de persistencia actúa como una cache natural para reducir las idas y vueltas a la base de datos.

Más aún, debido a que Seam integra la capa ORM con la capa de negocios y presentación, podemos mostrar objetos ORM directamente, usar anotaciones para validaciones de base de datos y redireccionar las excepciones ORM a páginas de errores personalizadas.

5.3.1.3 Diseñado para aplicaciones web con estado

Las aplicaciones web son inherentemente aplicaciones multiusuario, y las aplicaciones de comercio electrónico (e-commerce) son inherentemente con estado y transaccionales. Sin embargo, la mayoría de los Frameworks web existentes están orientados hacia aplicaciones sin estado. Por ello hay que lidiar con los objetos de la sesión HTTP para manejar ese estado deseado. Esto no sólo que desordena la aplicación con código que no está relacionado con la lógica de negocio, sino que también lleva a tener en cuenta muchas cuestiones de eficiencia.

En Seam, todos los componentes básicos de la aplicación son inherentemente con estado. Son mucho más fáciles de usar que con las sesiones HTTP debido a que Seam maneja declarativamente sus estados. No es necesario escribir código para el manejo de ellos, sólo basta anotar los componentes con su contexto, los métodos del ciclo de vida, otras propiedades de estado y Seam se encarga del resto. Los componentes con estados también permiten un control más fino sobre los estados del usuario que el que permite una sesión HTTP. Por ejemplo, podemos tener múltiples “conversaciones”, cada una correspondiente a una secuencia de solicitudes web y llamadas a métodos de negocio, en una sesión HTTP.

Más aún, las transacciones y caches de base de datos pueden ser automáticamente enlazadas con el estado de la aplicación en Seam. Seam automáticamente maneja las actualizaciones en memoria y los commits a la base de datos solo al final de una conversación.

En resumen, cada componente en Seam tiene un alcance o contexto. Por ejemplo, un componente carrito de compra de un shopping es creado al comienzo de la conversación y destruido al final de la conversación cuando todos los ítems fueron chequeados. De esta forma, el componente vive en el contexto de la conversación. La aplicación simplemente declara este contexto vía anotaciones sobre el componente y Seam automáticamente maneja la creación del componente, el estado y su eliminación o destrucción. Seam provee muchos niveles de contextos de estados, que van desde una simple solicitud web a una conversación multipágina, una sesión HTTP, o un proceso de negocio de largo alcance.

5.3.1.4 Preparado para la Web 2.0

Seam está completamente optimizado para el estilo de aplicaciones web 2.0. Este provee múltiples formas para AJAX, además provee un modelo de concurrencia avanzado que permite manejar eficientemente múltiples solicitudes AJAX desde el mismo usuario.

Un gran desafío para las aplicaciones AJAX es el incremento en la carga de trabajo de la base de datos. Una aplicación AJAX hace muchas más solicitudes al servidor que una aplicación que no usa AJAX. Si todas esas solicitudes AJAX tendrían que ser servidas por la base de datos, la base de datos podría no ser capaz de manejar la carga. El contexto de persistencia en Seam actúa como una cache en memoria. Esta puede mantener la información a través de una conversación de larga duración y, de esta forma, ayudar a reducir las idas y vueltas a la base de datos.

5.3.1.5 Servicios POJO vía inyección de dependencia

Seam une los componentes POJOs usando un patrón de diseño conocido como inyección de dependencia. Bajo este patrón, el framework Seam maneja el ciclo de vida de todos los componentes. Cuando un componente necesita usar otro, este declara su dependencia a Seam

usando anotaciones. Seam determina donde obtener esta dependencia dependiendo del estado actual de la aplicación e inyecta este dentro del componente solicitante.

Expandiendo el concepto de inyección de dependencia, un componente Seam A puede además crear un componente B y devolver (outject) el componente a Seam para que otros componentes, como C, lo usen luego.

5.3.1.6 Configurar solo por excepción

La clave principal que hace a Seam fácil de usar es que sólo hay que configurar en casos excepcionales. La idea es tener un conjunto de comportamientos por default para los componentes. El desarrollador necesita configurar un componente sólo cuando se necesita un comportamiento que no es el default. El resultado global es que la metadata para configuración en Seam es más simple que en el resto de los Frameworks java. Como resultado, la mayoría de las aplicaciones Seam pueden ser configuradas adecuadamente con un número reducido de simples anotaciones java.

5.3.1.7 Evita el Abuso de XML

En tiempos anteriores de J2EE, XML era visto como "la salvación" para administrar las configuraciones, pero los archivos de configuración XML eran altamente repetitivos. La comunidad de java reconoció este problema y ha tenido un intento exitoso al reemplazar los archivos XML con anotaciones en el código Java. EJB3 es el esfuerzo de la comunidad java para usar anotaciones en componentes Java empresariales. Seam se adhiere a las anotaciones de EJB3 y expande el modelo de programación basado en anotaciones a la aplicación web entera.

Por supuesto, XML no es completamente malo para configuración. Los diseñadores de Seam reconocen que el XML es bueno para especificar el flujo de las páginas en una aplicación web o para definir los workflows de procesos de negocio.

5.3.1.8 Diseñado para Testing

Seam fue diseñado desde cero para facilitar el testeo. Como todos los componentes Seam son simplemente objetos POJOs con anotaciones, estos son fáciles de testear. Podríamos crear instancias de objetos POJOs usando la palabra clave Java new y luego ejecutar cualquiera de los métodos en nuestro framework de Testing (por ejemplo, JUnit o TestNG). Si necesitamos interacción en el test entre múltiples componentes Seam, podríamos instanciar dichas componentes individualmente y luego configurar sus relaciones manualmente.

Con el framework de Testing de Seam, podríamos escribir scripts simples para simular la interacción del usuario web, y luego testear el resultado.

5.4 Maven2

Maven [33] es una palabra alemana que significa acumulador de conocimiento, comenzó originalmente como un intento para simplificar el desarrollo de procesos en el proyecto Jakarta Turbine. Había muchos proyectos con sus propios archivos Ant los cuales eran ligeramente diferentes y contenían archivos JARs que estaban controlados dentro de un CVS. Se buscaba una manera estándar de construir los proyectos, una definición clara sobre la estructura de un proyecto, una forma fácil de publicar información del proyecto y una manera de compartir JARs a través de varios proyectos.

El resultado es una herramienta que ahora puede ser usada para construir y administrar cualquier proyecto basado en java. Esta herramienta debería hacer el trabajo día a día más fácil para los desarrolladores java y ayudar con la compresión de cualquier proyecto basado en java.

5.4.1 Objetivo

El objetivo principal de Maven es permitir a un desarrollador emplear el menor esfuerzo de desarrollo posible. Con el fin de alcanzar este objetivo, hay varias aéreas de preocupación contra la que Maven intenta lidiar:

- Hacer que el proceso de construcción sea simple.
- Proveer un sistema uniforme de construcción.
- Proveer calidad en la información del proyecto.
- Proveer una guía de buenas prácticas de desarrollo.
- Permitir transparencia en la migración a nuevas características.

5.4.2 Características

Las siguientes son las características claves de Maven:

- Configuración de proyecto simple que sigue las buenas prácticas.
- Uso consistente a través de todos los proyectos, lo cual significa que no incrementamos tiempos para agregar nuevos desarrolladores a un proyecto.
- Administración de dependencias superiores incluyendo actualizaciones automáticas, dependencias cerradas (también conocidas como dependencias transitivas).
- Capacidad para trabajar fácilmente con múltiples proyectos al mismo tiempo.
- Un repositorio grande y creciente de librerías para usar por fuera.
- Extensible, con la capacidad de escribir fácilmente plugins en java o en lenguajes de scripting.
- Acceso instantáneo a las nuevas características con un poco o casi nada de configuración extra.
- Tareas ANT para el manejo de dependencias y el deployment fuera de Maven.

- Construcciones basadas en modelos: Maven es capaz de construir cualquier número de proyectos con tipos predefinidos de salida como es JAR, WAR, o distribución basada sobre la metadata del proyecto, sin la necesidad de hacer scripting en la mayoría de los casos.
- Ubicación coherente de la información del proyecto: usando la misma metadata que en el proceso de construcción, Maven es capaz de generar un sitio web o un PDF incluyendo cualquier documentación que se quiera agregar, y agrega a estos reportes estándares el estado de evolución del proyecto.
- Gestión de Versiones y distribución de publicaciones: sin demasiada configuración adicional, Maven se integra con tu sistema de control de código fuente, como CVS o SVN, y maneja la versión de un proyecto basado en un determinado tag.
- Gestión de dependencias: Maven fomenta el uso de un repositorio central de JARs y otras dependencias. Maven tiene un mecanismo que permite a los proyectos bajar las librerías requeridas para las construcciones desde un repositorio centralizado al igual que el CPAN para Perl.

6 Caso de Estudio – Transformación UML2RDBMS

Para un mejor entendimiento del concepto de transformación de modelos, presentamos un ejemplo extraído del manual de QVT. El mismo se describe en lenguaje QVTR y QVTO.

El caso de estudio se basa en la transformación de un modelo del metamodelo UML a un modelo del metamodelo RDBMS.

6.1 Lenguaje QVTR

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 instancia de esta transformación podría ser, por ejemplo, la que se muestra en la imagen 14 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.

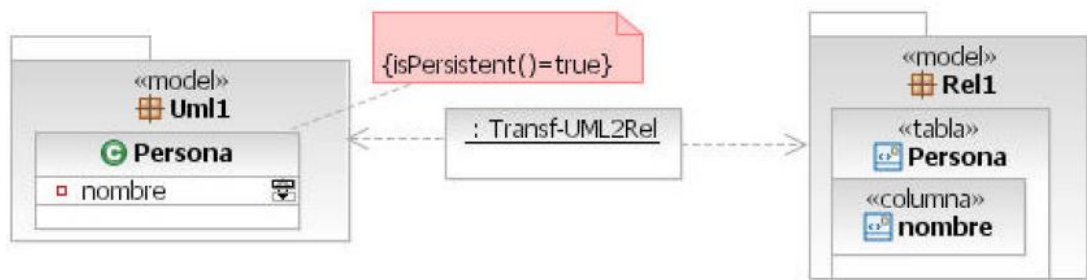


Imagen 14 - Instanciación de una transformación

6.2 Lenguaje QVTO

Este ejemplo muestra la especificación textual de una transformación llamada Simpleuml_To_Rdb en QVT Operacional (QVTO).

La Signatura de la transformación declara que requiere un modelo de tipo UML como entrada, y que como resultado produce un modelo de tipo RDB. El modelo de tipo UML es referenciado como 'uml' en el resto de la transformación, mientras que para referenciar al modelo de salida no es necesario un nombre.

```

transformation Simpleuml_To_Rdb(in uml : UML, out RDB);
    
```

El inicio o punto de entrada de una transformación es el main. La referencia 'uml' es usada para recolectar todos los rootObjects del modelo. La operación rootObjects está disponible sobre todos los objetos del modelo QVT y retorna aquellos objetos en la raíz del modelo de entrada. La

invocación del mapping `model2RDBModel()` se realiza usando el operador `->` el cual es un shortcut de la sentencia imperativa `collect`. Alternativamente se podría escribir de la siguiente forma:

```
uml.rootObjects()[UML::Model]->xcollect(a | a.map model2RDBModel());

main() {
    uml.rootObjects()[UML::Model]->map model2RDBModel();
}
```

Este mapping produce un objeto tabla RDB desde una clase UML persistente. La clausula `when` usa la query `isPersistent()` para ver si la clase (Class) tiene el valor 'persistent' en su propiedad `stereotype`. La atributo `name` se mapea directamente. La clase es mapeada a un conjunto (set) de objetos `TableColumn` usando el mapping `class2column()`, cuyos resultados son ordenados por nombre usando la operación OCL `sortedBy()`. La clave primaria (`primaryKey`) es seteada usando el mapping `class2primaryKey()`, mientras que las claves foráneas (`foreignKeys`) son seteadas usando la función `resolveIn`.

```
mapping UML::Class::persistentClass2table() : RDB::Table
    when { self.isPersistent() }
{
    name := self.name;
    columns := self.map class2columns(self)->sortedBy(name);
    primaryKey := self.map class2primaryKey();
    foreignKeys :=
        self.attributes.resolveIn(
            UML::Property::relationshipAttribute2foreignKey,
            RDB::constraints::ForeignKey)->asOrderedSet();
}
```

Este mapping retorna un `OrderedSet` de objetos `TableColumn` mediante la invocación del mapping `primitiveAttribute2column()` para cada atributo de `DataType`.

```
mapping UML::DataType::primitiveAttributes2columns(in targetType:
    UML::DataType) : OrderedSet(RDB::TableColumn)
{
    init {
        result := self.attributes->map
            primitiveAttribute2column(targetType)->asOrderedSet()
    }
}
```

Este mapping crea un objeto `TableColumn` a partir de una `Property` cuando el query `isPrimitive()` retorna `true`. Los mappings para el nombre (`name`) y si es clave primaria (`isPrimaryKey`) son simples, mientras que el tipo (`type`) es creado usando la palabra clave `object` para crear un nuevo objeto `PrimitiveDataType` con el nombre inicializado a el resultado de la query `umlPrimitive2rdbPrimitive()` con el nombre del tipo pasado como parámetro.

```
Mapping UML::Property::primitiveAttribute2column(in targetType:
    UML::DataType) : RDB::TableColumn
    when { self.isPrimitive() }
{
```

```

isPrimaryKey := self.isPrimaryKey();
name := self.name;
type := object RDB::datatypes::PrimitiveDataType { name :=
    umlPrimitive2rdbPrimitive(self.type.name); };
}

```

6.3 Conclusión

A pesar del hecho que QVT ofrece dos perspectivas de modelado – esto nos permite especificar que hace la transformación (QVT declarativo) y también cómo se concreta su ejecución (QVT operacional) – muchos de los trabajos actuales sobre transformación de modelos presentan esencialmente naturaleza operacional y ejecutable. Las transformaciones pueden también ser vistas como modelos descriptivos que establecen solamente las propiedades que una transformación tiene que cumplir y omiten detalles de ejecución.

En cuanto a su implementación, QVT Declarativo y QVT Operacional tienen enfoques totalmente distintos entre sí.

QVT operacional está planteado como un lenguaje imperativo donde cada relación es una función o procedimiento con algunas restricciones. Tiene un punto de arranque “main” que es igual al cuerpo del programa principal que existe en la mayoría de los lenguajes imperativos. Cuenta con definición de parámetros formales y con mecanismos de reutilización de librerías.

El flujo de control de una transformación operacional arranca en el cuerpo de la transformación y luego se delega en las invocaciones a los mappings de manera secuencial.

QVT Declarativo tiene un enfoque diferente. Se basa en un conjunto de reglas que deben cumplirse simultáneamente para determinar si existe consistencia entre dos conjuntos (dominio y codominio). Si no se cumpliera la consistencia entre conjuntos, entonces se crean los elementos necesarios (habitualmente en el codominio) para forzar la consistencia. Ese mecanismo de fuerza se lo puede interpretar como una transformación de los elementos del dominio en elementos del codominio según las reglas (relations) definidas en la transformación.

Para este caso, no existe flujo de control de la ejecución. Todas las reglas se tienen que satisfacer simultáneamente. Lo que se produce es un grafo de reglas, donde alguna regla tiene como precondition tanto a predicados como a otras reglas (en el when) y como pos condición, también predicados y otras reglas de transformación (en el where). A veces uno se ve tentado a asociar a las pos condiciones de otras reglas con “invocaciones” a funciones (relations), pero no es así. Tampoco hay reglas que puedan interpretarse como que se ejecutan “primero” sino que en realidad, una top relation tiene precedencia sobre una relation que no es top.

7 Nuestra Propuesta (Especificación e Implementación de Transformaciones de Modelos de Software aplicando Estándares)

El capítulo comienza con la motivación en la cual se basa parte de la tesis. Luego se presenta el lenguaje y la herramienta a ser usadas para alcanzar los objetivos, además se brinda una breve introducción al ambiente Eclipse Modeling Project y finalmente se detallan las transformaciones que fueron implementadas en QVT.

7.1 Motivación: Repositorio de transformaciones de ATL

La arquitectura dirigida por modelos permite realizar transformaciones de modelos, hecho por el cual encontramos que podía ser de gran utilidad desarrollar transformaciones de modelos, los cuales representan problemas recurrentes y comunes en el desarrollo de software, en particular en la etapa de diseño.

Nuestra investigación inicial se basó en el estudio del repositorio de transformaciones desarrolladas en ATL [37] [38]. En el mismo se encuentran realizadas transformaciones que solucionan problemas recurrentes en el diseño de modelos de software, pero desarrolladas en un lenguaje que no es el estándar definido por la OMG.

La propuesta de este trabajo es realizar un aporte a la solución de problemas recurrentes en el diseño de diagramas de clases UML, mediante el desarrollo de transformaciones, análogas a las que se encuentran en el repositorio ATL, utilizando el lenguaje estándar definido por la OMG y de esta forma fomentar el desarrollo de transformaciones bajo este lenguaje.

Como complemento, y tomando como referencia el repositorio existente de transformaciones en lenguaje ATL, se propondrá la arquitectura y diseño de un repositorio donde alojar las transformaciones.

7.2 Elección de herramienta y lenguaje de desarrollo.

El lenguaje de desarrollo estándar definido por la OMG es QVT, el cual tiene dos implementaciones, QVT Operacional y QVT Declarativo. La implementación de QVT que vamos a utilizar para el desarrollo de las transformaciones de la tesis es la operacional. A continuación describimos los motivos por los cuales elegimos QVT Operacional:

- Está planteado como un lenguaje imperativo donde cada relación es una función o procedimiento con algunas restricciones.
- Tiene un punto de arranque “main” que es igual al cuerpo del programa principal que existe en la mayoría de los lenguajes imperativos.
- Cuenta con definición de parámetros formales y con mecanismos de reutilización de librerías.
- El flujo de control de una transformación operacional arranca en el cuerpo de la transformación y luego se delega en las invocaciones a los mappings de manera secuencial.

- Al tener un flujo de control es más sencillo de debuguear y testear.
- Al ser un lenguaje más convencional que el declarativo, los aportes de documentación y ejemplos son mayores, de mejor legibilidad y calidad.

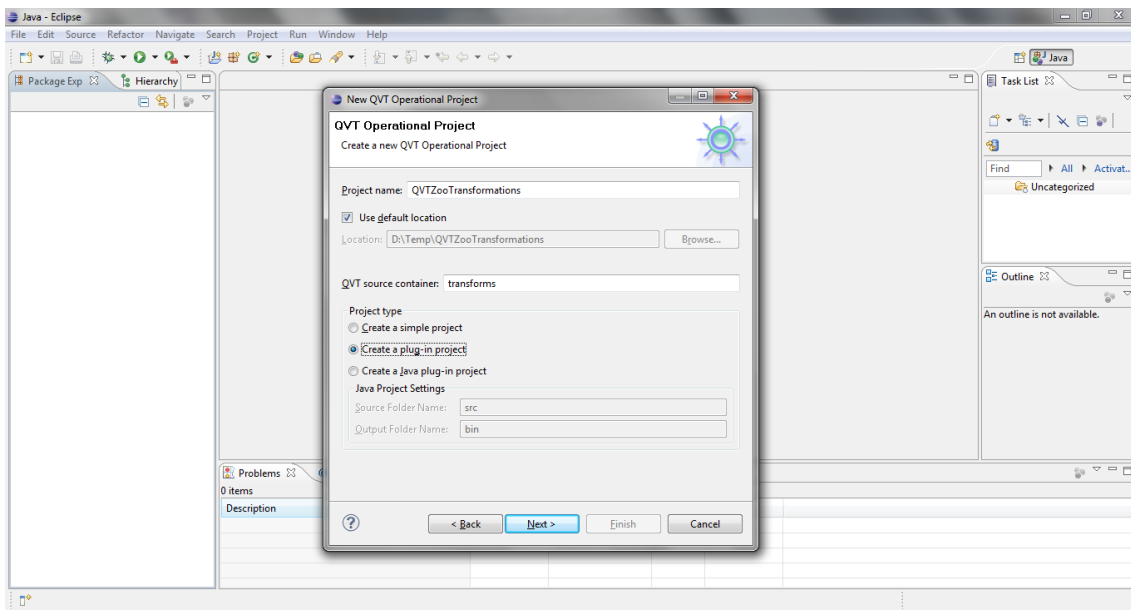
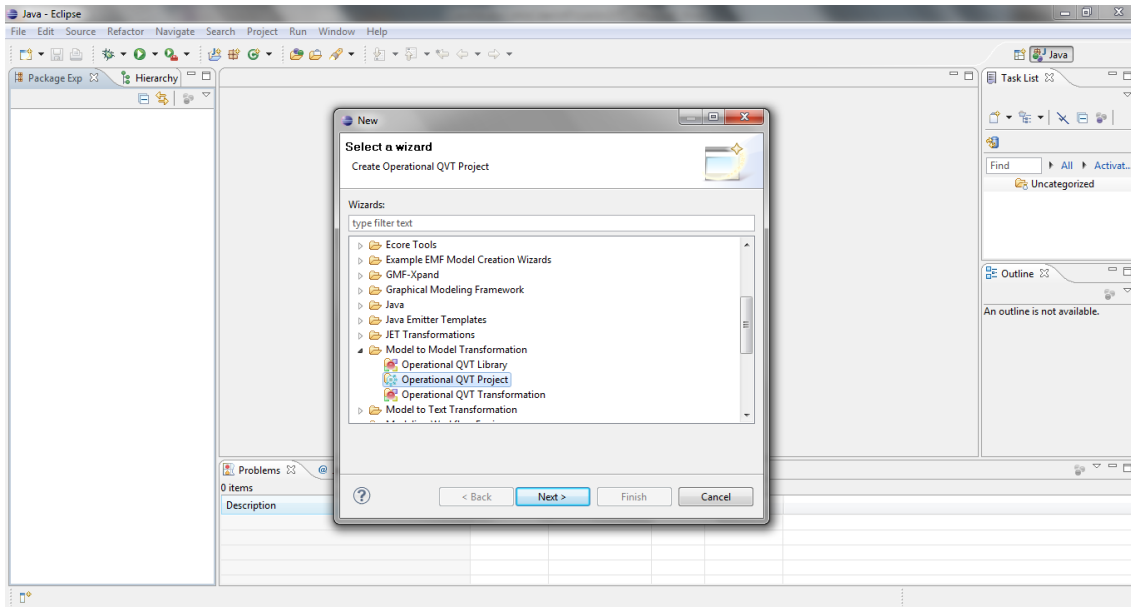
Una vez tomada la decisión respecto a la implementación del lenguaje a utilizar se buscó la herramienta pertinente. De hace un tiempo la comunidad Eclipse brinda un total soporte a herramientas MDD, buena muestra de ello es el proyecto Eclipse Modeling Project, el cual se basa en el Eclipse Modeling Framework (EMF).

