MD-SPL Cupi2: UNA LÍNEA DE PRODUCTOS BASADA EN MODELOS

CARLOS ANDRÉS PARRA ACEVEDO

UNIVERSIDAD DE LOS ANDES

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS

MAESTRÍA EN INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

BOGOTÁ D.C.

2007

MD-SPL Cupi2: UNA LÍNEA DE PRODUCTOS BASADA EN MODELOS

CARLOS ANDRÉS PARRA ACEVEDO

Tesis de grado

Directora

PhD. Rubby Casallas Gutiérrez

Profesora asociada del departamento de sistemas y computación

UNIVERSIDAD DE LOS ANDES

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS

MAESTRÍA EN INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

BOGOTÁ D.C.

2007

Nota de aceptación

____

____

____

____

Director de proyecto

____

Jurado

____

Jurado

____

Bogotá D.C., enero de 2007

TABLA DE CONTENIDO

1 Introducción 5
2 Planteamiento del problema y objetivos 8
3 Estado del Arte 11
3.1 Arquitectura Basada En Modelos (MDA) 11
3.1.1 Modelos 11
3.1.2 Clasificación de Modelos 12
3.1.2.1 Plataforma 12
3.1.2.2 Modelo independiente de la computación (CIM) 12
3.1.2.3 Modelo Independiente de la Plataforma (PIM) 12
3.1.2.4 Modelo de Plataforma Específica (PSM) 13
3.1.3 Metamodelos y Metametamodelos 13
3.1.3.1 MOF (Meta-Object Facility) 13
3.1.4 Transformaciones 14
3.1.4.1 Reglas de transformación 14
3.1.4.2 Kermeta 15
3.1.4.3 Kermeta como lenguaje de Transformaciones 15
3.1.5 Entrelazado 16
3.2 Líneas De Producto 16
4 Propuesta 18
4.1 Identificación y Creación de Activos 18
4.1.1 Metamodelos 18
4.1.2 Transformaciones 18
4.1.3 Modelos de Rasgos 19
4.1.4 Plantillas de Código 19
4.2 Construcción del Producto 20
4.3 Implementación 20
4.4 Alcances y Limitaciones 21
4.4.1 Características del componente Mundo 21
4.4.1.1 Estructuras de Datos 21
4.4.1.2 Persistencia 22
4.4.2 Características de la interfaz de Usuario 22
4.4.2.1 Vista Principal 22
4.4.2.2 Vista de Extensión 22
4.4.2.3 Vista de Conjunto 22
4.4.2.4 Vista de Información 22
4.4.2.5 Vista de Agregación 22
5 Metamodelos de la MD-SPL de Cupi2 24
5.1 Metamodelo de Mundo 24
5.1.1 Modelos Conformes al Metamodelo de Mundo 25
5.2 Metamodelo de Arquitectura 26
5.2.1 Metamodelo de Negocio 26
5.2.2 Metamodelo de Interfaz 27
5.2.3 Creación de modelos conformes al metamodelo de arquitectura 28
5.2.3.1 Modelo conforme al metamodelo de Negocio (Arquitectura) 28
5.2.3.2 Modelo conforme al metamodelo de Interfaz (Arquitectura) .29

5.3 Metamodelo de Tecnología (Java) 30
5.3.1 Modelo conforme al metamodelo de Java 31
6 Transformaciones de la MD-SPL de Cupi2 33
6.1 Transformaciones de Tipo Modelo a Modelo 34
6.1.1 Reglas de correspondencia 35
6.1.2 Reglas de Control 35
6.1.3 Reglas específicas 36
6.1.4 Expresión de variabilidad a través de las reglas de transformación 36
6.2 Transformaciones Modelo a Modelo para la línea de Cupi2 37
6.2.1 Transformación Mundo a Negocio 37
6.2.1.1 Regla Entrada 37
6.2.1.2 Regla Agrupador 37
6.2.1.3 Regla Simple 38
6.2.1.4 Regla Atributo 38
6.2.1.5 Regla AtributoCupi2 38
6.2.1.6 Regla AsociacionElemento 38
6.2.1.7 Regla PersistenciaArchivo2Negocio 38
6.2.1.8 Regla PersistenciaSerializacion2Negocio 38
6.2.1.9 Regla crearMetodosListas 39
6.2.1.10 Regla crearMetodosListasDobles 39
6.2.1.11 Regla crearServiciosComparables 39
6.2.1.12 Regla crearSetterGetter 39
6.2.2 Transformación Mundo a Interfaz 39
6.2.2.1 Regla Entrada 39
6.2.2.2 Regla agregarVistaInterfaz 39
6.2.2.3 Regla agregarComponenteVisualizacionVista 40
6.2.2.4 Regla agregarComponenteInteraccionVista 40
6.2.2.5 Regla agregarConstructor 40
6.2.2.6 Regla agregarManejador 40
6.2.3 Transformación Arquitectura a Java 40
6.2.3.1 Regla Entrada 40
6.2.3.2 Regla Agrupador2Clase 40
6.2.3.3 Regla Simple2Clase 40
6.2.3.4 Regla Vista2Clase 41
6.2.3.5 Reglas Servicio2Method y Regla ServicioInterfaz2Method 41
6.2.3.6 Reglas AtributoNegocio2Field y AtributoCupi2Negocio2Field 41
6.2.3.7 Reglas Interaccion2Field y Visualizacion2Field 41
6.3 Transformación Modelo a Código Fuente 41
7 Implementación 43
7.1 Entorno de desarrollo Eclipse 43
7.2 Creación de Metamodelos EMF y Omondo 43
7.3 Lenguajes de Transformación 44
7.3.1 ATL 44
7.3.2 Acceleo 45
7.4 Entrelazado de Modelos AMW 46
8 Conclusiones y aportes de la Investigación 48