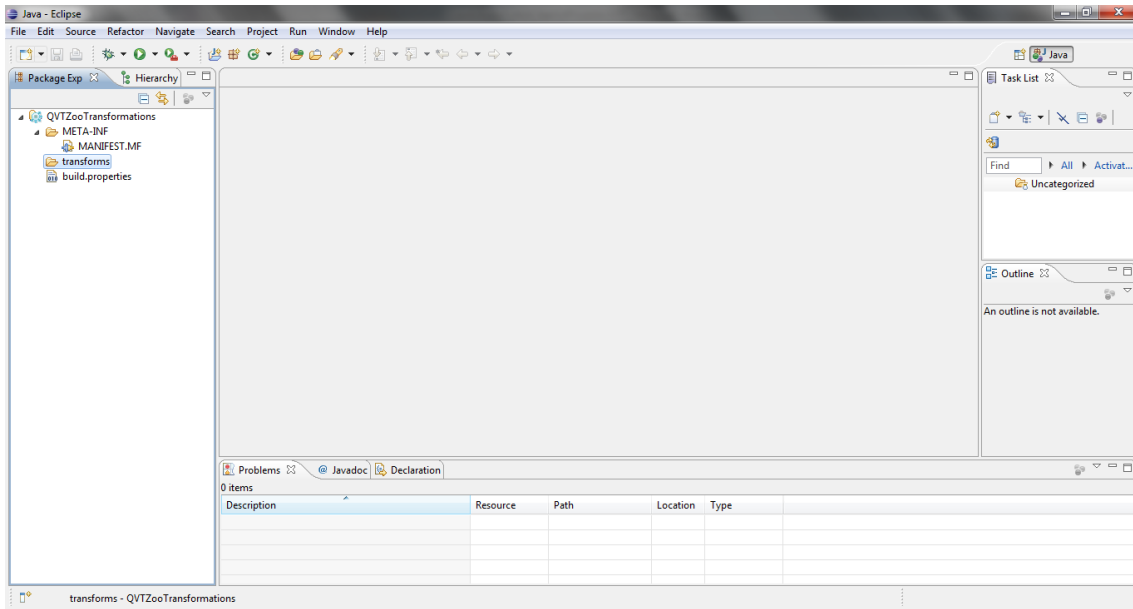
El proyecto Eclipse Modeling Project se enfoca en la evolución y promoción de tecnologías de desarrollo orientadas a modelos dentro de la comunidad de Eclipse proveyendo un conjunto de Frameworks de modelado, herramientas e implementaciones estándares. Uno de los Frameworks provistos es el Model To Model Transformation (M2M) que es el que usaremos para desarrollar las transformaciones en QVT Operacional.

Entre otras cosas, la elección de esta herramienta se basó en que es un proyecto Open source que está en continuo desarrollo y mantenimiento, y que corre bajo Eclipse, por ende es multiplataforma.

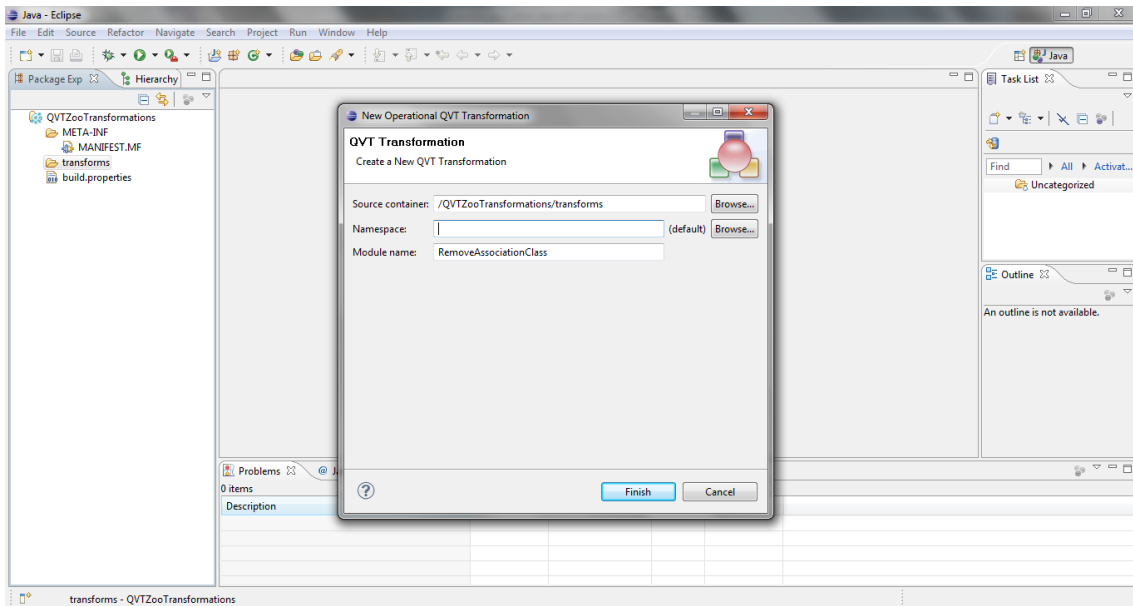
7.3 Visión de la herramienta de desarrollo Eclipse Modeling

Crear un Proyecto QVT Operacional: File / New / Other / Model to Model Transformation / Operational QVT Project.



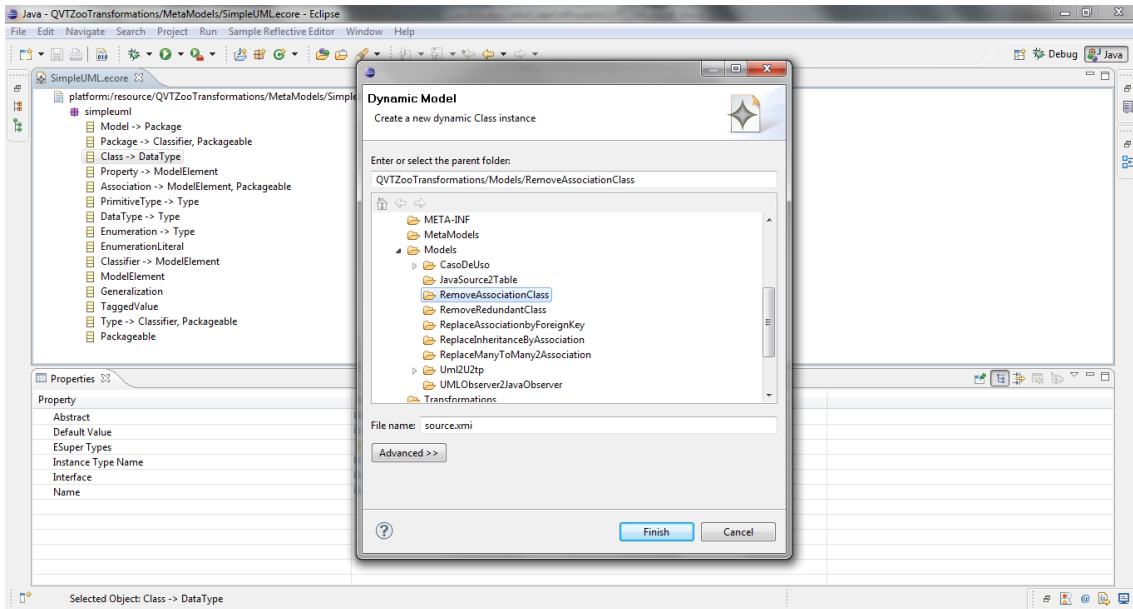


Crear una nueva Transformación: File / New / Operational QVT Transformation.



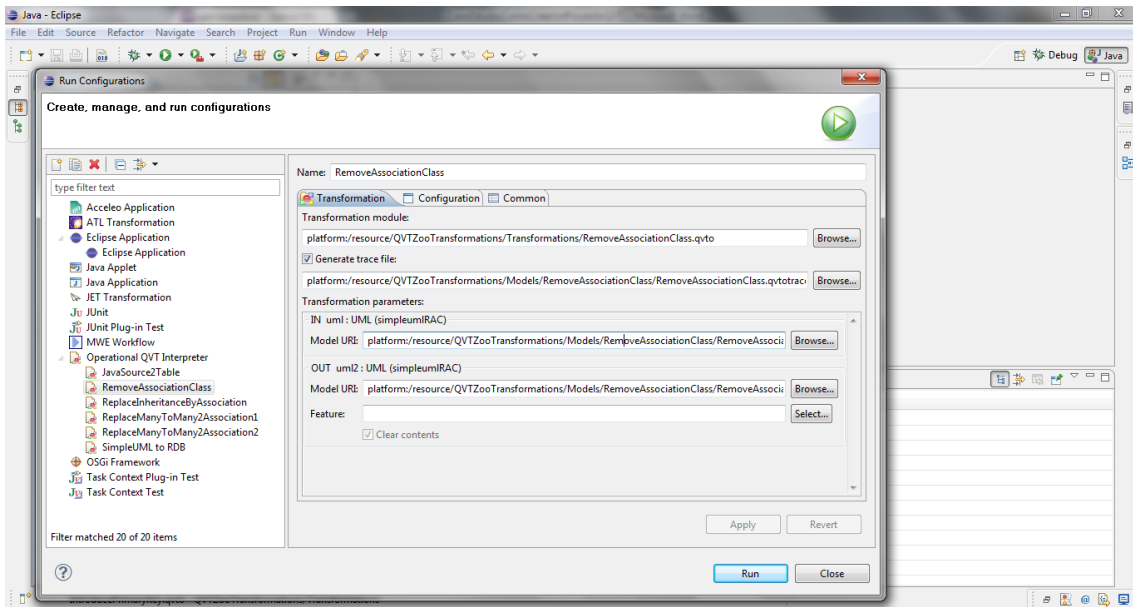
Crear una instancia de un metamodelo sobre el cual vamos a ejecutar una transformación:

Para crear un Modelo sobre un metamodelo, abrir el archivo .ecore y hacer botón derecho sobre el elemento del metamodelo del cual queremos crear una instancia, por ejemplo, el elemento Class del metamodelo SimpleUml.ecore y seleccionar Create Dynamic Instance...



Ejecutar una transformación:

Para ejecutar una transformación, selecciónela y presione el botón derecho, Run As/ Run Configuration



Además de determinar la transformación a ejecutar se deben configurar los siguientes parámetros:

- El nombre del archivo de Trace que se genera al ejecutar la transformación siempre y cuando este seleccionado el tilde Generate Trace File.

- El archivo que representa el modelo de entrada de la transformación.
- El nombre del archivo donde se generara el modelo de salida.

7.4 Transformaciones seleccionadas para la tesis

En este punto se detallan las transformaciones que componen nuestra tesis. De cada una de ellas se presenta una breve descripción, los metamodelos utilizados, las reglas de transformación y finalmente el código fuente de las mismas.

7.4.1 JavaSource2Table

Descripción

El ejemplo JavaSource2Table describe una transformación desde un código fuente Java a una tabla la cual totaliza cuantas veces un método declarado en el código fuente java es llamado dentro la definición de cualquier otro método declarado. Usaremos, para esta transformación, un metamodelo Table básico y abstracto el cual podría ser fácilmente mapeado a representaciones de tablas existentes (XHTML, ExcelML, etc.).

El objetivo de esta transformación es generar, a partir de un código fuente Java, una tabla que totaliza cuantas veces un método es llamado dentro de la definición de los métodos declarados. La tabla generada se organiza de la siguiente manera:

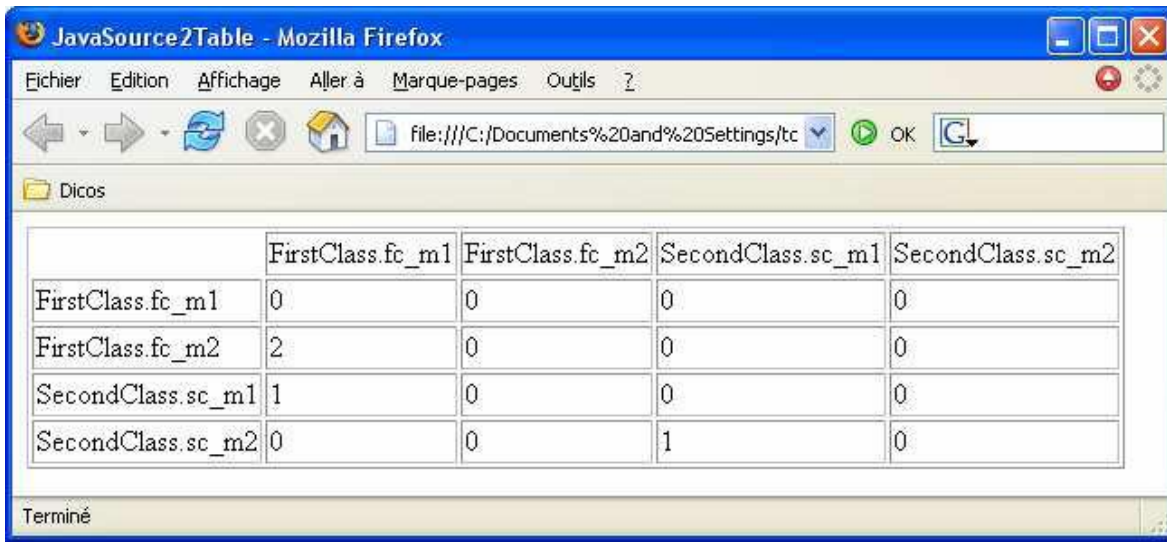
Excepto la primera celda, la cual está vacía, la primera fila contiene la lista de métodos declarados en el código fuente java, ordenados de acuerdo al texto “nombre_clase.nombre_metodo”, donde “nombre_metodo” es el nombre del método, y “nombre_clase” es el nombre de la clase en la cual el método está definido.

La primera columna está organizada de la misma forma que la primera fila.

Las celdas de las filas subsiguientes contienen el número de llamadas del método en la columna dentro de la declaración del método en la fila.

FirstClass.java	SecondClasss.java
<pre> public class FirstClass { public void fc_m1() { } public void fc_m2() { this.fc_m1(); this.fc_m1(); } } </pre>	<pre> public class SecondClass { public void sc_m1() { FirstClass a = new FirstClass(); a.fc_m1(); } public void sc_m2() { this.sc_m1(); } } </pre>

Imagen 15 - Código Fuente Java



	FirstClass.fc_m1	FirstClass.fc_m2	SecondClass.sc_m1	SecondClass.sc_m2
FirstClass.fc_m1	0	0	0	0
FirstClass.fc_m2	2	0	0	0
SecondClass.sc_m1	1	0	0	0
SecondClass.sc_m2	0	0	1	0

Terminé

Imagen 16 - Invocaciones de métodos totalizadas para FirstClass y SecondClass.

Como ejemplo de esta transformación, la imagen 16 provee la tabla en XHTML correspondiente al código fuente Java compuesto por dos clases, FirstClass y SecondClass, definidas en la imagen 15. Todos los métodos definidos aparecen tanto en las filas como en las columnas, ordenadas por su clase y su nombre. La tabla debe ser leída de la siguiente manera: el método “FirstClass.fc_m2” (fila 3) es llamado dos veces en la definición del método “FirstClass.fc_m1” (columna 2), pero no en las definiciones de los otros métodos.

MetaModelos

Esta transformación se basa en un metamodelo básico, JavaSource, el cual sólo contiene la información que es relevante para el alcance de esta transformación.

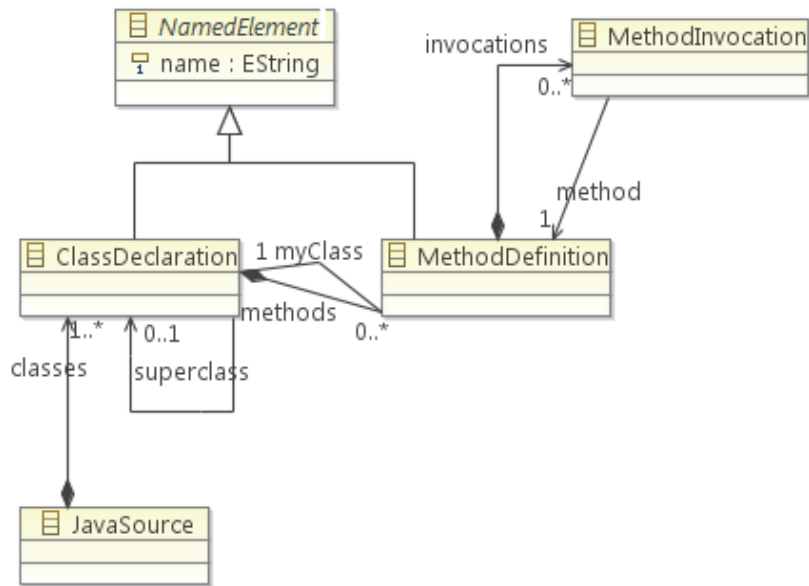


Imagen 17 - Metamodelo básico, JavaSource

Un Modelo de código fuente java es modelizado por un elemento **JavaSource**. Este elemento está compuesto de **ClassDeclaration**. Cada **ClassDeclaration** está compuesto de **MethodDefinition**. Ambos, **ClassDeclaration** y **MethodDefinition**, heredan de la clase abstracta **NamedElement** (la cual le provee un nombre). Un **MethodDefinition** está compuesto por **MethodInvocation** (una llamada a un método). Cada **MethodInvocation** está asociado con un solo **MethodDeclaration** (el método llamado).

La transformación también usa una definición abstracta **Table**.

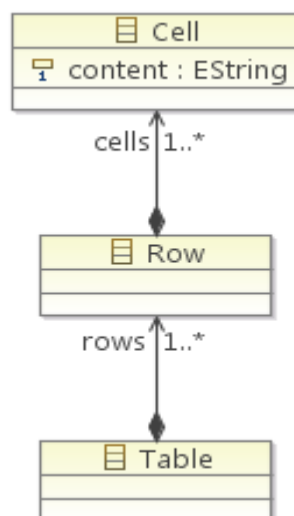


Imagen 18 - Definición Abstracta Table

En este metamodelo, una tabla está asociada a un elemento Table. Tal elemento está compuesto por Row las cuales a su vez están compuestas por Cell.

Reglas de Transformación

Las reglas para transformar de un modelo JavaSource a un modelo Table son las siguientes:

Para el elemento JavaSource, se crean los siguientes elementos:

- Un elemento Table compuesto por una secuencia de filas
- Un elemento Row, linkeado al elemento Table, que representa la primera fila de la tabla. Este elemento está compuesto de la siguiente secuencia de elementos:
 - Un elemento Cell con contenido vacío, linkeado al elemento Row, la cual es la primera celda de la primera fila.
 - Un elemento Cell, linkeado al elemento Row, por cada MethodDefinition. El contenido de la celda es igual al texto “nombre_clase.nombre_metodo”. Dentro de la secuencia las celdas están ordenadas de acuerdo a su contenido.
- Para cada MethodDefinition, los siguientes elementos son creados:
 - o Un elemento Row linkeado al elemento Table. Este elemento está compuesto por la siguiente secuencia de elementos:
 - o Un elemento Cell de título, linkeado al elemento Row correspondiente. Su contenido es el texto “nombre_clase.nombre_metodo”, donde “nombre_clase” es el nombre de la clase asociada con el MethodDefinition en cuestión, y “nombre_metodo” es el nombre del MethodDefinition en cuestión.
 - o Un elemento Cell, linkeado al elemento Row correspondiente, para cada MethodDefinition. El contenido de esta celda corresponde al número de llamadas del método en la columna dentro de la definición del método actual en la fila.

Código Fuente

El código QVT de la transformación JavaSource2Table consiste de 2 helpers y 2 mappings. El helper allMethodDefs devuelve la secuencia de todas las definiciones de métodos en todas las clases existentes. La secuencia retornada esta ordenada de acuerdo al par (class_name, method_name).

El helper computeContent devuelve el número de llamadas a un MethodDefinition en una columna, provisto como segundo parámetro del helper en cuestión, dentro del MethodDefintion que se recibe como primer parámetro del helper. Para este propósito, se seleccionan dentro de las

invocaciones del `MethodDefinition`, pasado como primer parámetro, aquellas que tienen el mismo nombre de clase y el mismo nombre de método que el `MethodDefinition` pasado como segundo parámetro. El helper luego retorna el tamaño de la secuencia creada.

El mapping `javaSource2Table` retorna la estructura del modelo `Table`, esto es, el elemento `Table` (“table”) y la primera fila de la tabla (“first_row”). Esta fila está compuesta de una primera celda vacía (“first_col”) y las celdas asociadas con los métodos declarados (“title_cols”). Luego dentro de este mapping se llama al mapping para obtener una fila a partir de la definición de un método.

El mapping `method2Row` retorna un elemento `Row` por cada `MethodDefinition` declarado. Este mapping crea un elemento `Row` (“row”) el cual está compuesto de un primer elemento celda (“title_col”) y la secuencia de celdas asociadas con cada método declarado (“othersCells”). El contenido de la secuencia de celdas se obtiene llamando al helper `computeContent` para cada elemento del conjunto devuelto por el helper `allMethodDefs`.

```
modeltype TableMetaModel uses 'TableModel';
modeltype JAVA uses 'JavaSource';
transformation JavaSource2Table(in java : JAVA, out table:TableMetaModel);
main()
{
  java.rootObjects()[JAVA::JavaSource]->map javaSource2Table();
}
mapping JAVA::JavaSource::javaSource2Table() : TableMetaModel::Table
{
  in it{
    var first_col : TableMetaModel::Cell;
    object first_col:TableMetaModel::Cell{ content := ''};

    var title_cols : OrderedSet(TableModel::Cell);

    var first_row: TableMetaModel::Row;
    object first_row:TableModel::Row{cells:= OrderedSet{}};

    first_row.cells:= allMethodDefs()-> collect( m:JAVA::MethodDefinition|
      object TableMetaModel::Cell{ content :=
        m.myClass.name.repr() + '.' + m.name.repr() } )
      -> asOrderedSet();

    if((first_row.cells -> size()) > 0) then
      first_row.cells := first_row.cells -> insertAt(1, first_col)
    else
      first_row.cells := OrderedSet{first_col}
    endif;

    var table : TableMetaModel::Table;

    object table: TableModel::Table{rows :=
      self.classes.methods -> map method2Row() -> asOrderedSet{}};

    log('table.rows -> size()', table.rows -> size());

    if((table.rows -> size()) > 0) then
      table.rows := table.rows -> insertAt(1, first_row)
    else
```

```

        table.rows := OrderedSet{first_row}
    endif;

    log('table.rows -> size() post', table.rows -> size());

    result := table;
}

}

mapping JAVA::MethodDefinition::method2Row() : TableMetaModel::Row
{
    init{
        var title_cel : TableMetaModel::Cell;
        object title_cel:TableMetaModel::Cell{ content :=
            self.myClass.name.repr() + '.' + self.name.repr() };

        var othersCells : OrderedSet(TableModel::Cell);
        var row: TableMetaModel::Row;
        object row:TableModel::Row{cells:= OrderedSet{}};
        row.cells := allMethodDefs()-> collect( m: JAVA::MethodDefinition|
            object TableMetaModel::Cell{ content := computeContent(self, m)} ) ->
            asOrderedSet();

        log('row.cells->size()', row.cells);
        if((row.cells -> size()) > 0) then
            row.cells := row.cells -> insertAt(1, title_cel)
        else
            row.cells := OrderedSet{title_cel}
        endif;

        log('row.cells->size()', row.cells);
        result := row;
    }
}

/*
Este helper devuelve la secuencia de todas las definiciones de métodos en todas las clases
existentes. La secuencia retornada esta ordenada de acuerdo a la dupla (class_name,
method_name).
*/
helper allMethodDefs(): Sequence(JAVA::MethodDefinition)
{
    return java.objectsOfType(JAVA::MethodDefinition)->asSequence()->
        sortedBy(e: JAVA::MethodDefinition | e.myClass.name.repr() + '_' +
            e.name.repr());
}

/*
Este helper devuelve el contenido de la celda de tabla asociado con el contexto de la
definición del método (fila) y el parámetro col. El valor corresponde al número de llamadas
del segundo método dentro de la definición del primer método.
*/

helper computeContent(inputMethod: JAVA::MethodDefinition, col: JAVA::MethodDefinition):
String
{
    return inputMethod.invocations
        ->select(i | i.method.name = col.name and i.method.myClass.name =
            col.myClass.name) -> size().toString();
}

```

7.4.2 IntroducePrimaryKey

Descripción

Esta transformación se aplica a cualquier clase persistente. Si la clase aún no posee una clave primaria, la transformación agrega un atributo de identidad de tipo Entero. Este es un paso esencial para la implementación de un modelo de datos en una base de datos relacional.

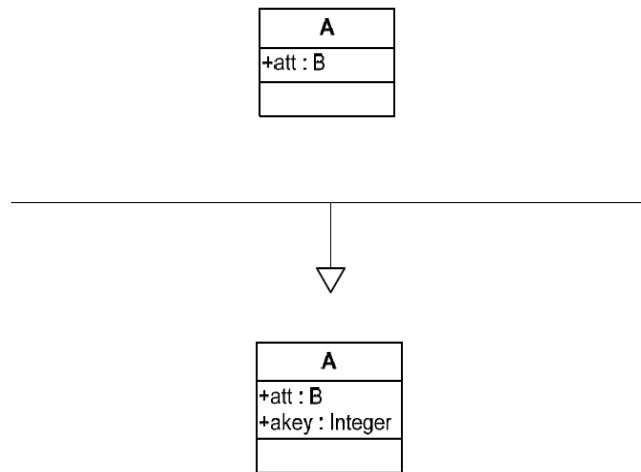


Imagen 19 - Modelo IntroducePrimaryKey

MetaModelos

SimpleUML

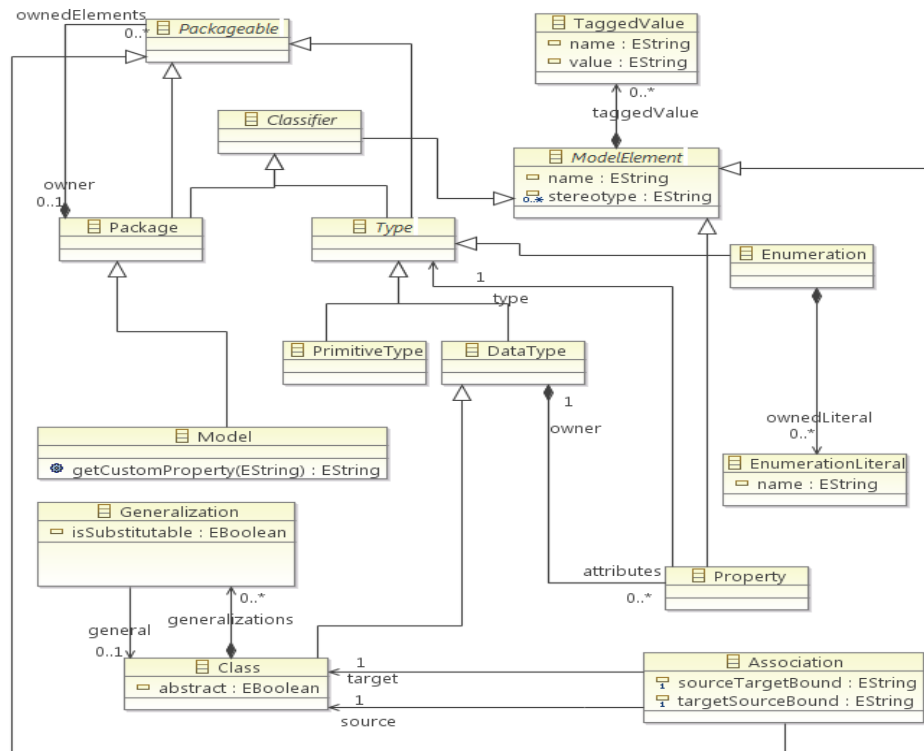


Imagen 20 - Metamodelo Para IntroducirPrimaryKey

Reglas de Transformación

Esta transformación tiene el mismo metamodelo origen y destino, simpleuml. Usamos dos nombres de modelos diferentes (uml y uml2) pero ambos se refieren al mismo metamodelo. Las reglas para transformar de un modelo uml a un modelo uml2 son las siguientes:

- Para un elemento Model, otro elemento Model es creado:
 - Con el mismo nombre.
 - Linkeando a los mismos elementos excepto las clases que son persistentes y que no tienen un atributo clave primaria.
- Para un elemento Class que no tiene un atributo clave primaria, otro elemento Class es creado:
 - Con los mismos valores en todas las propiedades del elemento Class.
 - Con una nueva propiedad con nombre nombre_clase+'key' la cual representa la clave primaria de la clase.
 - Con stereotype = 'PrimaryKey'

- Con type = 'Integer'

Código Fuente

Se considera que una propiedad de una clase es la clave primaria si dentro de su secuencia de stereotypes está el valor 'PrimaryKey'.

```
modeltype UML uses 'simpleuml';
transformation IntroducePrimaryKey(in uml : UML, out uml2:UML);
main() {
    uml.rootObjects()[UML::Model]->map model2Model();
}

/* Se asume que una propiedad que es clave primaria es aquella que tiene como
stereotype = 'primaryKey'
Se asume que las claves primarias son de tipo int y que son simples.
*/
mapping UML::Model::model2Model() : UML::Model {
    log('model2Model', self);
    name := self.name;
    ownedElements := (self.ownedElements[UML::Class]->map
    classtoClassWithPrimaryKey()->asSequence())-> union
    (self.ownedElements->select(c :UML::Packageable | c.isNotClass())-
    >asSequence());
}

mapping UML::Class::classtoClassWithPrimaryKey()
: UML::Packageable
when {not self.hasPrimaryKey()}
{
    init {
        var newProperty:UML::Property;
        var otherClass : UML::Class;

        object otherClass:UML::Class{};

        otherClass.stereotype := self.stereotype;
        otherClass.name := self.name;
        otherClass.taggedValue := self.taggedValue;
        otherClass.owner := self.owner;
        otherClass.attributes := self.attributes;
        otherClass.generalizations := self.generalizations;
        otherClass._abstract := self._abstract;

        object newProperty:UML::Property{ name := self.name + 'Key' ;
            owner := otherClass; stereotype := Set{'PrimaryKey'}
        };
        newProperty.type := object UML::DataType{ name := 'Integer';
        otherClass.attributes -> insertAt(otherClass.attributes -> size(),
            newProperty);
```

```

        result := otherClass;
    }
}

query UML::Class::hasPrimaryKey(): Boolean{
    return self.attributes -> select(isPrimaryKey())->notEmpty();
}

query UML::Property::isPrimaryKey() : Boolean {
    return self.stereotype->includes('primaryKey')
}

query UML::Packageable::isNotClass() : Boolean {
    log('isNotClass', self);
    return not self.oclIsTypeOf(UML::Class)
}

```

7.4.3 ReplaceInheritanceByAssociation

Descripción

Esta transformación reemplaza una relación de herencia entre dos clases por una asociación entre las clases. Esta transformación es muy usada cuando refinamos un PIM hacia un PSM para una plataforma, la cual no soporta herencia, como es el modelo de datos relacional. Esta también puede ser usada para eliminar la herencia múltiple en plataformas que no soportan la misma.

Cualquier expresión en el modelo original el cual tiene a B como clasificador contextual, y el cual usa una característica f heredada desde A, debería ser modificada en el nuevo modelo para que en su lugar se use como ar.f.

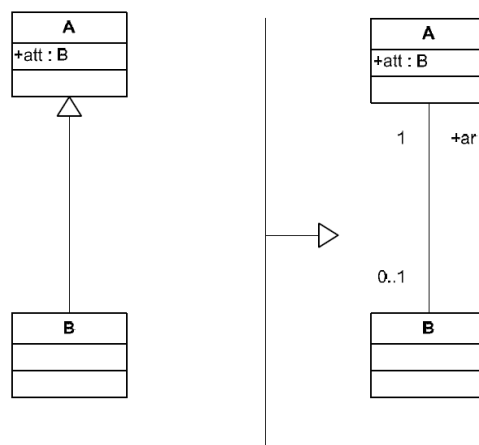


Imagen 21 - Modelo ReplaceInheritanceByAssociation

MetaModelos

SimpleUML

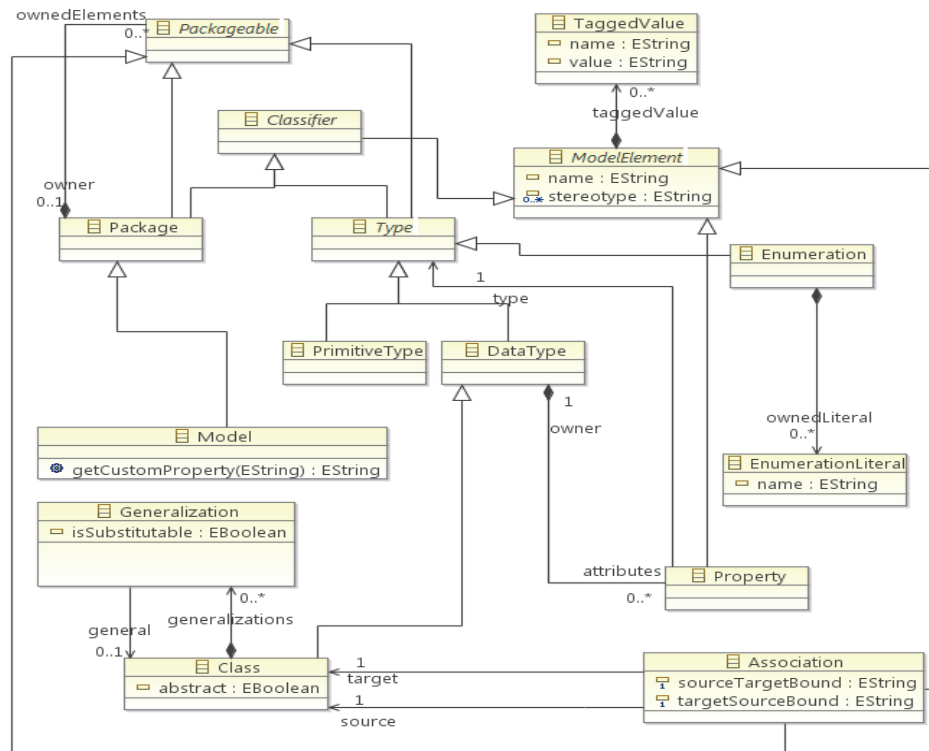


Imagen 22 - Metamodelo ReplaceInheritanceByAssociation

Reglas de Transformación

Esta transformación tiene el mismo metamodelo origen y destino, simpleuml. Las reglas para transformar de un modelo uml a un modelo uml2 son las siguientes:

- Para un elemento Model, otro elemento Model es creado:
 - Con el mismo nombre.
 - Linkeado a los mismos elementos.
- Para un elemento Class se recorren las superclases mediante la secuencia generalizations y por cada Elemento Generalization:
 - Se crea un elemento Association
 - Con nombre “nombre_claseToNombre_Superclase”
 - Source = Class
 - SourceTargetBound = 1
 - Target = Generalization.general
 - TargetSource = 0..1

Código Fuente

```

modeltype UML uses simpleuml('simpleuml');

transformation replaceInheritanceByAssociation(in uml : UML, out UML);
main() {
    uml.rootObjects()[UML::Model]->map model2Model();
}

mapping UML::Model::model2Model() : UML::Model {
    name := self.name;
    ownedElements:=self.ownedElements[UML::Class]-> map
    generalizations2Associations()->flatten()->
    union(self.ownedElements[UML::Class]-> map
    class2class()->asSequence())->
    union(self.ownedElements->select(
    c :UML::Packageable| c.isNotBecomes())->asSequence());
}

query UML::Packageable::isNotBecomes() : Boolean {
    log('isBecomes', self);
    return (not self.oclIsTypeOf(UML::Class));
}

mapping UML::Class::class2class() : UML::Packageable{
    init{
        var newClass:UML::Class;
        result:= object newClass:UML::Class{
            name := self.name;
            attributes := self.map properties2properties(self);
        };
    }
}

mapping UML::Class::properties2properties(in targetType: UML::Class) :
OrderedSet(UML::Property) {
    init {
        result := self.map dataType2properties(targetType);
    }
}

mapping UML::Class::generalizations2Associations()
:OrderedSet(UML::Packageable) {

    init{
        result:= self.generalizations ->map
        generalization2Association(self)->asOrderedSet();
    }
}

/*Mapeo una generalización a una asociación de una clase determinada.*/
mapping UML::Generalization::generalization2Association(in sourceClass:
UML::Class) : UML::Association {

```

```

//object es el constructor que me permite instanciar un modelo de
Association, al cual le agregaremos
//los datos de la generalización de clase origen, la clase origen y la
clase destino.
init{

    result:=object UML::Association{
        name:=sourceClass.name +"To"+self.general.name;
        owner:= sourceClass.owner;
        source:= sourceClass;
        sourceTargetBound := '1';
        target:=self.general;
        targetSourceBound := '0..1';
    }
}

mapping UML::Packageable::dataType2properties(in targetType : UML::DataType) :
OrderedSet(UML::Property) {

    init {
        result := targetType.attributes->map
            property2property()->asOrderedSet()
    }
}

mapping UML::Property::property2property() : UML::Property
{
    log('Property', self.type);
    name := self.name;
    stereotype := self.stereotype;
    type := self.type;
}

query UML::Package::getSubpackages() : OrderedSet(UML::Package) {

    return self.ownedElements[UML::Package]->asOrderedSet()
}

```

7.4.4 ReplaceAssociationbyForeignKey

Descripción

Esta transformación se aplica a cualquier asociación explícita “muchos a uno” entre clases persistentes. Se asume que las claves primarias existen entre las clases relacionadas por la asociación. La transformación reemplaza la asociación agregando los valores de la clave de la entidad del lado “uno” de la asociación a la entidad del lado “muchos”.

Este es un paso esencial para la implementación de un modelo de datos relacional.

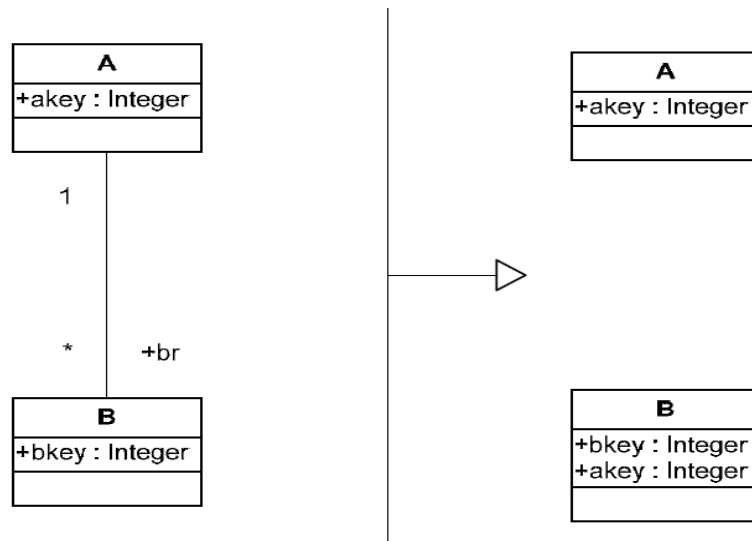


Imagen 23 – Modelo ReplaceAssociationbyForeignKey

MetaModelos

SimpleUML

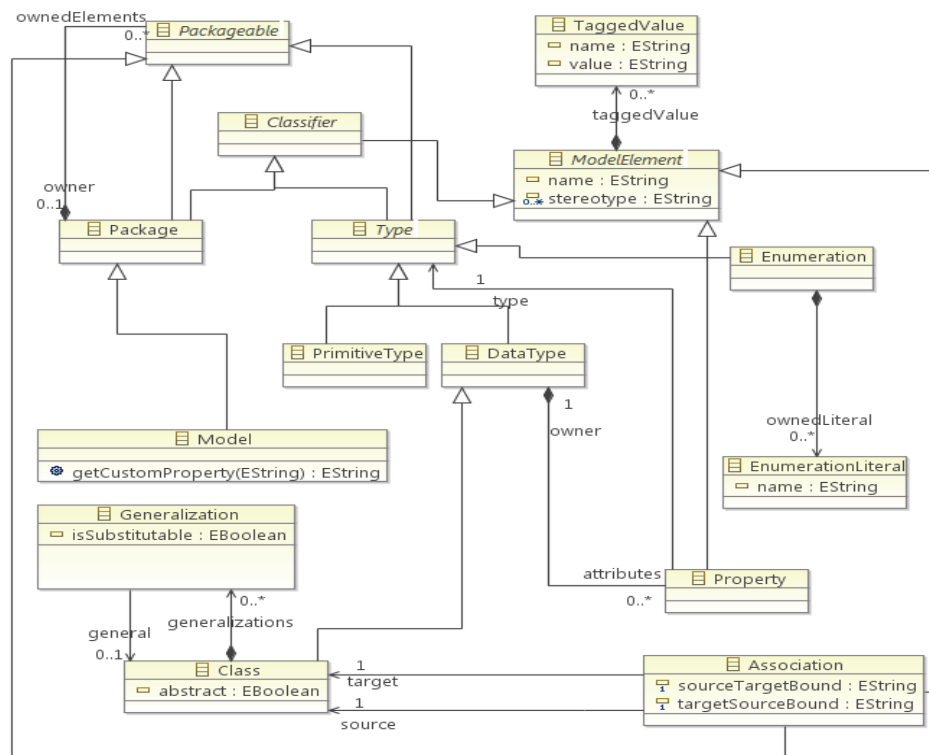


Imagen 24 - Metamodelo ReplaceAssociationbyForeignKey

Reglas de Transformación

Esta transformación tiene el mismo metamodelo origen y destino, simpleuml. Las reglas para transformar de un modelo uml a un modelo uml2 son las siguientes:

- Para un elemento Model, otro elemento Model es creado:
 - Con el mismo nombre.
 - Linkeado a los mismos elementos.
- Para un elemento Class si la misma está en una asociación de “uno a muchos” o “muchos a uno” en el lado “muchos” de la asociación, o está en una asociación “muchos a muchos”:
 - Se agrega un nuevo elemento Property
 - Con nombre = “FK”+ nombre_PropiedadPrimaryKey
 - Type = PropiedadPrimaryKey.Type
 - Stereotype = ForeignKey

Código Fuente

Se asume que una propiedad que es clave primaria es aquella que tiene como stereotype = 'primaryKey'. La transformación consiste en tomar todas las clases y por cada una ver si está en una relación en la parte N, si es así se busca la clase de la parte 1, se toma su primary key y con el mismo nombre + FK de prefijo se crea una nueva propiedad en la clase de la parte N.

```
modeltype UML uses 'simpleuml';
modeltype RDB uses 'rdb';

transformation ReplaceAssociationbyForeignKey(in uml : UML, out uml2:UML);

main() {
    uml.rootObjects()[UML::Model]->map model2Model();
}

mapping UML::Model::model2Model() : UML::Model {

    log('model2Model', self);
    name := self.name;
    ownedElements := (self.ownedElements[UML::Class]->map
        classInAssociation2ForeignKey(ownedElements[UML::Association])
        ->asSequence())-> union
        (self.ownedElements->select(c :UML::Packageable|
            c.isNotClass())->asSequence());
}

mapping UML::Class::classInAssociation2ForeignKey(elements :
OrderedSet(UML::Association) )
: UML::Packageable
{
    init
    {
        elements[classInNAssociation(self)]-> map
            addForeignKeyToClass(self);
    }
}

mapping UML::Association::addForeignKeyToClass(cla : UML::Class )
: UML:: Class
{
    init
    {
        var otherClass : UML::Class;
        otherClass = getOtherClassInAssociation(self, cla);
        var newProperty:UML::Property;
        object newProperty:UML::Property{ name := 'FK' +
            getPrimaryKey(otherClass).name ;
            owner := cla; type := getPrimaryKey(otherClass).type;
            stereotype := Set{'ForeignKey'}
        };
        cla.attributes -> insertAt(cla.attributes -> size(), newProperty);
        result := cla;
    }
}
```

```
}
```

```

helper getOtherClassInAssociation(assoc : UML::Association, cla :
UML::Class):UML::Class
{
    var claux : UML::Class;
    claux = if assoc.target <> cla then assoc.target
        else assoc.source endif;
    return claux;
}

helper getPrimaryKey( cla : UML::Class): UML::Property
{
    return cla.attributes -> select(isPrimaryKey()) ->at(0);
}

query UML::Association::classInNAssociation(cla :UML::Class) : Boolean {
    return (self.is1ToNAssociation() and self.target = cla )
    or (self.isNTto1Association() and self.source = cla )
    or (self.isNTtoNAssociation() and self.target = cla )
    or (self.isNTtoNAssociation() and self.source = cla );
}

query UML::Property::isPrimaryKey() : Boolean {
    return self.stereotype->includes('primaryKey')
}

query UML::Packageable::isNotClass() : Boolean {
    log('isNotClass', self);
    return not self.oclIsTypeOf(UML::Class)
}

query UML::Association::is1ToNAssociation() : Boolean {
    return self.sourceTargetBound = '1' and self.targetSourceBound = 'N';
}

query UML::Association::isNTto1Association() : Boolean {
    return self.sourceTargetBound = 'N' and self.targetSourceBound = '1';
}

query UML::Association::isNTtoNAssociation() : Boolean {
    return self.sourceTargetBound = 'N' and self.targetSourceBound = 'N';
}

```

7.4.5 ReplaceManyToMany2Association

Descripción

El propósito de esta transformación es sustituir asociaciones de “muchos a muchos” agregando una clase nueva con dos asociaciones “muchos a uno”. Las asociaciones “muchos a muchos” no pueden ser implementadas usando foreign keys en una base de datos relacional sino que se necesita una tabla intermedia. Esta transformación es el equivalente en OO para agregar dicha tabla.

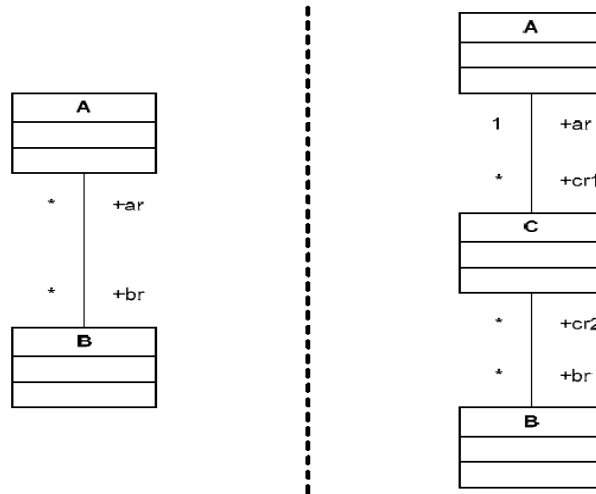


Imagen 25 - Modelo ReplaceManyToMany2Association

MetaModelos

SimpleUML

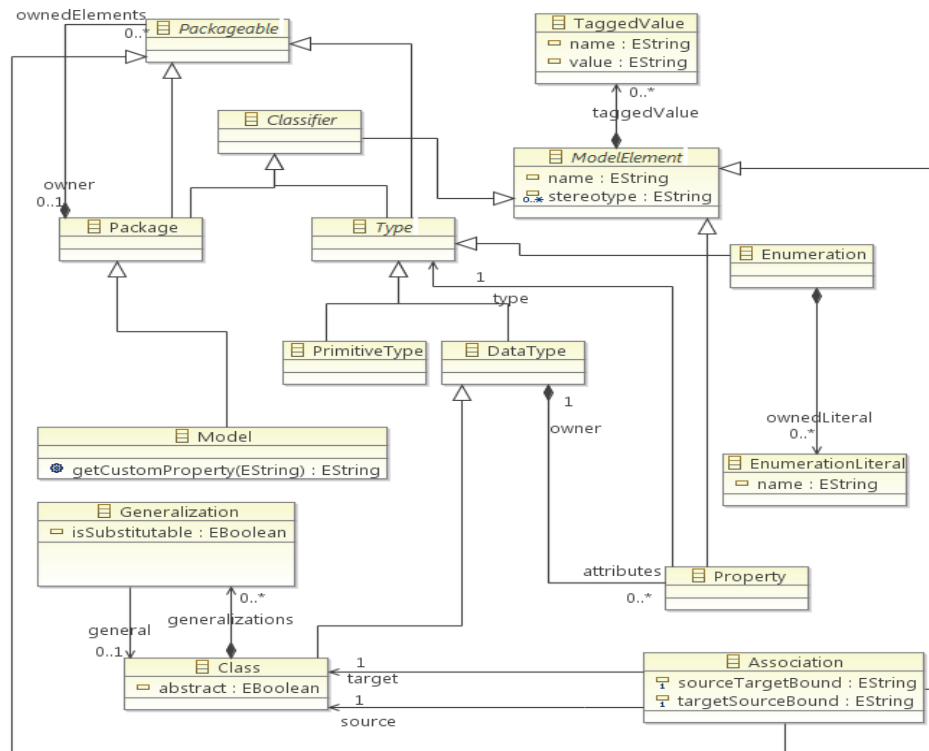


Imagen 26 - Metamodelo ReplaceManyToMany2Association

Reglas de Transformación

Esta transformación tiene el mismo metamodelo origen y destino, simpleuml. Las reglas para transformar de un modelo uml a un modelo uml2 son las siguientes:

- Para un elemento Model, otro elemento Model es creado:
 - Con el mismo nombre.
 - Linkeado a los mismos elementos.
- Para un elemento Packageable, si no es un elemento Association o si lo es pero no es una asociación de “muchos a muchos”, un elemento Packageable es creado:
 - Con el mismo nombre.
 - Linkeado a los mismos elementos.
- Para un elemento Association que representa una asociación de “muchos a muchos”, se crean los siguientes elementos:
 - Un elemento Class
 - Con nombre NombreClaseOrigen NombreClaseDestino

- Un elemento Association
 - Con nombre NombreClaseOrigenTo
NombreClaseOrigen_NombreClaseDestino
 - Con sourceTargetBound = 1
 - Con Source = ClaseOrigen del elemento Association de entrada para el mapping
 - Con targetSourceBound = N
 - Con Target = ClaseNueva
- Un elemento Association
 - Con nombre NombreClaseOrigen_NombreClaseDestino
ToNombreClaseDestino
 - Con sourceTargetBound = 1
 - Con Source = ClaseNueva
 - Con targetSourceBound = N
 - Con Target = ClaseDestino del elemento Association de entrada para el mapping

Código Fuente

```

modeltype UML uses 'simpleuml';
modeltype RDB uses 'rdb';

transformation ReplaceManyToMany2Association(in uml : UML, out uml2:UML);

main() {
    uml.rootObjects()[UML::Model]->map model2Model();
}

mapping UML::Model::model2Model() : UML::Model {
    log('model2Model', self);
    name := self.name;
    ownedElements := (self.ownedElements[UML::Association]->map
        associationN2N2associationN2N()->flatten()-> union
        (self.ownedElements->select(c :UML::Packageable|
            c.isNotAssociationNToN()->asSequence()));
}

mapping UML::Association::associationN2N2associationN2N()
    : OrderedSet(UML::Packageable)
when{ self.isNToNAssociation() }
{
    init
    {
        log('Entro associationN2N2associationN2N', self);
        var newClass:UML::Class;
    }
}

```

```

var assoc1:UML::Association;
var assoc2:UML::Association;
object newClass:UML::Class{ name := self.source.name + '_' +
    self.target.name};

object assoc1:UML::Association{ name := self.source.name + 'To' +
    newClass.name ;sourceTargetBound:='1';targetSourceBound:='N'
};
assoc1.source := self.source;
assoc1.target := newClass;
assoc1.taggedValue := self.taggedValue;
assoc1.stereotype := self.stereotype;
assoc1.owner := self.owner;

object assoc2:UML::Association{ name := newClass.name + 'To' +
    self.target.name ;sourceTargetBound:='N';targetSourceBound:='1'
};
assoc2.source := self.target;
assoc2.target := newClass;
assoc2.taggedValue := self.taggedValue;
assoc2.stereotype := self.stereotype;
assoc2.owner := self.owner;

result := OrderedSet{newClass, assoc1, assoc2};
log('RESULT associationN2N2associationN2N', result);
}
}

query UML::Packageable::isNotAssociationNTon() : Boolean {
    log('Prueba', self);
    return (self.oclIsTypeOf(UML::Association) and
not(self.oclAsType(UML::Association)).isNTonAssociation())
    or (not self.oclIsTypeOf(UML::Association));
}

query UML::Association::isNTonAssociation() : Boolean {
    return self.sourceTargetBound = 'N' and self.targetSourceBound = 'N';
}

```

7.4.6 RemoveAssociationClass

Descripción

Esta transformación reemplaza una clase asociación con una nueva clase con dos asociaciones. Las clases asociación no pueden ser representadas en muchos lenguajes de programación orientados a objetos.

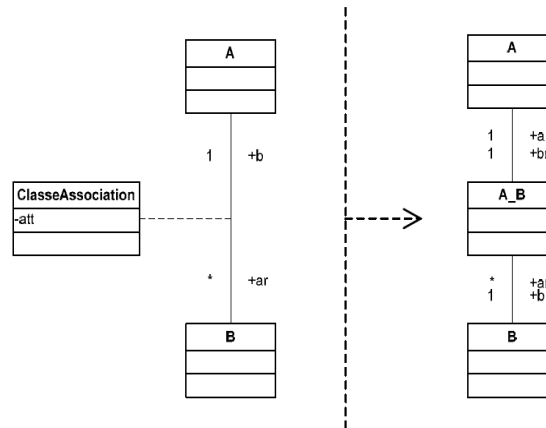


Imagen 27 - Modelo RemoveAssociationClass

MetaModelos

Para esta transformación es necesario modificar el metamodelo SimpleUML dado que en el mismo no se define un elemento AssociationClass, por ello se crea el metamodelo SimpleUMLToRemoveAssociationClass y en el mismo se define el elemento AssociationClass que hereda de DataType para tener la colección de atributos y de Association para tener las clases referenciadas y el bound de cada referencia.

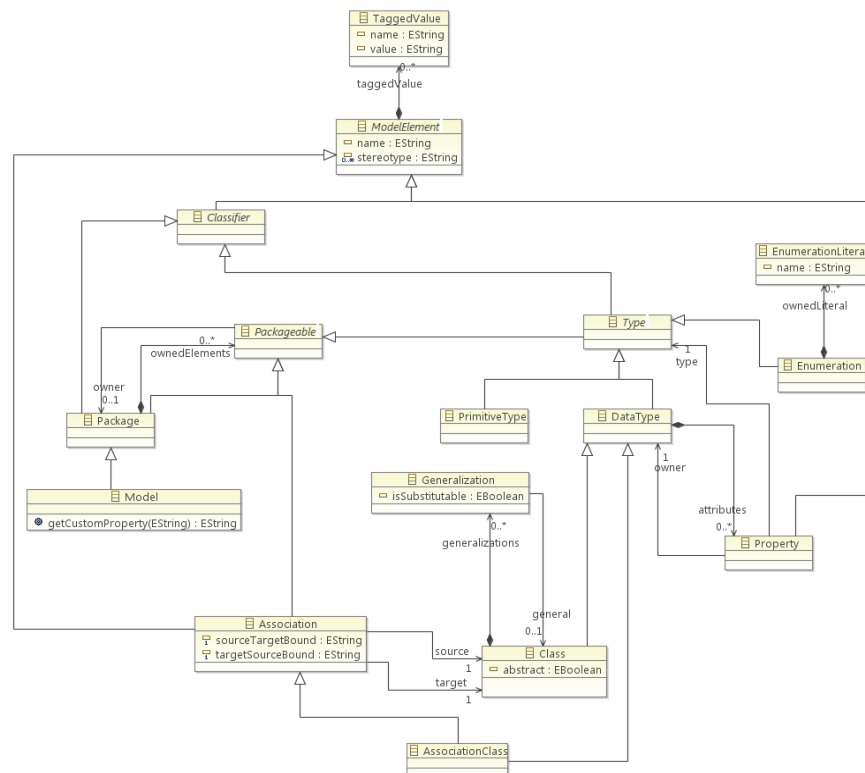


Imagen 28 - Metamodelo RemoveAssociationClass

Reglas de Transformación

Esta transformación tiene el mismo metamodelo origen y destino, simpleumlRAC. Las reglas para transformar de un modelo uml a un modelo uml2 son las siguientes:

- Para un elemento Model, otro elemento Model es creado:
 - Con el mismo nombre.
 - Linkeado a los mismos elementos.
- Para un elemento Packageable otro elemento Packageable es creado:
 - Con el mismo nombre.
 - Linkeado a los mismos elementos.
- Para un elemento AssociationClass se crean los siguientes elementos:
 - Un elemento Class
 - Con nombre igual al del elemento AssociationClass
 - Un elemento Association
 - Con nombre NombreClaseOrigenToNombreAssociationClass
 - Con sourceTargetBound = 1
 - Con Source = ClaseOrigen del elemento AssociationClass de entrada para el mapping
 - Con targetSourceBound = targetSourceBound del elemento AssociationClass de entrada para el mapping
 - Con Target = ClaseNueva
 - Un elemento Association
 - Con nombre NombreClaseDestinoToNombreAssociationClass
 - Con sourceTargetBound = 1
 - Con Source = ClaseDestino del elemento AssociationClass de entrada para el mapping
 - Con targetSourceBound = targetSourceBound del elemento AssociationClass de entrada para el mapping
 - Con Target = ClaseNueva

Código Fuente

Supongamos que tenemos dos clases A y B, y la clase asociación entre ambas llamada AB. A es el source de AB y B es el target. El objetivo es crear una nueva clase llamada AB, y dos nuevas asociaciones donde cada asociación va a tener como target a AB y como source a A o a B según corresponda, y el nombre de cada asociación será AToAB y BToAB. Luego los bound serán siempre 1 para el source y el AToAB.targetBound = AB.sourceBound y BToAB.targetBound = AB.targetBound.

```
modeltype UML uses 'simpleumlRAC';
modeltype RDB uses 'rdb';
```

```
transformation RemoveAssociationClass(in uml : UML, out uml2:UML);
```

```

main() {

    uml.rootObjects()[UML::Model]->map model2Model();
}

mapping UML::Model::model2Model() : UML::Model {

    name := self.name;
    ownedElements := (self.ownedElements->select(c :UML::Packageable|
        c.isNotAssociationClass())->asSequence())->union
        (self.ownedElements[UML::AssociationClass]->map
            associationClass2Class()->flatten());
}

mapping UML::AssociationClass::associationClass2Class()
: OrderedSet(UML::Packageable)
{
    init
    {
        var newClass:UML::Class;
        var assoc1:UML::Association;
        var assoc2:UML::Association;
        object newClass:UML::Class{ name := self.name};

        object assoc1:UML::Association{ name := self.source.name + 'To' +
            self.name};
        assoc1.source := self.source;
        assoc1.target := newClass;
        assoc1.sourceTargetBound := '1';
        assoc1.targetSourceBound := self.sourceTargetBound;

        object assoc2:UML::Association{ name := self.target.name + 'To' +
            self.name};
        assoc2.source := self.target;
        assoc2.target := newClass;
        assoc2.sourceTargetBound := '1';
        assoc2.targetSourceBound := self.sourceTargetBound;

        newClass.attributes := self.attributes;

        result := OrderedSet{newClass, assoc1, assoc2};
    }
}

query UML::Packageable::isNotAssociationClass() : Boolean {
    return not self.ocLIsTypeOf(UML::AssociationClass);
}

```

7.4.7 RemoveRedundantClass

Descripción

Las clases pueden ser redundantes porque son, esencialmente, duplicados de otras clases en el modelo pero con diferente nombre (Sinónimos), o porque no son necesarias en el sistema que estamos definiendo.

Las clases duplicadas podrían llevar a modelos más complejos los cuales son difíciles de modificar y analizar.

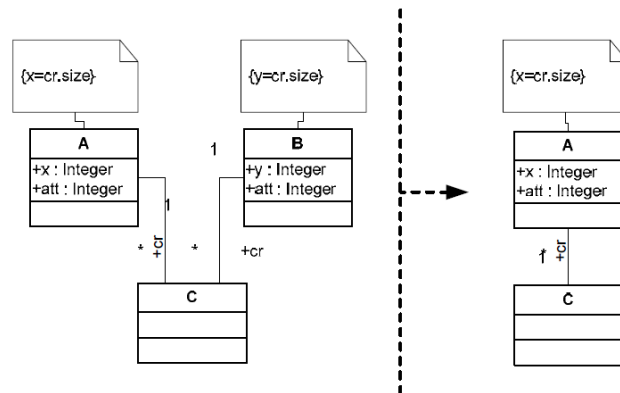


Imagen 29 - Modelo RemoveRedundantClass

MetaModelos

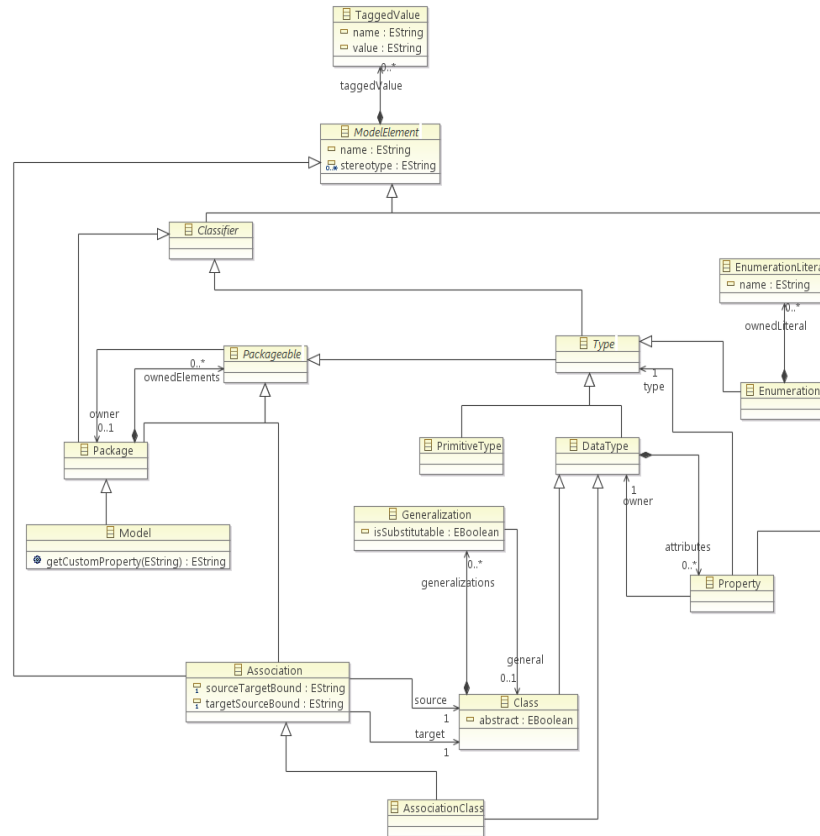


Imagen 30 - Metamodelo RemoveRedundantClass

Reglas de Transformación

Esta transformación tiene el mismo metamodelo origen y destino, `simpleumlRAC`. Las reglas para transformar de un modelo `uml` a un modelo `uml2` son las siguientes:

- Para un elemento Model, otro elemento Model es creado:
 - Con el mismo nombre.
 - Linkeado a los mismos elementos.
- Para un elemento Packageable otro elemento Packageable es creado:
 - Con el mismo nombre.
 - Linkeado a los mismos elementos.
- Para un elemento Class, si la misma no es una clase redundante, se crea un elemento Class con los mismos valores, excepto que para la secuencia de superclases para las cuales se aplica la misma regla.

La definición de una clase redundante es subjetiva y es necesario adaptar los criterios de acuerdo al contexto.

Un elemento Class es considerado redundante si otro elemento Class tiene las mismas propiedades. Así que antes de generar el modelo resultado, los elementos Class redundantes son buscados y almacenados en un conjunto.

Código Fuente

La idea de esta transformación es primero armar un conjunto con aquellas clases que son redundantes. Se va a considerar que una clase es redundante si no se usa en todo el modelo o si hay una clase sinónimo, es decir, que es igual, con las mismas propiedades y características, pero con otro nombre. Por ello, si hay N clases sinónimos sólo debe quedar una y se deben cambiar todas las referencias a las clases sinónimos que se eliminan para que referencien a la clase que va a quedar.

El helper `getRedundantClasses`, dado el conjunto de clases del modelo origen, arma un conjunto de clases que son redundantes. Para determinar si una clase es redundante se aplica el query `isRedundantClass`.

El query `isRedundantClass`, dado una clase, devuelve verdadero si la clase es redundante. Una clase es redundantes si:

- Tiene la misma cantidad de atributos y de los mismos tipos.
- Tiene los mismos nombres de `taggedValue`
- Tiene las mismas superclases.

```
modeltype UML uses 'simpleumlRAC';
modeltype RDB uses 'rdb';

transformation RemoveRedundantClass(in uml : UML, out uml2:UML);

main() {
    uml.rootObjects()[UML::Model]->map model2Model();
}

mapping UML::Model::model2Model() : UML::Model {
    log('model2Model', self);
    var redundantClasses : Set(UML::Class);
    redundantClasses := getRedundantClasses(self.ownedElements[UML::Class]);
    name := self.name;
    ownedElements := (self.ownedElements[UML::Class]->map
        classNotRedundant2Class(redundantClasses)->asSequence())-> union
        (self.ownedElements->select(c :UML::Packageable|
            c.isNotClass())->asSequence());
}
```

```

mapping UML::Class::classNotRedundant2Class(redundantClasses :
Set(UML::Class)):UML::Packageable
    when { redundantClasses-> excludes(self) }
{
    init
    {
        result := self.map class2class(redundantClasses);
    }
}

mapping UML::Class::class2class(redundantClasses : Set(UML::Class)):UML::Class
{
    name := self.name;
    stereotype := self.stereotype;
    taggedValue := self.taggedValue;
    _abstract := self._abstract;
    generalizations:= self.generalizations -> select(gen
        :UML::Generalization| redundantClasses-> excludes(gen.general));
    attributes := self.attributes;
}

query UML::Packageable::isNotClass() : Boolean {
    log('Prueba', self);
    return not self.oclIsTypeOf(UML::Class);
}

helper getRedundantClasses( classes : OrderedSet(UML::Class)): Set(UML::Class)
{
    var checkedClasses: Sequence(String) = Sequence{};
    UML::Class.allInstances()->iterate(
        r : UML::Class ;
        rs : Sequence(UML::Class) = Sequence{}
    | UML::Class.allInstances()->iterate(
        s : UML::Class ;
        rs1 : Sequence(UML::Class) = rs
        |
        if(isRedundantClass(r,s,checkedClasses))
            then rs1->append(s)
            else rs1
            endif
        )
    );
    return Set{};
}

query isRedundantClass( c1: UML::Class, c2: UML::Class, checkedClasses:
Sequence(String)): Boolean
{
    if
        (c1<>c2
        and c1.attributes ->size() = c2.attributes->size()
        and c1.taggedValue->collect(a|a.name)->asSet() = c2.taggedValue-
            >collect(a|a.name)->asSet())

```

```

    and c1.generalizations->collect(a|a.general.name)->asSet() =
      c2.generalizations->collect(a|a.general.name)->asSet()
    and c1.attributes->collect(a|a.type)->asSet() = c2.attributes-
      >collect(a|a.type)->asSet()
    and (not isAlreadyConsidered(c1, c2, checkedClasses))
  )
  then

    return true
  endif;
return false;
}

query isAlreadyConsidered( c1: UML::Class, c2: UML::Class, checkedClasses:
Sequence(String)): Boolean
{
  if(checkedClasses->includes(c2.name+'____'+c1.name))
    then return true
  else
    {
      checkedClasses-> append(c1.name+'____'+c2.name);
      return false
    }
  endif;
}

```

7.4.8 UMLObserver2JavaObserver

Descripción

Esta transformación adapta un modelo SimpleUML a un modelo SimpleUMLO2JO. Agrega al modelo origen la infraestructura para implementar las interfaces Observer y Observable.

Para generar los métodos de las interfaces Observer y Observable de java nos basaremos en la definición de la API 1.4.2 de java

MetaModelos

Para esta transformación es necesario modificar el SimpleUML dado que en tal metamodelo no se define el elemento Method y el elemento Parameter, por ello se crea el metamodelo SimpleUMLToUMLObserver2JavaObserver y en el mismo se define el elemento Method que tiene una colección de parámetros (Elemento Parameter).

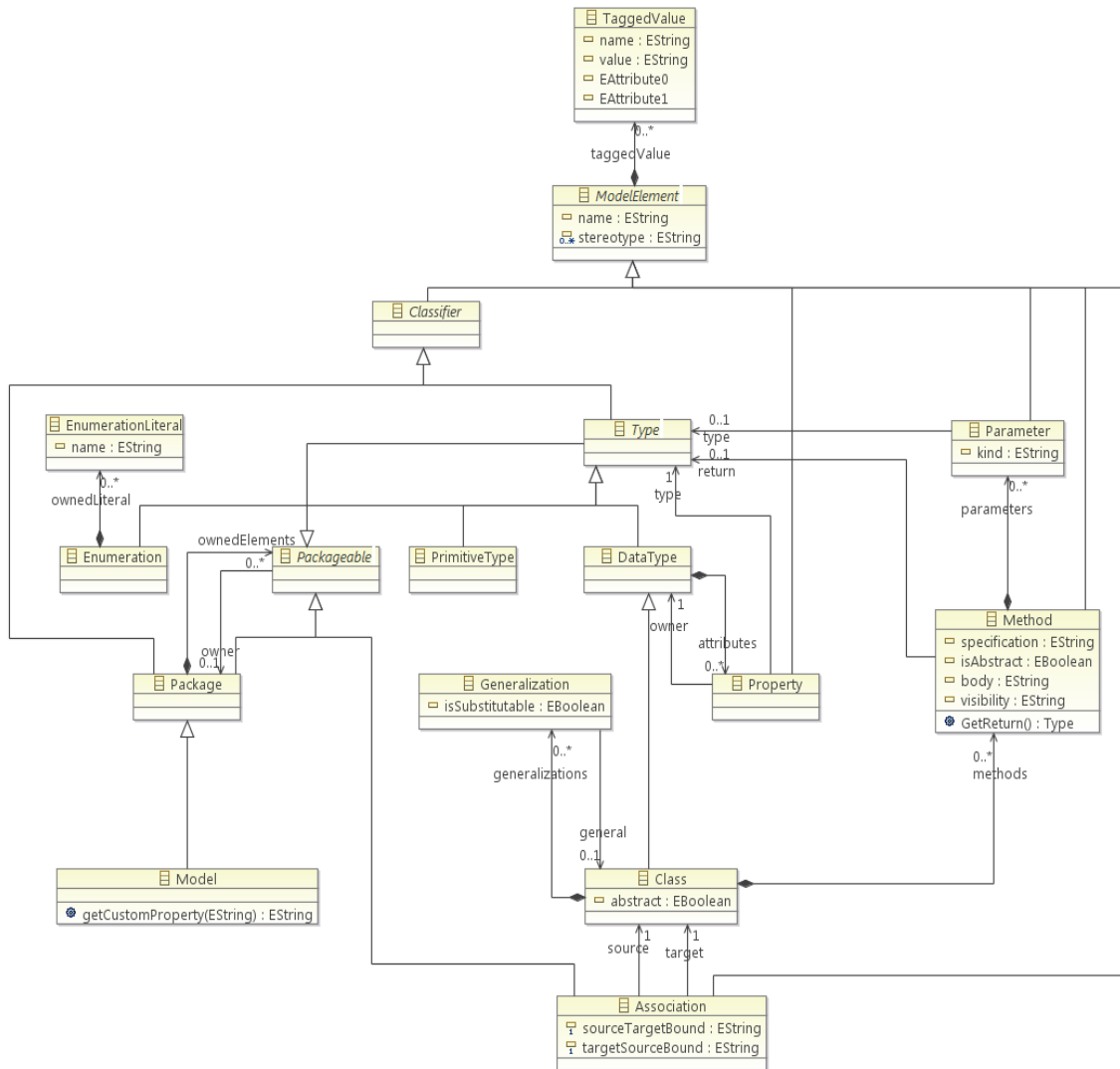


Imagen 31 - Metamodelo UMLObserver2JavaObserver

Reglas de Transformación

Esta transformación tiene un metamodelo origen SimpleUML y un metamodelo destino SimpleUMLRAC. Las reglas para transformar de un modelo uml a un modelo uml2 son las siguientes:

- Para un elemento Model, otro elemento Model es creado:
 - Con el mismo nombre.
 - Linkeado a los mismos elementos.
- Para un elemento Packageable otro elemento Packageable es creado:
 - Con el mismo nombre.
 - Linkeado a los mismos elementos.
- Para un elemento Class, si tiene el Stereotype Observable, se crea un elemento Class:
 - Con el mismo nombre, atributos, superclases, métodos.
 - Con Stereotype "Observable"
 - Se agrega un nuevo elemento Property:
 - Con nombre observer
 - De tipo Vector
 - Se agrega un nuevo elemento Property:
 - Con nombre changed
 - De tipo Boolean
 - Se agrega un nuevo elemento Method:
 - Con nombre addObserver
 - Con visibility = "public"
 - Con return de tipo void
 - Con body = "observers.addElement(observer);"
 - Se crea un nuevo elemento Parameter
 - Con nombre observer
 - De tipo Observer
 - De clase IN
 - Se agrega un nuevo elemento Method:
 - Con nombre deleteObserver
 - Con visibility = "public"
 - Con return de tipo void
 - Con body = "observers.removeElement(observer);"
 - Se crea un nuevo elemento Parameter
 - Con nombre observer
 - De tipo Observer
 - De clase IN
 - Se crea un nuevo Se agrega un nuevo elemento Method:
 - Con nombre notifyObservers
 - Con visibility = "public"

- Con return de tipo void
 - Con body = “ if(hasChanged){
 Iterator i = observers.iterator();
 while (i.hasNext()) {
 Observer o = (Observer) i.next();
 o.update(name, value);
 }
 }”
 - Se crea un nuevo elemento Parameter
 - Con nombre name
 - De tipo String
 - De clase IN
 - Se crea un nuevo elemento Parameter
 - Con nombre value
 - De tipo Object
 - De clase IN
- Se agrega un nuevo elemento Method:
 - Con nombre setChanged
 - Con visibility = “public”
 - Con return de tipo void
 - Con body = “changed := true;
- Se agrega un nuevo elemento Method:
 - Con nombre hasChanged
 - Con visibility = “public”
 - Con return de tipo boolean
 - Con body = “return changed;”
- Se crea un nuevo Se agrega un nuevo elemento Method:
 - Con nombre clearChanged
 - Con visibility = “public”
 - Con return de tipo void
 - Con body = “ changed := false;”
- Para un elemento Class, si tiene el Stereotype Observer, se crea un elemento Class:
 - Con el mismo nombre, atributos, superclases, métodos.
 - Con Stereotype “Observer”
 - Se agrega un nuevo elemento Method:
 - Con nombre update
 - Con visibility = “public”
 - Con return de tipo void
 - Con body = “
 if (name.equals(“PropertyA”))
 {
 onPropertyAChanged(value);
 }
 if (name.equals(“PropertyB”))

```

    { onPropertyBChanged(value);
    }
    .
    .
    .
    if (name.equals("PropertyZ"))
    {
        onPropertyZChanged(value);
    }
    "

```

- Se crea un nuevo elemento Parameter
 - Con nombre name
 - De tipo String
 - De clase IN

Código Fuente

Se considera que una clase o alguna de sus clases padres es Observable si algunos de sus stereotypes = 'Observable'. Se considera que una clase o alguna de sus clases padres es Observer si algunos de sus stereotypes = 'Observer'. Para generar los métodos de las interfaces Observer y Observable de java nos basaremos en la definición de la API 1.4.2 de java.

```

modeltype UMLSource uses 'simpleuml';
modeltype UML uses 'simpleumlO2JO';

transformation UMLObserver2JavaObserver(in uml : UMLSource, out uml2:UML);

main() {
    uml.rootObjects()[UML::Model]->map model2Model();
}

mapping UML::Model::model2Model() : UML::Model {
    log('model2Model', self);
    name := self.name;
    ownedElements := (self.ownedElements[UML::Class]->map
        classObservable2JavaClassObservable()->asSequence()-> union
        (self.ownedElements[UML::Class]->map
            classObserver2JavaClassObserver()->asSequence()-> union
            (self.ownedElements->select(c :UML::Packageable|
                c.isNotClass()->asSequence());
    }

mapping UML::Class::classObservable2JavaClassObservable():UML::Packageable
    when { self.isObservableKind() }
{
    init
    {
        result := self.map classObservable2JavaClassObservableIntern();
    }
}

```

```

}

mapping UML::Class::classObservable2JavaClassObservableIntern():UML::Class
{
    name := self.name;
    stereotype := self.stereotype;
    taggedValue := self.taggedValue;
    _abstract := self._abstract;
    generalizations:= self.generalizations;
    attributes := self.attributes;
    methods := self.methods;
    // Agrega La Propiedad observers
    var observersAttribute:UML::Property;
    object observersAttribute:UML::Property{ name:= 'observers'; type :=
        object UML::DataType {name := 'java.util.Vector' } };
    attributes := self.attributes ->append(observersAttribute);

    // Agrega La Propiedad privada changed
    var changedAttribute:UML::Property;
    object changedAttribute:UML::Property{ name:= 'changed'; type := object
        UML::DataType {name := 'Boolean' } };
    attributes := self.attributes ->append(changedAttribute);

    // Agrega el metodo addObserver
    var paramAddObserverMethod: UML::Parameter;
    object paramAddObserverMethod:UML::Parameter{
        name := 'observer';
        type:= object UML::DataType {name := 'Observer' };
        kind := 'IN' };
    var addObserverMethod:UML::Method;
    object addObserverMethod:UML::Method{ name:= 'addObserver';
        parameters := OrderedSet{paramAddObserverMethod};
        visibility := 'public';
        _return := object UML::DataType {name := 'void' };
        _body := 'observers.addElement(observer)';
    };
    methods := self.methods ->append(addObserverMethod);

    // Agrega el metodo deleteObserver
    var paramDeleteObserverMethod: UML::Parameter;
    object paramDeleteObserverMethod:UML::Parameter{
        name := 'observer';
        type:= object UML::DataType {name := 'Observer' };
        kind := 'IN' };

    var deleteObserverMethod:UML::Method;
    object deleteObserverMethod:UML::Method{ name:= 'deleteObserver';
        parameters := OrderedSet{paramDeleteObserverMethod};
        visibility := 'public';
        _return := object UML::DataType {name := 'void' };
        _body := 'observers.removeElement(observer)';
    };
    methods := self.methods ->append(deleteObserverMethod);
}

```

```

// Agrega el metodo notifyObserver(name, value)

var paraNameNotifyObserversMethod: UML::Parameter;
object paraNameNotifyObserversMethod:UML::Parameter{
    name := 'name';
    type:= object UML::DataType {name := 'String' };
    kind := 'IN' };

var paraValueNotifyObserversMethod: UML::Parameter;
object paraValueNotifyObserversMethod:UML::Parameter{
    name := 'value';
    type:= object UML::DataType {name := 'Object' };
    kind := 'IN' };

var notifyObserversMethod:UML::Method;
object notifyObserversMethod:UML::Method{ name:= 'notifyObservers';
    parameters := OrderedSet{paraNameNotifyObserversMethod,
        paraValueNotifyObserversMethod};
    visibility := 'public';
    _return := object UML::DataType {name := 'void' };
    _body := '
        if(hasChanged(){
            Iterator i = observers.iterator();
            while (i.hasNext()) {
                Observer o = (Observer) i.next();
                o.update(name, value);
            }
        }';
};
methods := self.methods ->append(notifyObserversMethod);

// Agrega el metodo setChanged()
var setChangedMethod:UML::Method;
object setChangedMethod:UML::Method{ name:= 'setChanged';
    parameters := OrderedSet{};
    visibility := 'public';
    _return := object UML::DataType {name := 'void' };
    _body := ' changed := true;';
};
methods := self.methods ->append(setChangedMethod);
// Agrega el metodo hasChanged()

var hasChangedMethod:UML::Method;
object hasChangedMethod:UML::Method{ name:= 'hasChanged';
    parameters := OrderedSet{};
    visibility := 'public';
    _return := object UML::DataType {name := 'Boolean' };
    _body := 'return changed;';
};
methods := self.methods ->append(hasChangedMethod);
// Agrega el metodo clearChanged()

var clearChangedMethod:UML::Method;
object clearChangedMethod:UML::Method{ name:= 'clearChanged';
    parameters := OrderedSet{};

```

```

        visibility := 'public';
        _return := object UML::DataType {name := 'void' };
        _body := 'changed := false;';
    };
    methods := self.methods ->append(clearChangedMethod);
}

mapping UML::Class::classObserver2JavaClassObserver():UML::Packageable
    when { self.isObservableKind() }
{
    init
    {
        result := self.map classObserver2JavaClassObserverIntern();
    }
}

mapping UML::Class::classObserver2JavaClassObserverIntern():UML::Class
{
    name := self.name;
    stereotype := self.stereotype;
    taggedValue := self.taggedValue;
    _abstract := self._abstract;
    generalizations := self.generalizations;
    attributes := self.attributes;
    methods := self.methods;

    // Agrega el metodo update(String name, Object value)
    var paramNameUpdateMethod: UML::Parameter;
    object paramNameUpdateMethod:UML::Parameter{
        name := 'name';
        type:= object UML::DataType {name := 'String' };
        kind := 'IN' };

    var paramValueUpdateMethod: UML::Parameter;
    object paramValueUpdateMethod:UML::Parameter{
        name := 'value';
        type:= object UML::DataType {name := 'Object' };
        kind := 'IN' };

    var updateMethod:UML::Method;
    object updateMethod:UML::Method{ name:= 'update';
        parameters :=
            OrderedSet{paramNameUpdateMethod,paramValueUpdateMethod};
        visibility := 'public';
        _return := object UML::DataType {name := 'void' };
        _body := updateBody(self);
    };
    methods := self.methods ->append(updateMethod);
}

helper handlers(cla :UML::Class ) : Sequence(UML::Method)
{
    return      cla.methods ->select(f| f.name.startsWith('on') and
f.name.endsWith('Change') and f.name.size() > 8)

```

```

->asSequence()
};

helper updateBody(cla :UML::Class ) : String
{
    return handlers(cla)->iterate(o; acc : String = ''|acc +
updateBody(o,'value'));
};

helper propertyName(method:UML::Method) : String
{
    return method.name.substring(3, method.name.size() - 6); --
onPropertyChange -> Property
}

helper javaBoxedType( type : String) : String
{
    if type = 'OclAny' then
        return 'Object'
    else if type = 'Natural' then
        return 'Long'
    else if type = 'UnlimitedNatural' then
        return 'Integer'
    else if type = 'Real' then
        return 'Double'
    else
        return type
    endif endif endif endif;
    return type;
}

helper javaUnboxing(type : String) : String
{
    if type = 'Boolean' then
        return '(' + type + ').booleanValue()'
    else if type = 'Integer' then
        return '(' + type + ').intValue()'
    else if type = 'Natural' then
        return '(' + type + ').longValue()'
    else if type = 'UnlimitedNatural' then
        return '(' + type + ').intValue()'
    else if type = 'Real' then
        return '(' + type + ').doubleValue()'

    endif endif endif endif endif;
    return type;
}

helper javaTypedValue(value : String, type : String) : String =
    return '(' + javaBoxedType(type) + ')' + value;

helper javaUnboxValue( value :String ,type : String) : String
{
    return javaUnboxing(javaTypedValue(value , type));
}

```

```

}

helper parTypeName(method:UML::Method) : String
{
    return method.parameters->first().type.name;
}

helper updateBody(method:UML::Method , value :String) : String
{
    return
        '\nif (name.equals("'" + propertyName(method) + '"')) {\n' +
        '    ' + method.name + '(' + javaUnboxValue(value, parTypeName( method))
+ '); \n' +
        '}'';
};

query UML::Packageable::isNotClass() : Boolean {
    log('Prueba', self);
    return not self.oclIsTypeOf(UML::Class);
}

query UML::Class::isObservableType() : Boolean
{
    not self.stereotype->select(s|s ='Observable')->isEmpty();
}

query UML::Class::isObservableKind() : Boolean
{
    if self.isObservableType() then
        true
    else
        not self.generalizations->select(
            g|g.general.isObservableKind())->isEmpty()
        endif;
}

query UML::Class::isObserverType() : Boolean
{
    not self.stereotype->select(s|s ='Observer')->isEmpty();
}

query UML::Class::isObserverKind() : Boolean
{
    if self.isObservableType() then
        true
    else
        not self.generalizations->
            select(g|g.general.isObserverKind())->isEmpty()
        endif;
}

```

7.4.9 UML2TestUML

Descripción

Esta transformación es una adaptación de la transformación presentada en la tesis “Testing basado en modelos: especificación gráfica y derivación de código fuente”[56]. La transformación presentada en dicha tesis toma como modelo de entrada una instancia del metamodelo UML+U2TP y genera como resultado un modelo instancia del metamodelo U2TP que sólo contiene los elementos de testing. El metamodelo origen está generado como un perfil UML (UML Profile) que es un mecanismo de extensión provisto por UML para modelar los conceptos de ciertos dominios particulares. Un perfil es un paquete de UML que agrega nuevos constructores al metamodelo de UML.

La presente transformación se basa en que el metamodelo Origen no está implementado con el perfil UML sino que el metamodelo origen es simplemente UML. Por compatibilidad con el resto de las transformaciones presentadas en esta tesis, se tomará como metamodelo de origen una derivación del SimpleUML. En particular, como un perfil en definitiva lo que hace es agregar elementos Stereotypes a un metamodelo UML, para simular el perfil lo que vamos a hacer es usar la propiedad Stereotypes que tienen todos los elementos del metamodelo SimpleUML.

MetaModelos

SimpleUMLToUMLObserver2JavaObserver

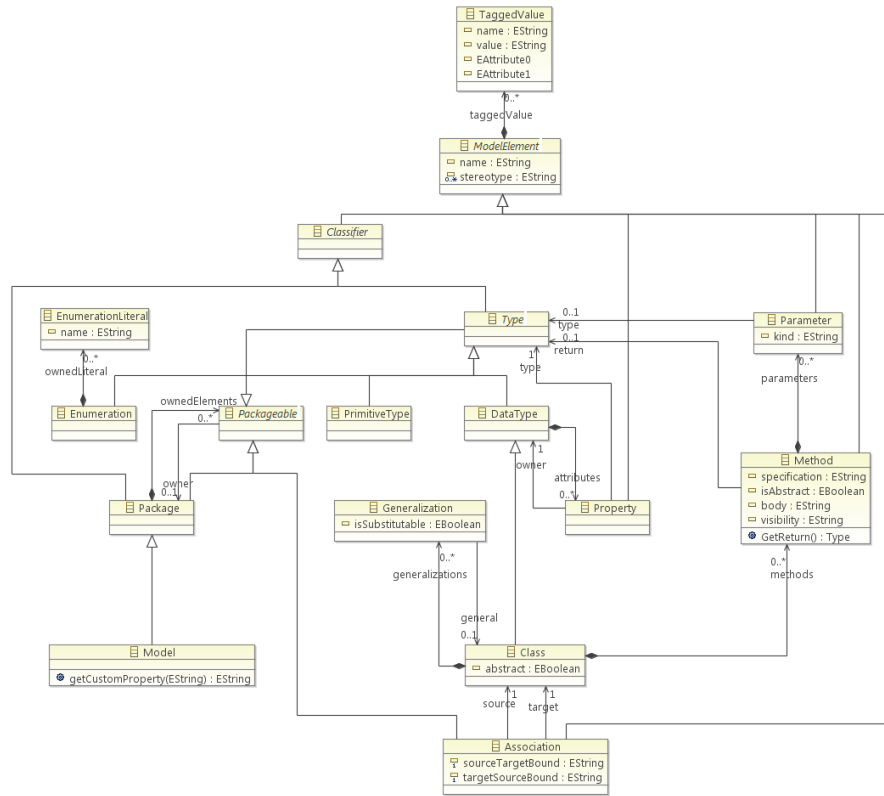


Imagen 32 – Metamodelo1 UML2TestUML

UML2TestUML

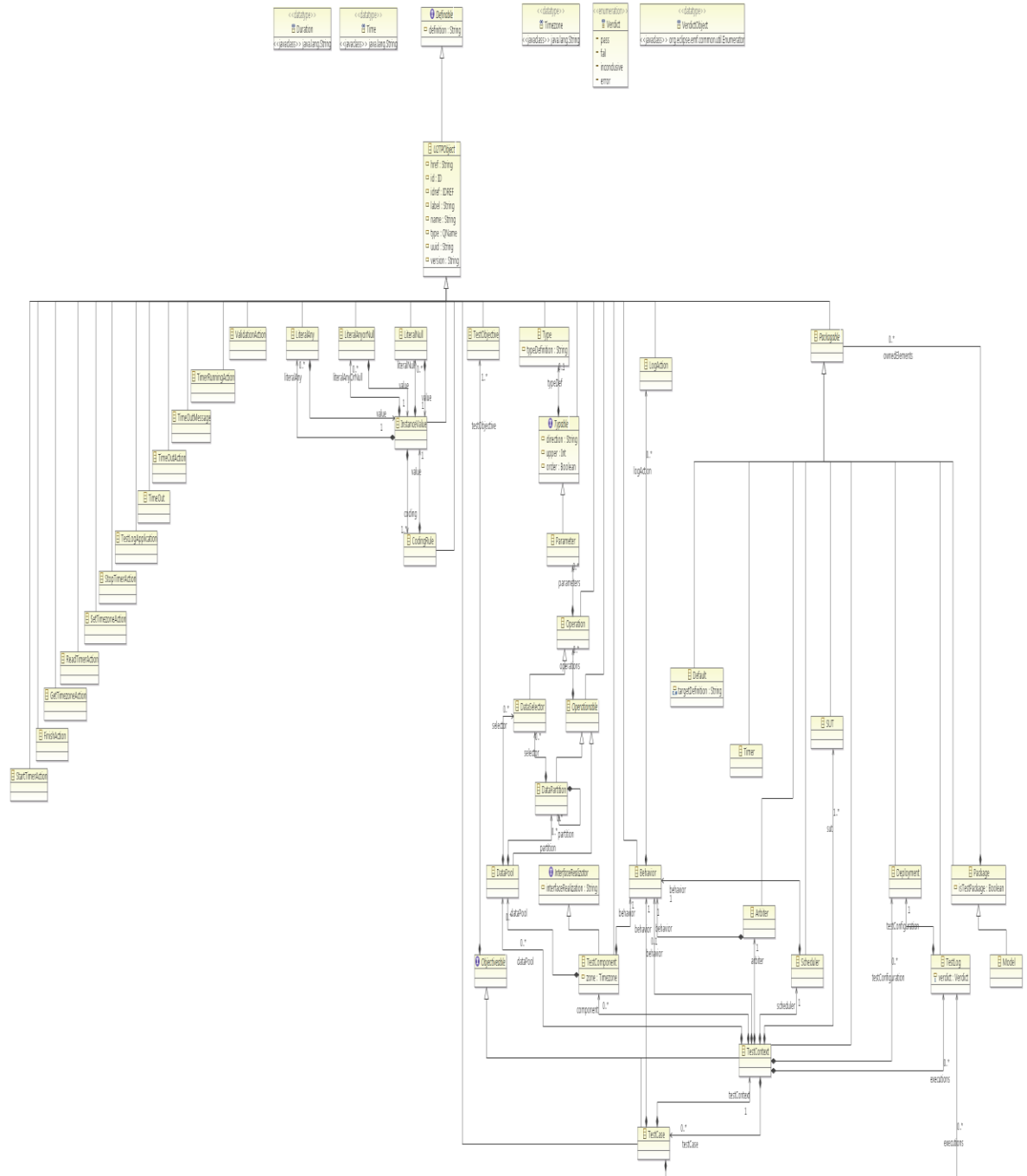


Imagen 33 - Metamodelo2 UML2TestUML

Reglas de Transformación

Esta transformación tiene un metamodelo origen simpleumlO2JO y un metamodelo destino u2TPSA. Las reglas para transformar de un modelo uml instancia de simpleumlO2JO a un modelo uml2 instancia de u2TPSA son las siguientes:

- Para un elemento Model, otro elemento Model es creado:
 - Con el mismo nombre.
 - Linkeado a los mismos elementos.
- Para un elemento Packageable otro elemento Packageable es creado:
 - Con el mismo nombre.
 - Linkeado a los mismos elementos.
 - Con isTestPackage si alguno de sus stereotypes empieza con el texto “U2TP”
- Para un elemento Class, si no tiene un stereotype con valor “U2TP::TestContext”, el elemento Class se pasa tal cual al modelo destino. Si contiene un stereotype con valor “U2TP::TestContext”, se crea un elemento TestContext:
 - Se crea un elemento Arbiter
 - Se crea un elemento TestComponent
 - Se crea un elemento DataPool
 - Se crea un elemento Scheduler
 - Se crea un elemento SUT
 - Se crea un elemento TestCase
 - Se crea un elemento TestObjective

Código Fuente

```

modeltype UML uses 'simpleuml02JO';
modeltype PSM uses 'u2TPSA';

transformation UML2TestUML(in uml : UML, out psm: PSM);

main() {
    uml.rootObjects()[UML::Model]->map toModel();
}

mapping UML::Model::toModel() : PSM::Model {
    name := self.name;
    ownedElements := self.ownedElements->collect(
        map ElementToNew(self))->asOrderedSet();
}

mapping Packageable::ElementToNew(model:UML::Model) : PSM::Packagable
when {false} {
    init {
        result := object PSM::Model{
            name := 'nonMapped--' + self.metaClassName();
        }
    }
}

mapping UML::Model::ElementToNew(model:UML::Model) : PSM::Packagable
when {self.hasTestElements()}
{
    init {
        result := object PSM::Model{

```

```

        name:= self.name;
        ownedElements := self.ownedElements->collect(map
            ElementToNew(model))->asOrderedSet();
    };
}

mapping UML::Package::ElementToNew(model:UML::Model) : PSM::Packagable
when {self.hasTestElements()}
{
    init {
        result := object PSM::Package{
            name:= self.name;
            definition := self.name;
            ownedElements := self.ownedElements->collect(map
                ElementToNew(model))->asOrderedSet();
            isTestPackage := self.hasU2TPApplied();
        };
    }
}

query UML::Package::hasTestElements() : Boolean {
    if(self.hasU2TPApplied()) then {
        return true;
    }
    else {
        return self.ownedElements[UML::Package]
            ->exists(hasTestElements());
    }
    endif;
}

query UML::Package::hasU2TPApplied() : Boolean {
    return self.stereotype->exists(aux : String | aux.startsWith('U2TP'));
}

mapping UML::Class::ElementToNew(model:UML::Model) : PSM::Packagable
when {self.hasU2TPTestContextApplied()}
{
    init {
        result := self.map class2TestContext(model);
    }
}

mapping UML::Class::class2Arbiter() : PSM::Arbiter {
    definition := self.name;
    name:= self.name;
}

mapping UML::Class::class2TestContext(model:UML::Model) : PSM::TestContext {
    arbiter := model.ownedElements[UML::Association]->select(c
        :UML::Association | c.source = self and
        c.target.hasAppliedStereotype('Arbiter'))
        ->first().target.map class2Arbiter();
}

```

```

component := self.getPropertiesWithStereotypedType(
  'U2TP::TestComponent')->oclAsType(UML::Class)->map
  classifier2TestComponent()->asOrderedSet();
dataPool := self.getPropertiesWithStereotypedType('U2TP::DataPool')->map
  classifier2DataPool()->asSequence()->
  union(self.getStereotypedPropertiesType('U2TP::DataPool')->map
    classifier2DataPool()->asSequence()->asOrderedSet());
name:= self.name;
scheduler := model.ownedElements[UML::Association]->select(c
  :UML::Association| c.source = self and
  c.target.hasAppliedStereotype('Scheduler'))
  ->first().target.map class2Scheduler();
sut := self.getStereotypedPropertiesType('U2TP::SUT')->map
  classifier2SUT()->asOrderedSet();

testCase := self.getStereotypedOperations('U2TP::TestCase')->map
  operation2TestCase(model)->asSequence()->
  union(self.getStereotypedOperations('U2TP::TestCase')->map
    operation2TestCase(model)->asSequence()->asOrderedSet());
definition := self.name;
testObjective := self.getStereotypedDependencySuppliers(
  'U2TP::TestObjective',model)->map namedElement2TestObjective()
  ->asOrderedSet();
}

mapping UML::Method::behavior2Behavior() : PSM::Behavior {
  name:= self.name;
  definition := self.specification;
}

mapping UML::Method::operation2TestCase(model:UML::Model) : PSM::TestCase {
  name:= self.name;
  behavior := self.map behavior2Behavior();
  definition := self.specification;
}

mapping UML::Class::namedElement2TestObjective() : PSM::TestObjective {
  name := self.name;
  definition := self.name;
}

query UML::Class::getStereotypedDependencySuppliers(stereotype: String,
model:UML::Model) : Sequence(UML::Class) {
  return model.ownedElements[UML::Association]->select(c :UML::Association|
    c.source = self and c.hasAppliedStereotype(stereotype))
    ->collect(c :UML::Association| c.target)->flatten();
}

mapping UML::Class::classifier2SUT() : PSM::SUT {
  name := self.name;
  definition:= self.name;
}

mapping UML::Class::class2Scheduler() : PSM::Scheduler {

```

```

    name := self.name;
    definition := self.name;
}

mapping UML::Class::classifier2TestComponent() : PSM::TestComponent {
    name := self.name;
    dataPool := self.getStereotypedPropertiesType('U2TP::DataPool')->map
        classifier2DataPool()->asSequence()->
        union(self.getPropertiesWithStereotypedType('U2TP::DataPool')->map
            classifier2DataPool()->asSequence()->asOrderedSet());
    zone := null;
    definition := self.name;
}

mapping UML::Class::classifier2DataPool() : PSM::DataPool {
    definition := self.name;
    partition := self.getPropertiesWithStereotypedType('U2TP::DataPartition')
        ->map classifier2DataPartition()->asOrderedSet();
    selector := self.getStereotypedOperations('U2TP::DataSelector')->
        map operation2DataSelector()->asOrderedSet();
    operations := self.methods[isDataPoolOperation()]->map
        operation2Operation()->asOrderedSet();
    name := self.name;
}

mapping UML::Class::classifier2DataPartition() : PSM::DataPartition {
    definition := self.name;
    name := self.name;
    operations := self.methods[isDataPoolOperation()]->map
        operation2Operation()->asOrderedSet();
    partition := self.getPropertiesWithStereotypedType('U2TP::DataPartition')
        ->map classifier2DataPartition()->asOrderedSet();
    selector := self.getStereotypedOperations('U2TP::DataSelector')->map
        operation2DataSelector()->asOrderedSet();
}

mapping UML::Method::operation2Operation() : PSM::Operation {
    name := self.name;
    parameters := self.parameters[UML::Parameter]->map
        parameterElement2Parameter()->asOrderedSet();
    definition := self.name;
}

mapping UML::Parameter::parameterElement2Parameter() : PSM::Parameter {
    definition := self.name;
    direction := self.kind;
    name := self.name;
    typeDef := self.type.map type2Type(self);
}

mapping UML::Type::type2Type(p : UML::Parameter) : PSM::Type {
    definition := 'PSM::'+ self.name;
    typeDefinition := self.name;
}

```

```

mapping UML::Method::operation2DataSelector() : PSM::DataSelector inherits
UML::Method::operation2Operation {

}

query UML::Method::isDataPoolOperation() : Boolean {
  return self.isAccesor() and
    (not (self.hasAppliedStereotype('U2TP::DataSelector')
      or
      self.GetReturn().hasAppliedStereotype('U2TP::DataPartition')));
}

query UML::Method::isAccesor() : Boolean {
  return self.GetReturn() != null and
    self.name.toLower().startsWith('get') and
    self.visibility = 'public';
}

query UML::Class::getStereotypedOperations(stereotype: String) :
Set(UML::Method) {
  return self.methods->select(hasAppliedStereotype(stereotype));
}

query UML::Class::getStereotypedPropertiesType(stereotype: String) :
Bag(UML::Class) {
  return self.attributes->select(hasAppliedStereotype(stereotype))->map
property2Class()->asBag();
}

query UML::ModelElement::hasAppliedStereotype(in s: String) : Boolean {
  return self.stereotype->exists(aux : String | aux = s);
}

query UML::Class::getPropertiesWithStereotypedType(stereotype: String) :
Bag(UML::Class) {
  return self.attributes->select(c :UML::Property|
c.hasU2TPTestComponentApplied()->map property2Class()->asBag();
}

query UML::Class::hasU2TPTestContextApplied() : Boolean {
  return self.stereotype->exists(aux : String | aux = 'U2TP::TestContext');
}

query UML::Property::hasU2TPTestComponentApplied() : Boolean {
  return self.stereotype->exists(aux : String | aux =
'U2TP::TestComponent');
}

mapping UML::Property::property2Class() : UML::Class {
  init {
    result := self.type.ocLAsType(Class);
  }
}

```

```

    }
}

```

8 Repositorio de Transformaciones

En este capítulo se presenta la solución que tiene como objetivo la construcción del repositorio de transformaciones QVT. En primera instancia se menciona la descripción general, luego se explica la arquitectura elegida. Seguido de ello se habla del diseño realizado y por último se detalla la implementación, mostrando el IDE de desarrollo y la solución empleada.

8.1 Descripción General

El repositorio busca como objetivo fomentar la difusión y expansión de la metodología MDD, particularmente transformaciones QVT, a la personas afines, como también brindar una puerta de entrada a este mundo a quienes aún no lo conocen.

Para facilitar la difusión de las transformaciones QVT el repositorio denominado “Repositorio QVT” presenta el formato de un sitio web, el cual hace sencilla la interactividad con los usuarios que deseen visitarlo.

El sitio presentará entre sus funcionalidades más destacadas una entrada en donde los usuarios encontrarán las transformaciones publicadas, pudiendo ver una sinopsis de cada una de ellas, un documento con las especificaciones y el código fuente de cada una, pudiendo bajar dichos documentos para su utilización, difusión y extensión de manera que el usuario crea conveniente. También se encuentran destacadas las últimas transformaciones publicadas, como las últimas subidas al repositorio y que están disponibles para ser supervisadas por el administrador responsable del contenido existente, antes de su publicación definitiva.

Otra funcionalidad existente, y quizás una de las más importante incorporada al repositorio, es la posibilidad de que los usuarios interesados que poseen transformaciones en QVT suban su material para poder publicarlo y difundirlo. Esto hace que los usuarios no estén limitados sólo al contenido entregado por el sitio en sí, sino que ellos sean la vía constante de mejora y enriquecimiento de la información que se difunde en el mismo.

8.2 Arquitectura

Se plantea una arquitectura en capas con el fin de hacer una aplicación lo suficientemente desacoplada entre ellas como para que cada una cumpla una funcionalidad específica, pero a la vez estén intensamente relacionadas una con otra.

Tendremos tres capas principales, una de presentación, una de lógica de negocio y otra de persistencia. Cada una de ellas contendrá los componentes necesarios para llevar adelante su función.

Para relacionar las capas mencionadas anteriormente se utilizará un framework Seam el cual permite relacionar las capas de una forma sencilla y transparente.

La elección del framework Seam se realiza con el objetivo de contar con un soporte de última generación open source, permitiendo bajar los niveles de configuración de los frameworks tradicionales como Struts, agilizando la construcción en base a componentes y facilitando una relación transparente entre ellos sin tener que realizar grandes y complejas configuraciones por separado de varios frameworks. Seam se basa en estos aspectos, ya que como se muestra en la imagen 34, por su naturaleza permite realizar lo dicho anteriormente.

Entre otros beneficios Seam ofrece una completa integración con el framework de presentación JSF, un framework para manejar las inyecciones de instancias al estilo spring y el uso de JPA de forma transparente para manejar la persistencia sin tener que usar necesariamente un framework más complejo del estilo de Hibernate. De esta manera incorporando Seam unimos muchos de los frameworks más importantes del mercado para la plataforma java de forma simple, que de otra manera se tendría que hacer de forma separada y configurar cada uno de ellos haciendo más engorroso y complejo la configuración y desarrollo del proyecto.

Otra de las posibilidades que brinda Seam, y que nos lleva a su elección para la implementación del repositorio, es que permite una gran escalabilidad en su arquitectura. Pudiendo emplear tanto un simple contenedor web (web container) como Tomcat o un completo servidor de aplicaciones (application server) como Jboss [32] u otro, permitiendo contar con todos los beneficios J2EE si fuese necesario. También nos permite tener autenticación de forma sencilla.

8.2.1 Visión de general de la arquitectura del repositorio de transformaciones

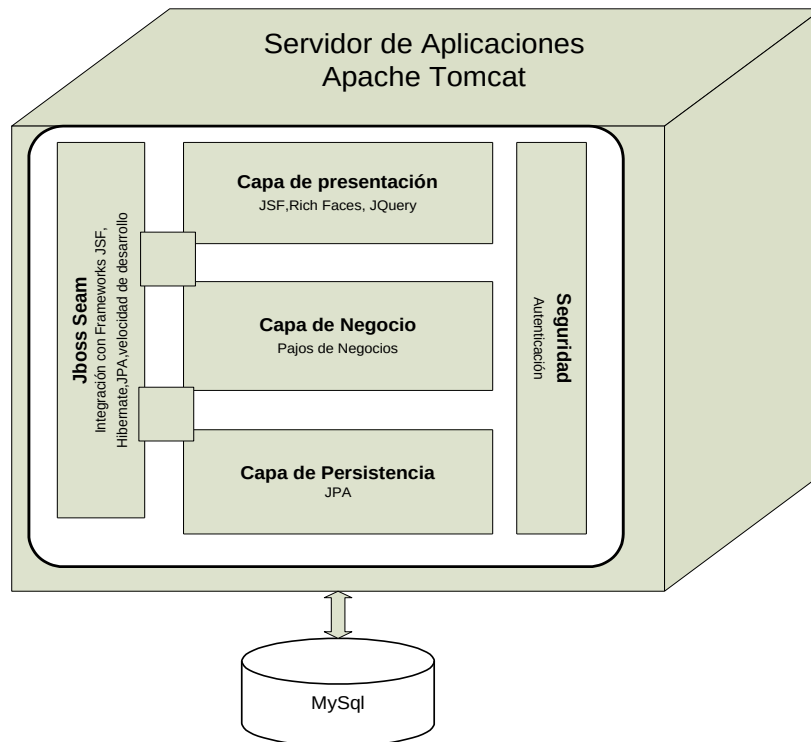


Imagen 34 - Visión de general de la arquitectura del repositorio

- Arquitectura de capas JEE.
- Capa de presentación que integra JSF, Richfaces.
- Capa de negocio en donde se resuelve la lógica más pesada de la aplicación.
- JPA: como framework ORM para manejo de la persistencia. Puente directo y automático entre los objetos de negocio y la base de datos
- Jboss Seam, brinda:
 - Integración de los diversos frameworks.
 - Autenticación integrada
- Internacionalización (i18n) de todos los menús, mensajes del sistema y de error, etcétera, para lograr que *sólo cambiando la configuración se cambie el idioma de la aplicación.*
- Apache Tomcat.

8.3 Diseño

8.3.1 Visión General de packages

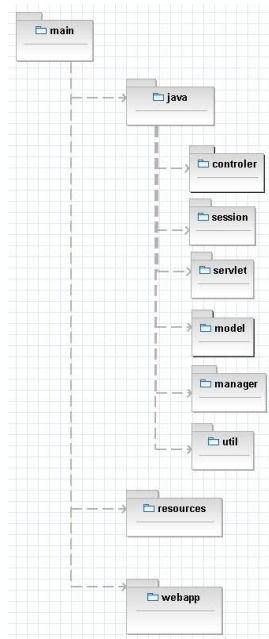


Imagen 35 - Visión General de packages

8.3.2 Dependencia entre packages

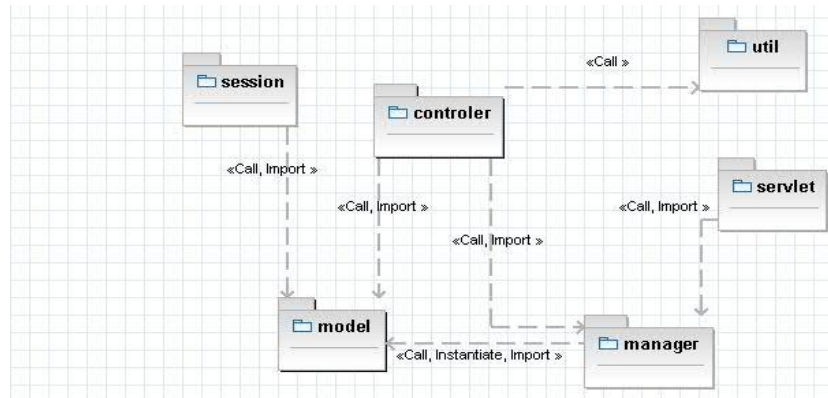


Imagen 36 - Dependencia entre packages

8.3.3 Visión detallada de packages

- java
 - controller: Este package contiene las clases encargadas de atender los pedidos de la vista e interactuar con la lógica de negocio.

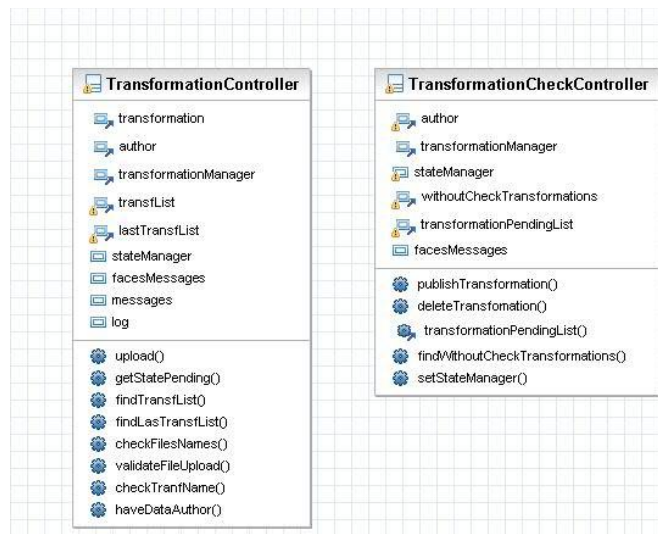


Imagen 37 - Visión detallada de packages - Controler

- model: En este package se encuentran las clases que representan el modelo de la aplicación.

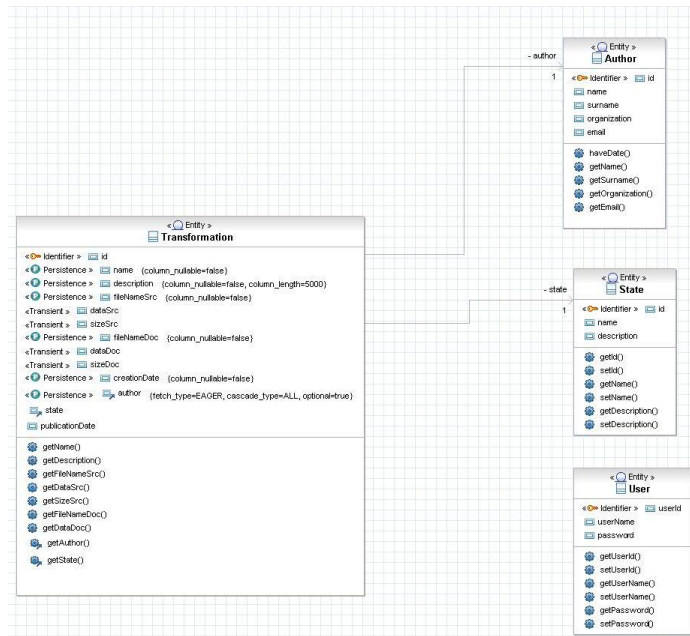


Imagen 38 - Visión detallada de packages - Model

- manager: Este package contiene las clases que realizan parte de la lógica y son las encargadas de llamar el framework de JPA para la persistencia.

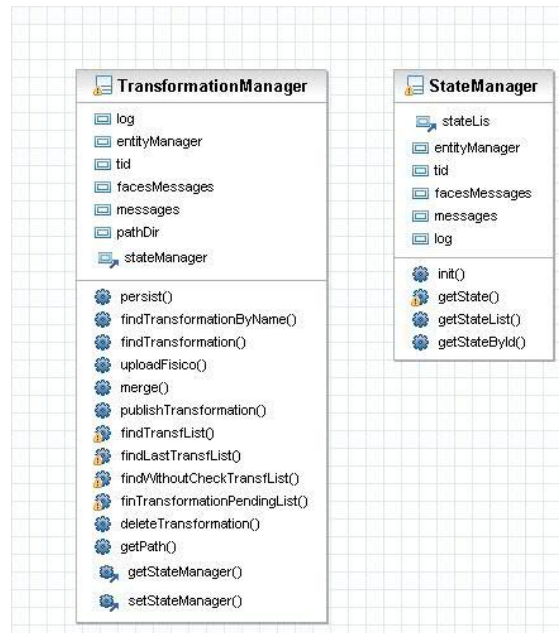


Imagen 39 - Visión detallada de packages –Manager

- session: Este package contiene las clases encargadas de realizar la autenticación del usuario.

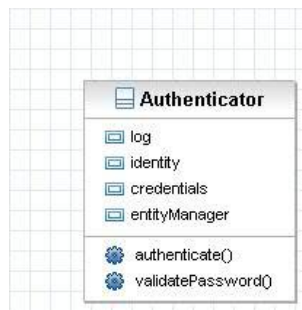


Imagen 40 - Visión detallada de packages - Session

- servlet: Este package contiene las clases encargas de realizar el download de los archivos del repositorio.

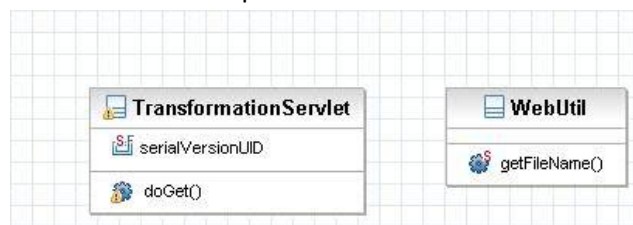


Imagen 41- Visión detallada de packages - Servlet

- util: Este paquete contiene clases auxiliares utilizadas por los tags de la vista.

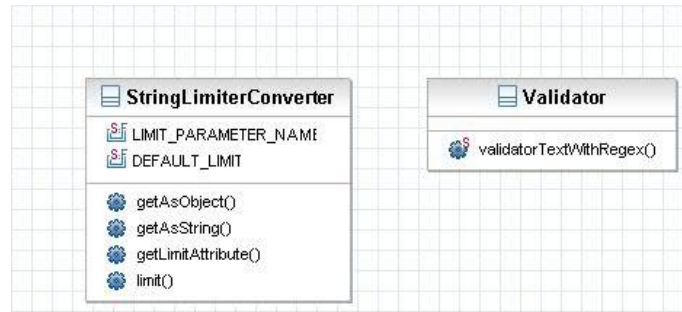


Imagen 42 - Visión detallada de packages - Util

- resources: En este package se encuentran los archivos de configuración de persistencia, los scripts de carga inicial de datos si fuese necesario, la configuración de log4j, archivos de propiedades necesarios para el funcionamiento de la aplicación y los archivos de internacionalización.
 - persistence.xml
 - componentes.properties (generado por seam)
 - propiedades.properties (usado por la aplicación para obtener expresiones regulares necesarias)
 - messages_es.properties
- webapp: Este package contiene las imágenes usadas en la vista, las clases java script, la definición del layout de la aplicación, los archivos de estilos, las paginas xhtml y archivos de configuración necesarios.

Modelo de Datos

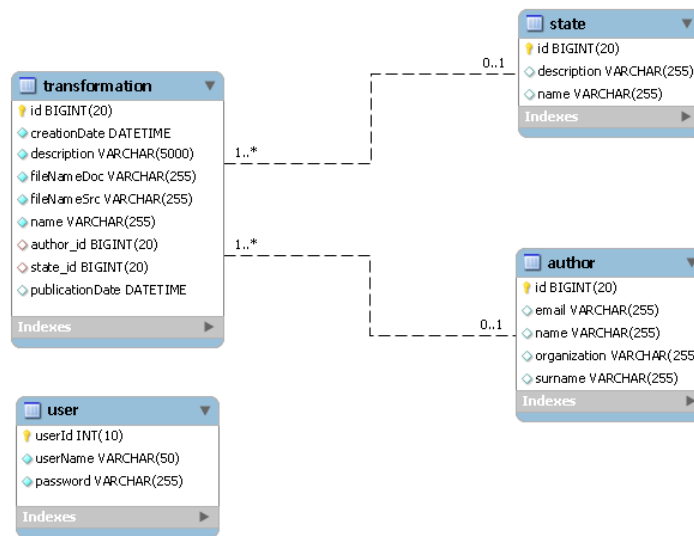


Imagen 43 - Modelo De Datos

8.4 Implementación

Para realizar la implementación, además de tomar como base lo ya explicado en las secciones anteriores 8.2 y 8.3, se utiliza como IDE de desarrollo eclipse ya que es a nuestro criterio una de las mejores herramientas open source del mercado. Para trabajar de forma más amena e interactiva con Seam, instalamos las herramientas que nos ofrece Jboss para tener una total integración entre eclipse y Seam.

Como motor de base de datos utilizamos MySQL 5.1 [51] y como cliente para administrar la base de datos usamos HeidiSQL [52].

8.4.1 Visión del IDE de desarrollo y Jboss Tools [34]

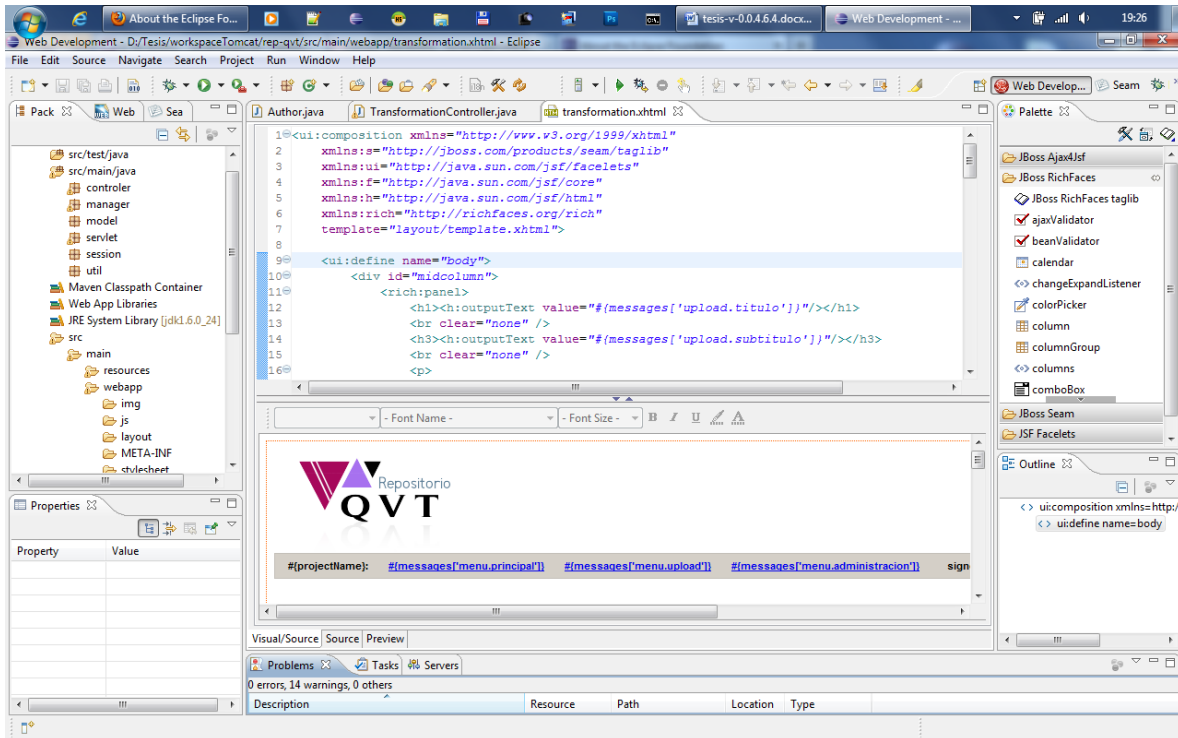


Imagen 44 - Visión del ide + Jboss

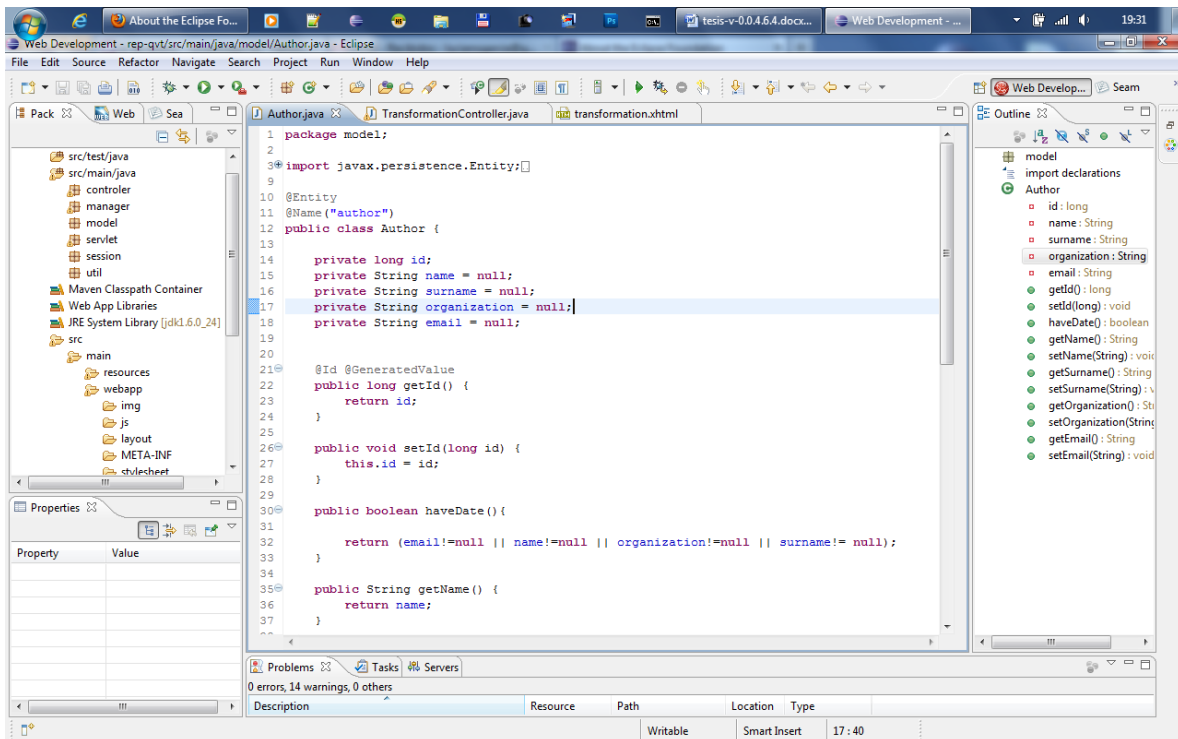


Imagen 45 - Visión del Ide vista código Java

8.4.2 Visión general de la funcionalidad del repositorio

Transformaciones Publicadas: el punto de entrada al sitio es la pantalla *Transformaciones Publicadas*, en el cual se pueden apreciar las transformaciones en la parte central, pudiendo buscar una transformación introduciendo un texto libre en el área de búsqueda. También se encuentran de forma destacada las últimas transformaciones publicadas que todavía están pendientes de evaluación.



Imagen 46 - Visión general del repositorio - Transformaciones Publicadas

Subir Transformaciones: en esta pantalla se encuentra la funcionalidad que le permite al usuario subir al repositorio una transformación, debiendo incluir un documento descriptivo y explicativo de la transformación como así también el código fuente de la misma.

The screenshot shows the 'Upload De Transformaciones QVT' page. The sidebar on the left contains the following links:

- Ultimas Transformaciones Publicadas
 - Umi2TestUml
- Transformaciones Pendientes de Publicacion
 - ReplaceManyToMany2Association

The main content area is titled 'Upload De Transformaciones QVT' and contains the following sections:

Datos de la nueva transformacion QVT

Nombre Transformación:

Descripcion:

Upload Src:

Upload Doc:

Datos de autoria

(Los datos de autoria son opcionales)

Nombre:

Apellido:

Organización:

Email:

Imagen 47 - Visión general del repositorio - Subir Transformaciones

Login: en esta pantalla el usuario podrá autenticarse y ser direccionado al área de administración de las transformaciones.

The screenshot shows the Login page. The sidebar on the left contains the following links:

- Ultimas Transformaciones Publicadas
 - Umi2TestUml
- Transformaciones Pendientes de Publicacion
 - ReplaceManyToMany2Association

The main content area is titled 'Por Favor Autentiquese' and contains the following form:

Usuario:

Contraseña:

Remember me: ☐

Note - You may login with the username 'admin' and 'admin' password.

Imagen 48 - Visión general del repositorio - Login

Administración de Transformaciones: en esta pantalla el usuario autenticado como administrador podrá supervisar tanto la documentación respaldatoria como el código fuente de las transformaciones que se deseen publicar, pudiendo publicarla si es correcto o borrarla si así lo ameritara.



Imagen 49 - Visión general del repositorio - Administración de Transformaciones

8.4.3 Visión detallada de la funcionalidad del repositorio

Transformaciones Publicadas: En esta pantalla podemos observar las secciones marcadas las cuales nos indican diferente información sobre las transformaciones o nos permiten realizar diversas acciones.

Como primer punto nos encontramos las acciones remarcadas con color verde claro, el cual representa el menú principal del repositorio, estando el mismo disponible en todo momento.

Seguidamente nos encontramos con una barra lateral en la cual se informan las últimas transformaciones publicadas y las que aún están pendientes esperando ser publicadas. Estas serán evaluadas por un administrador o supervisor, lo cual se describirá más adelante.

Por último observamos la zona central, donde encontramos dos secciones destacadas. La primera de ellas nos permite buscar una transformación según el texto que ingresemos. La segunda nos muestra toda la información relacionada a la transformación y nos permite bajar el documento que describe detalladamente la transformación (Documentación) y el archivo que contiene el código fuente (Código Fuente).



Imagen 50 - Visión detallada del repositorio - Transformaciones Publicadas

Subir Transformaciones: En esta pantalla se cargan datos pertenecientes a una transformación como nombre, descripción y datos de autoría.

En la imagen 52 observamos cómo hacer el upload de los archivos de documentación y código fuente correspondiente a la transformación.

Luego nos encontramos con la acción de “Registrar Transformación” remarcada en rojo, la cual nos permite subir al repositorio la transformación que deseamos compartir con el resto de los usuarios.

La imagen 53 nos muestra como se visualiza la transformación, luego de ser cargada (upload), en la sección de transformación pendientes de publicación.



Imagen 52 -Visión detallada del repositorio - Subir Transformaciones 1

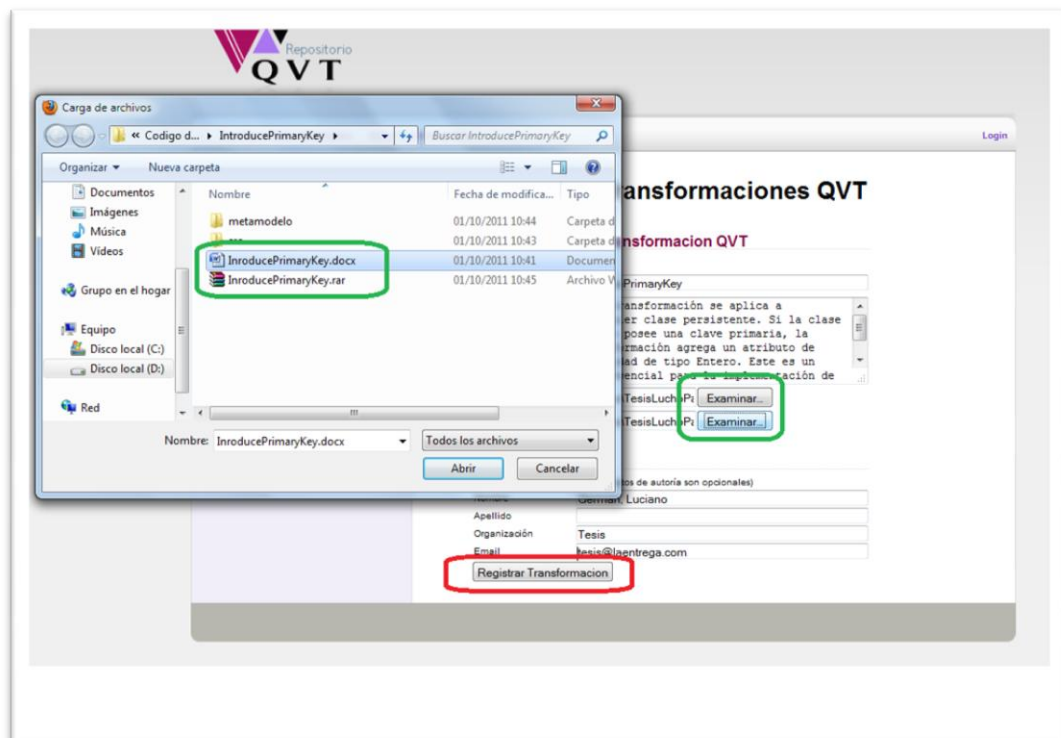


Imagen 51 - Visión detallada del repositorio - Subir Transformaciones 2

Administración de Transformaciones: En esta pantalla podremos publicar o borrar una transformación.

El administrador podrá bajar el documento donde se describe la transformación como también el archivo con el código fuente, los cuales deberán ser supervisados.

Para finalizar vemos en la Imagen 55 como se refleja la transformación recientemente publicada, quedando disponible para que los usuarios puedan utilizar.



Imagen 53 - Administración de Transformaciones - Principal



Imagen 54 - Administración de Transformaciones - DownLoad



Imagen 55 - Administración de Transformaciones - Barra Lateral

9 Caso de Estudio

En este capítulo se muestra la aplicación de algunas de las transformaciones desarrolladas en capítulos anteriores a través de un ejemplo concreto. La idea es especificar un modelo UML que permita aplicarle varias transformaciones, las cuales tienen como origen y salida al metamodelo SimpleUML. El objetivo es que el modelo UML resultante sea la representación en UML del Diagrama Entidad Relación (DER).

Se aplican consecutivamente algunas de las transformaciones presentadas en la tesis simulando en cierta forma lo que hace la transformación de UML a RDB analizada anteriormente.

9.1 Definición del modelo UML

El modelo UML a utilizar para el caso de estudio es una simplificación de un modelo UML que representa un sistema de una Facultad.

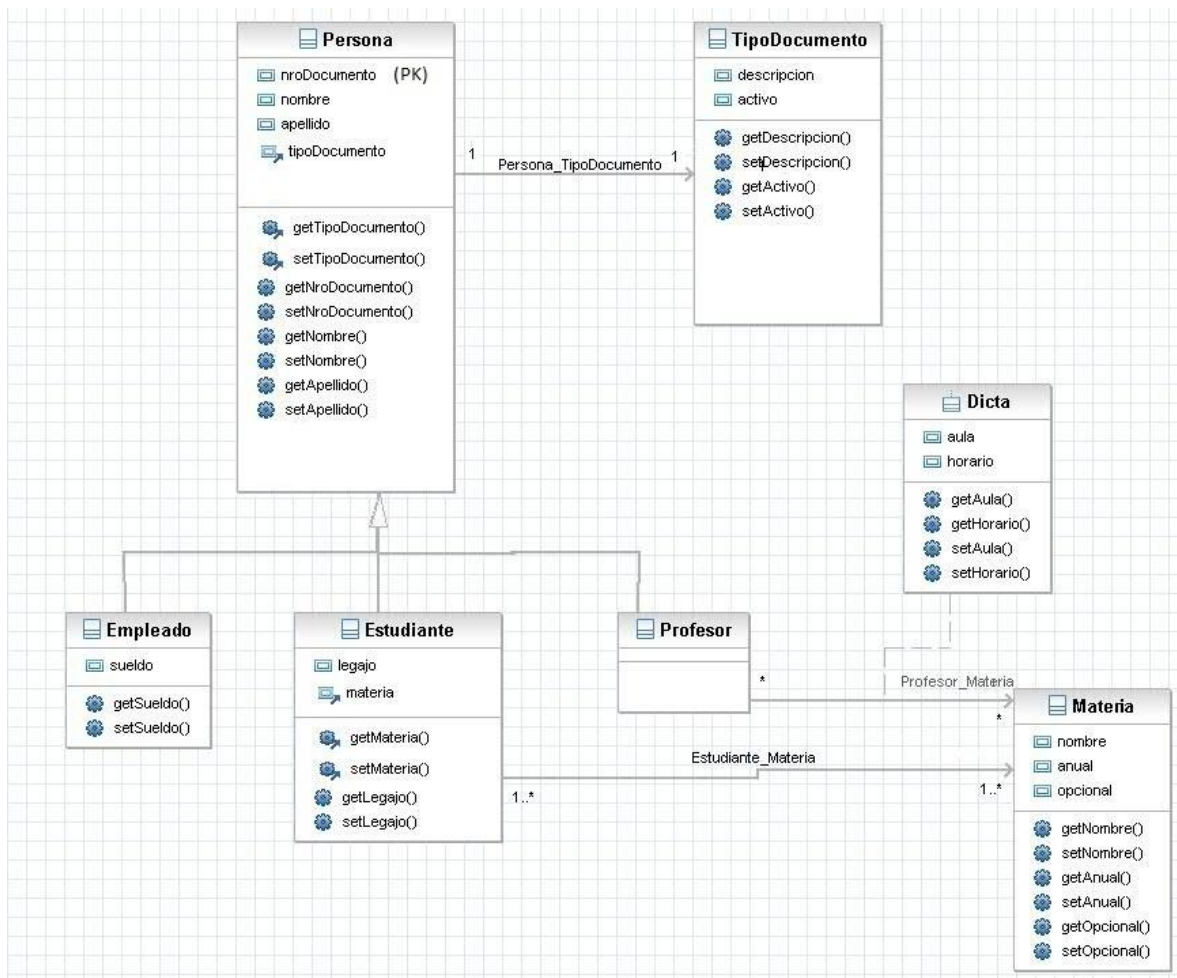


Imagen 56 - Modelo UML Caso de Estudio

9.2 Aplicación de Transformaciones

9.2.1 Transformación 1: RemoveAssociationClass

Al aplicar esta transformación sobre el modelo UML descrito en el punto anterior la clase asociación “Dicta” se transforma en una clase y se crean dos nuevas asociaciones, “Profesor_Dicta” y “Materia_Dicta”, ambas de 1 a N.

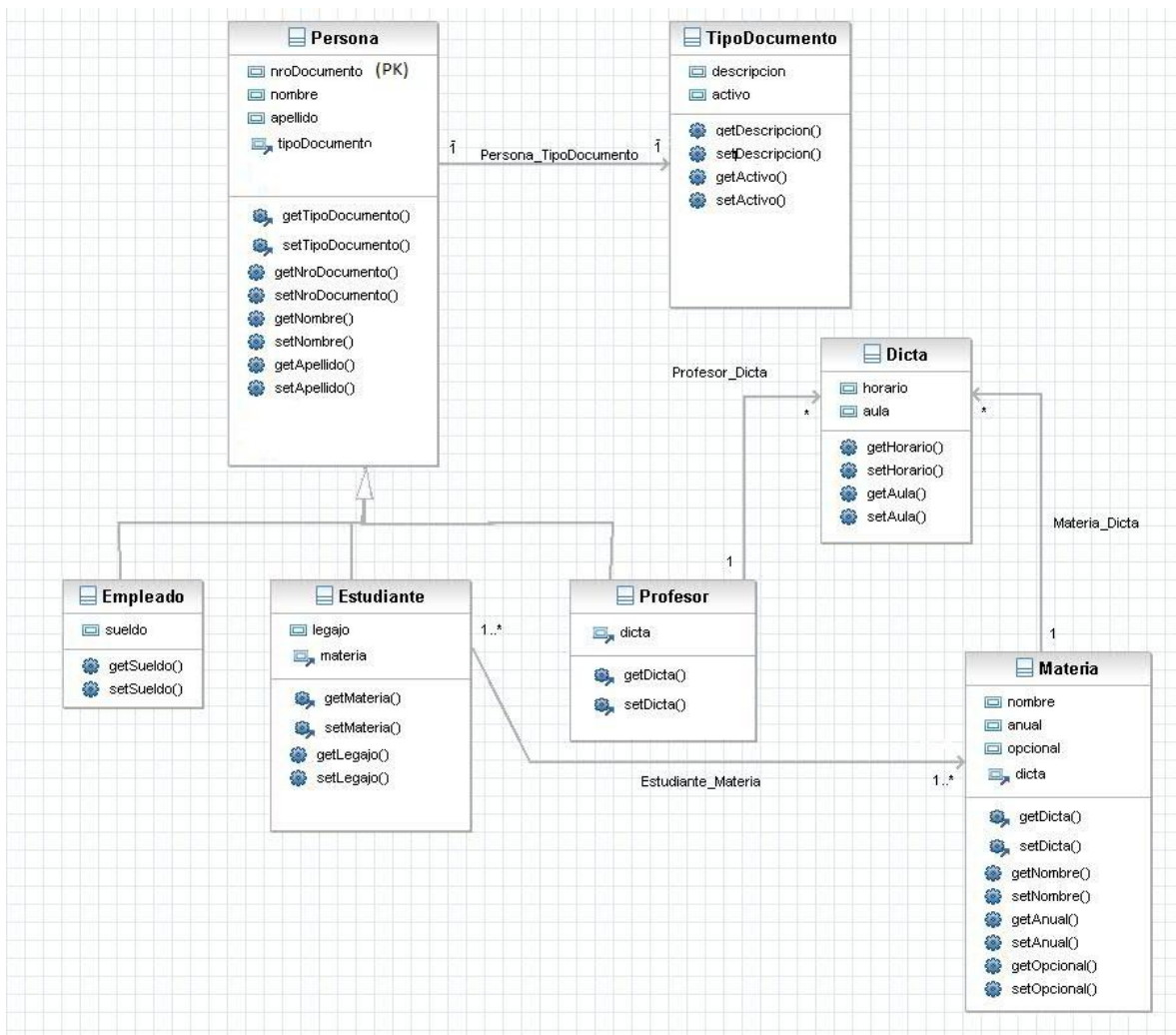


Imagen 57 - Modelo resultante de aplicar **RemoveAssociationClass**

9.2.2 Transformación 2: ReplaceManyToMany2Association

Al aplicar esta transformación sobre el modelo resultante de la transformación anterior, la relación N a N, “Estudiante_Materia”, se transforma en una nueva clase, “Estudiante_Materia”, y se crean dos nuevas asociaciones de 1 a N, “Estudiante_Estudiante_Materia” y “Materia_Estudiante_Materia”.

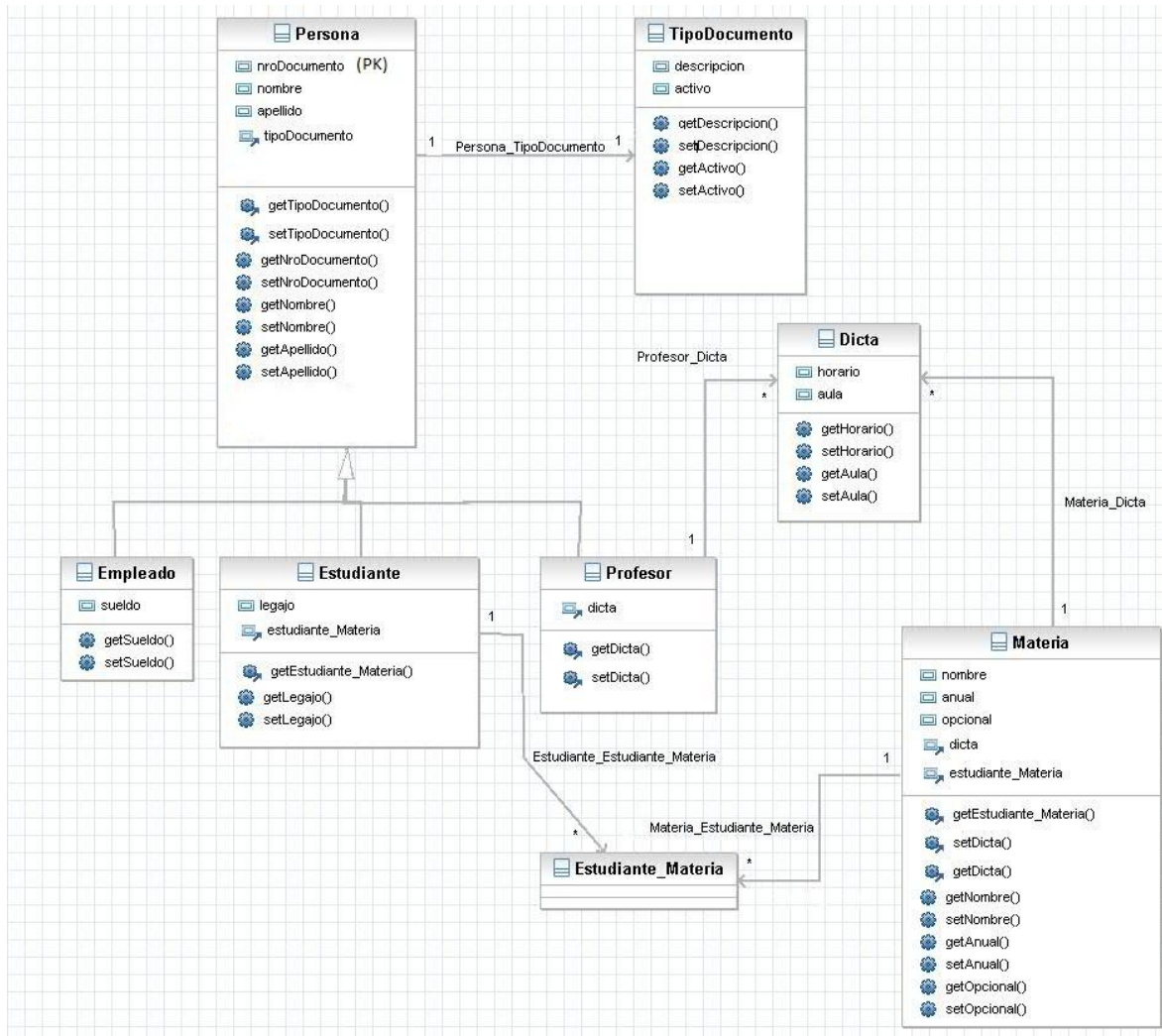


Imagen 58 - Modelo resultante de aplicar ReplaceManyToMany2Association

9.2.3 Transformación 3: ReplaceInheritanceByAssociation

Al aplicar esta transformación sobre el modelo resultante de la transformación anterior, las relaciones de herencia entre la clase “Persona” y las clases “Empleado”, “Estudiante” y “Profesor” se transforman en asociaciones de 1 a 1 entre dichas clases. Por ello, se crean tres nuevas asociaciones de 1 a 1, “Empleado_Persona”, “Estudiante_Persona” y “Profesor_Persona”.

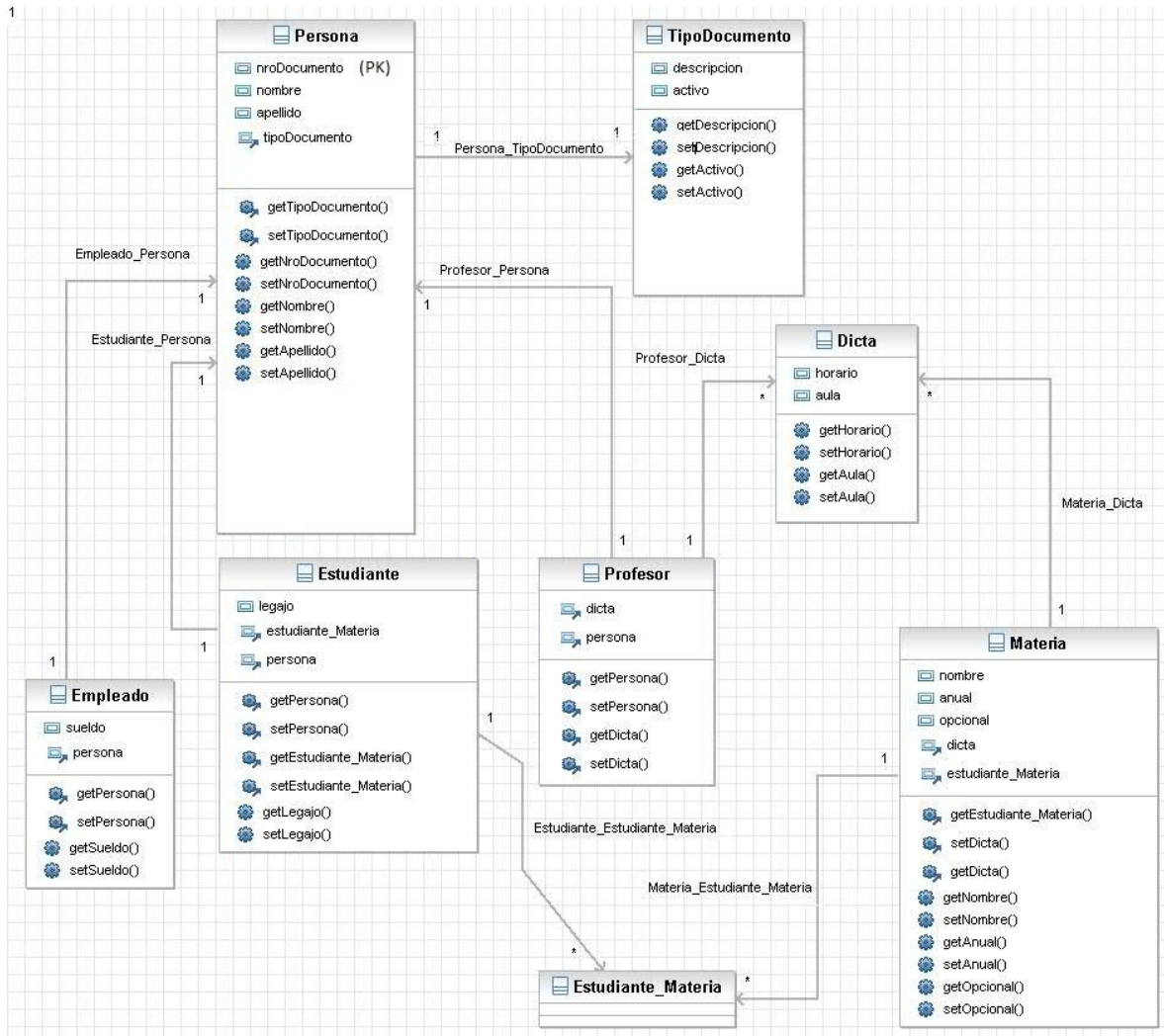


Imagen 59 - Modelo resultante de aplicar ReplaceInheritanceByAssociation

9.2.4 Transformación 4: IntroducePrimaryKey

Al aplicar esta transformación sobre el modelo resultante de la transformación anterior, se crea una nueva propiedad con stereotype = "PrimaryKey" en todas aquellas clases que aún no cuentan con una propiedad con ese stereotype. Como la clase Persona ya cuenta con una propiedad con stereotype = "PrimaryKey", no se crea una nueva propiedad con ese stereotype. En el resto de las clases del modelo UML se crea una nueva propiedad llamada ID, de tipo Integer y con stereotype = "PrimaryKey".

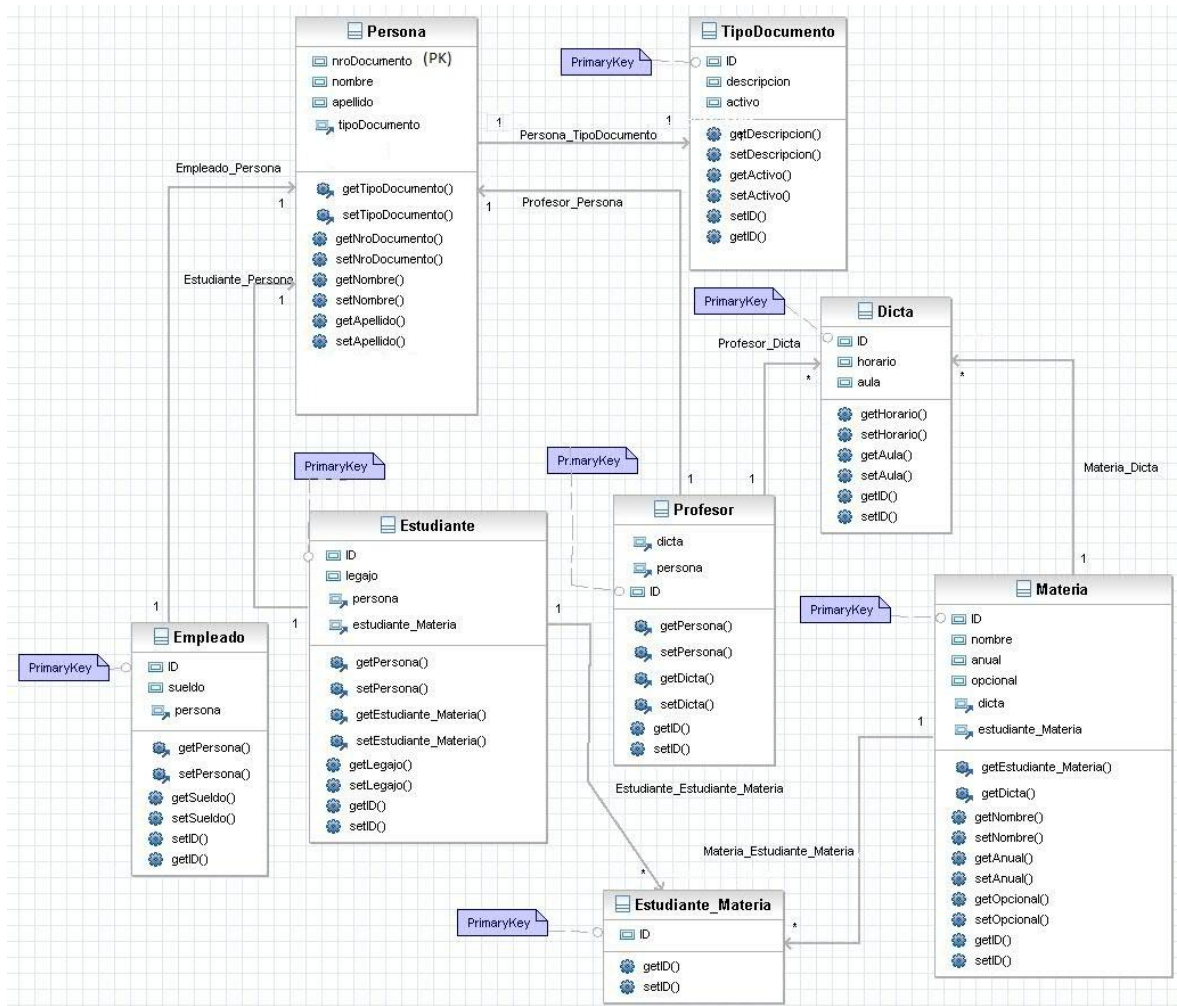


Imagen 60 - Modelo resultante de aplicar IntroducePrimaryKey

9.2.5 Transformación 5: ReplaceAssociationbyForeignKey

Al aplicar esta transformación sobre el modelo resultante de la transformación anterior, se eliminan todas las asociaciones del modelo, en su lugar se crean propiedades con Stereotype = “ForeignKey” del lado de las clases con cardinalidad N o del lado de las clases con cardinalidad 1 en las asociaciones de 1 a 1. La nomenclatura para el nombre de las nuevas propiedades es “NombrePrimaryKey_NombreClaseRelacionada”. Por ejemplo, la asociación 1 a 1 “Empleado_Persona” se reemplaza agregando la propiedad “nroDocumento_Persona” a la clase “Empleado”, la asociación 1 a N “Profesor Dicta” se reemplaza agregando la propiedad “ID_Profesor” a la clase “Dicta”.

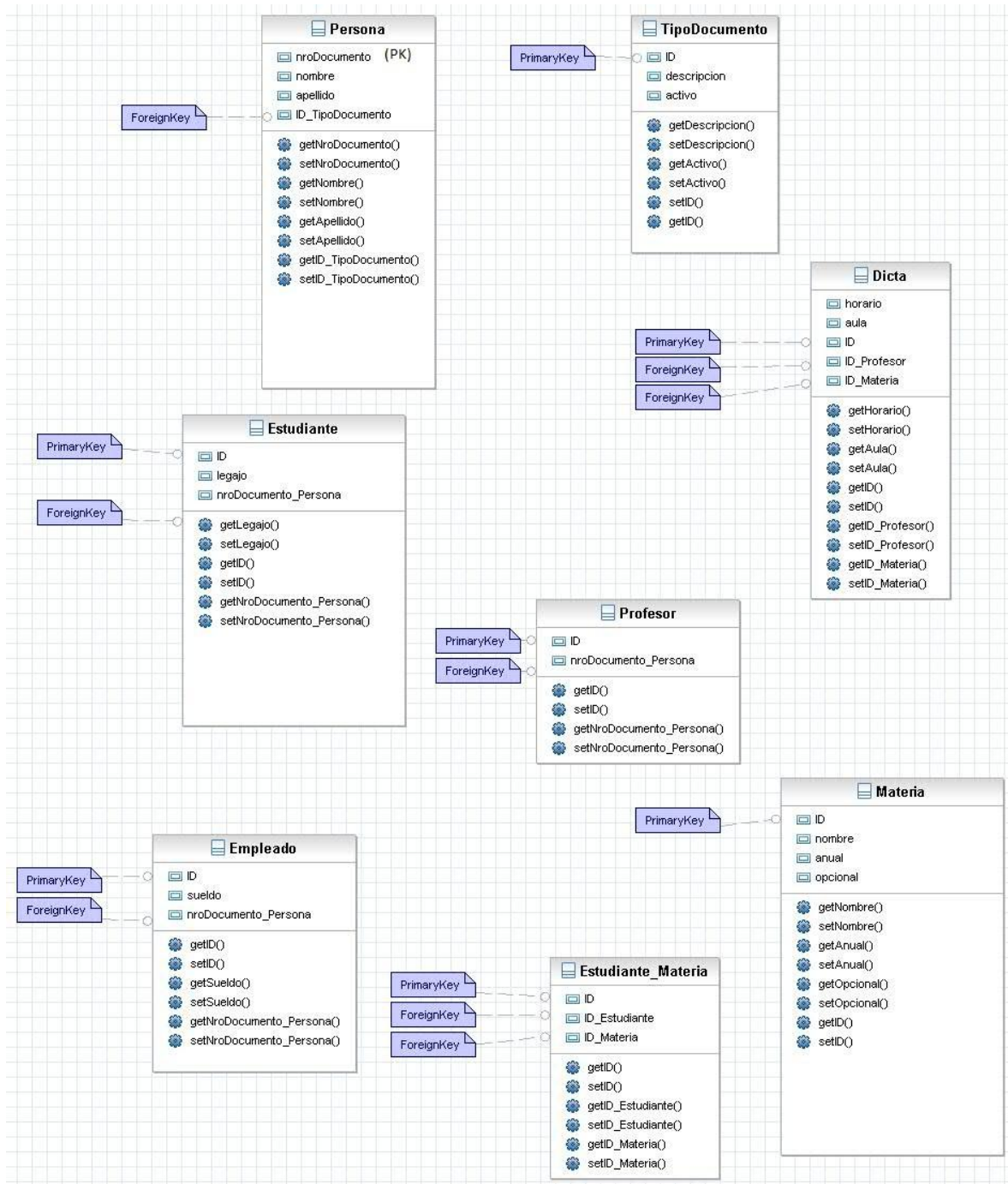


Imagen 61 - Resultado de aplicar `ReplaceAssociationbyForeignKey`

10 Conclusión y Trabajos Futuros

10.1 Conclusión

En el presente trabajo se implementó un repositorio de transformaciones las cuales representan modelos de software cuyo objetivo apunta a la difusión de su contenido facilitando a la comunidad MDD utilizar y enriquecer el conjunto de transformaciones que hoy en día se encuentran publicados en él.

MDD permite mejorar la construcción de software basándose en un proceso guiado por modelos soportado por potentes herramientas. Orientados al dominio, constituyen el foco principal en el desarrollo de software.

Las herramientas de transformaciones toman un modelo como entrada y producen uno como salida a partir de un conjunto de reglas.

De esta forma, MDD permite reducir el tiempo y costo en el diseño de los modelos, ya que posibilita adaptarlos rápidamente a los cambios tecnológicos y a modificaciones en los requisitos.

Como punto de partida se investigó el repositorio existente de transformaciones en lenguaje ATL. Se seleccionó un conjunto de ellas, para las cuales se definieron sus respectivos modelos e implementación en QVT.

Las transformaciones elegidas y desarrolladas que inicialmente forman parte del repositorio QVT son aquellas que toman como origen un modelo UML, dado que éste es el lenguaje de modelado más utilizado en el desarrollo con MDD como así también en el diseño tradicional de la ingeniería de software. Para simplificar la comprensión y legibilidad de las transformaciones de este trabajo se optó por utilizar un metamodelo simplificado de UML, llamado SimpleUML.

Como base para el desarrollo de transformaciones de modelos se utilizaron las herramientas de la plataforma eclipse, específicamente las ofrecidas en la iniciativa de código abierto Eclipse Modeling Project. Dicha plataforma brinda soporte al lenguaje de transformación entre modelos. Además contiene un conjunto de frameworks que implementan muchos de los estándares de la OMG. Con el mismo espíritu y tomando como soporte la Eclipse Foundation se decidió utilizar herramientas basadas en la plataforma Java para el desarrollo del repositorio.

Por otra parte, para la implementación del repositorio de transformaciones QVT se utilizaron varios frameworks innovadores basados en la plataforma java (Seam, JPA, Maven y JQuery). De esta forma garantizamos que el repositorio no requiera de grandes cambios tecnológicos en el corto plazo y que sea una motivación para los interesados mejorarlo y/o adaptarlo con el paso del tiempo.

Para concluir, destacamos el aporte novedoso que genera el presente trabajo para la comunidad MDD como para los usuarios en general, ya que en la actualidad no existe un repositorio de transformaciones QVT de estas características.

10.2 Trabajos Futuros

En base a los conceptos planteados a lo largo de esta tesis y a los resultados obtenidos, se proponen las siguientes líneas como trabajos futuros:

- Implementar en lenguaje QVT otras transformaciones de interés en el diseño de software, tales como el resto de las transformaciones existentes en el repositorio ATL, otros patrones de diseño y refactorings más complejos, etc..
- Extender las soluciones presentadas en esta tesis para el modelo UML completo.
- Plugins que permitan construir ecores sobre plataforma eclipse que use el metamodelo Simple UML como base para la construcción de diagramas simplificados.
- Implementación de un foro dentro del repositorio qvt donde discutir temas afines al desarrollo de transformaciones QVT.

11 Referencias

- [1] Booch,G., Rumbaugh,J. y Jacobson,I., The UML User Guide Addison Wesley Longman, Inc. (1998).
- [2] Standard Profiles UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification (chapter 18). <http://www.omg.org>.
- [3] UML 1.4, Object Management Group, The Unified Modeling Language (UML) Specification – Version 1.4, en <http://www.omg.or> (1999).
- [4] UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification. August 2003. <http://www.omg.org>.
- [5] Kent Beck. Extreme Programming Explained: Embrace Change. Boston: Addison Wesley, 2000
- [6] Cockburn, Alistair. Agile Software Development. Boston: Addison- Wesley, 2002.
- [7] Favre Jean-Marie, Estublier Jacky, Blay Mireille. Beyond MDA : ModelDrivenEngineering (L'Ingénierie Dirigée par les Modèles : au-delà du MDA).Edition Hezmes-Lavoisier, ISBN 2-7462-1213-7. (2006).
- [8] Kleppe, Anneke G. and Warmer Jos, and Bast, Wim. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [9] Mellor, Stephen J. and Scott, Kendall and Uhl, Axel and Weise Dirk. MDA Distilled,Principles of Model_Driven Architecture. Addison-Wesley, 2004.
- [10] Object Management Group, MDA Guide, v1.0.1, omg/03-06-01 (2003).

- [11] Object Management Group (OMG) <http://www.omg.org>
- [12] Meta Object Facility (MOF) 2.0. OMG Adopted Specification. October, 2003. <http://www.omg.org>.
- [13] The Unified Modeling Language Superstructure. version 2.0.,OMG Final Adopted Specification. April 2004. <http://www.omg.org>.
- [14] Java Metadata Interface (JMI), <http://java.sun.com/products/jmi/>
- [15] XML Metadata Interchange (XMI), v2.1, <http://www.omg.org/cgi-bin/doc?formal/2005-09-01> , (full specification)
- [16] OptimalJ (Compuware), 2007, www.compuware.com/products/optimalj
- [17] ArcStyler 5.5 (Interactive Objects), 2006, www.interactiveobjects.com/products/arcstyler
- [18] AndromDA, v3.2, Nov.2006, www.andromda.org/
- [19] Eclipse Modeling Framework EMF. <http://www.eclipse.org/modeling/emf/>
- [20] Meta Object Facility (MOF) 2.0. Query/View/Transformation Specification (QVT) - Version 1.0. April 2008. <http://www.omg.org>.
- [21] OMG. The Object Constraint Language Specification – Version 2.0, for UML2.0, revised by the OMG, <http://www.omg.org>, April 2004.
- [22] Jouault F., Kurtev I. Transforming Models with ATL. Workshop in Model Transformation in Practice at the MODELS 2005 Conference. Montego Bay, Jamaica, Oct 3, 2005
- [23] Lenguaje kermeta (<http://www.kermeta.org/>)
- [24] Medini Home page. <http://projects.ikv.de/qvt>
- [25] OSLO Home page. [Http://oslo-project.berlios.de/](http://oslo-project.berlios.de/)
- [26] Together Home page. <http://www.borland.com/us/products/together/index.html>
- [27] Together FAQ http://www.borland.com/resources/en/pdf/products/together/together_faq.pdf
- [28] Eclipse modeling project <http://www.eclipse.org/projects/project.php?id=modeling>
- [29] JSF <http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html>
- [30] JPA <http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html>
- [31] Seam <http://www.seamframework.org/Home>

- [32] JBoss Seam Simplicity and Power Beyond Java EE - Michael Juntao Yuan, Thomas Heute - Prentice Hall
- [33] Maven2 <http://maven.apache.org/>
- [34] Jboss Tools <http://www.jboss.org/tools>
- [35] Eclipse JEE <http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/galileosr2>
- [36] Claudia Pons, Roxana Giandini, Gabriela Pérez. "Desarrollo de Software Dirigido por Modelos. Teorías, Metodologías y Herramientas", Ed: McGraw-Hill Education. (2010).
- [37] Zoo de ATL <http://www.eclipse.org/m2m/atl/atlTransformations/>
- [38] "Catalogue of Model Transformations" de K.Lano
- [39] ATL (ATLAS Transformation Language) <http://www.eclipse.org/m2m/atl/>
- [40] Epsilon Documentation <http://www.eclipse.org/gmt/epsilon/doc/>
- [41] Atlas Model Weaver Project Web Page. <http://www.eclipse.org/gmt/amw/>, 2005.
- [42] SmartQVT Home page. <http://smartqvt.elibel.tm.fr/>
- [43] Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA: visual automated transformations for formal verification and validation of UML models. In: Proceedings 17th IEEE international conference on automated software engineering (ASE 2002), pp. 267–270, Edinburgh, UK. (2002).
- [44] MOFScript Home page - www.eclipse.org/gmt/mofscript/
- [45] Lawley M., Steel J. Practical Declarative Model Transformation with TefKat. Model Transformation in Practice Workshop at MODELS 2005 Conference. Montego Bay, Jamaica. (2005).
- [46] Atom3 home page. http://atom3.cs.mcgill.ca/index_html
- [47] Mola home page <http://mola.mii.lu.lv/>
- [48] D. H. Akehurst, W. G. Howells, K. D. McDonald-Maier, "Kent Model Transformation Language," Proceedings of Model Transformations in Practice Workshop, MoDELS Conference, Montego Bay, Jamaica (2005)
- [49] D. H. Akehurst y S. J. H. Kent, "A Relational Approach to Defining Transformations in a Metamodel," Proceedings of the 5th International Conference on the Unified Modeling Language, Dresden, Germany (2002), pp. 243–258.
- [50] Tomcat <http://tomcat.apache.org/download-60.cgi>

[51] MySQL <http://www.mysql.com/>

[52] Heidi <http://www.heidisql.com/>

[53] Java 2EE <http://www.oracle.com/technetwork/java/javaee/overview/index.html>

[54] Java 2SE <http://www.oracle.com/technetwork/java/javase/overview/index.html>

[55] EJB 3.0 <http://www.oracle.com/technetwork/java/javaee/ejb/index.html>

[56] Anselmo Francisco Abadia, Juan Barisich. Testing basado en modelos: especificación gráfica y derivación de código fuente. 2009.