8.1 Aportes de la investigación 49
8.1.1 Integración de MDA y SPL 49
8.1.2 Definición de los metamodelos 49
8.1.3 Definición de las transformaciones 49
8.1.4 Creación de activos basados en plantillas 49
8.1.5 Automatización del proceso de construcción de productos 50
9 Trabajos Futuros 51
9.1 Implementación de un esquema de trazabilidad 51
9.2 Extender la capacidad de la línea de productos 52
9.2.1 Componente de Pruebas 52
9.2.2 Elementos gráficos 52
9.2.3 Características de otros niveles 52
9.3 Crear dominios para otros lenguajes 52
9.4 Implementar soluciones para problemas reales con características similares 52
10 Bibliografía 54
11 Anexo A - Metamodelos de la MD-SPL Cupi2 56
11.1 Metamodelo de Mundo 56
11.2 Metamodelo de Arquitectura Negocio 58
11.3 Metamodelo de Arquitectura (Interfaz) 63
11.4 Metamodelo de Plataforma Tecnológica (Java) 66
12 Anexo B - Reglas de Transformación 70
12.1 Transformación Mundo A Negocio 70
12.2 Transformación Mundo A Interfaz 77
12.3 Transformación Arquitectura a Java 83

LISTA DE FIGURAS

Figura 1 - Proceso de generación 20
Figura 2 - Dominios de la MD-SPL Cupi2 24
Figura 3 - Metamodelo de Mundo 25
Figura 4 - Modelo para el ejemplo Exposición de Automóviles conforme con el metamodelo de mundo. 26
Figura 5 - Metamodelo de Negocio (Arquitectura) 27
Figura 6 - Metamodelo de Interfaz de usuario (Arquitectura) 28
Figura 7 - Modelo conforme al metamodelo de Negocio (Arquitectura) 29
Figura 8 - Modelo para el ejemplo Exposición de Automóviles conforme al metamodelo de Interfaz (Arquitectura) 30
Figura 9 - Metamodelo de Java 31
Figura 10 - Modelo del componente Mundo conforme al metamodelo de Java 32
Figura 11 - Transformaciones entre los dominios de Cupi2 33
Figura 12 - Encabezado de las transforamaciones ATL 34
Figura 13 - Ejemplo de un *helper* en ATL 34
Figura 14 - Regla Base 35
Figura 15 - Regla de Control 36
Figura 16 - Regla específica 36
Figura 17 - Plantilla para método que carga una estructura de datos 42
Figura 18 - Interfaz de creación de metamodelos de Omondo 44
Figura 19 - Esquema de utilización de ATL [15] 44
Figura 20 - Integración de ATL y Eclipse 45
Figura 21 - Acceleo y Eclipse 46
Figura 22 - Entrelazado a través de AMW 47

1 Introducción

A través de la corta historia del desarrollo de software, se han vivido diversas etapas ocasionadas por el acelerado avance tecnológico. El hardware continúa aumentando su poder de procesamiento y almacenamiento al mismo tiempo que reduce su tamaño.

De manera similar, las tecnologías de comunicación mejoran con el crecimiento de los canales de información y las velocidades de transmisión, haciendo posible el envío de altos volúmenes de datos entre distintos lugares sin importar su ubicación geográfica. Estos avances han hecho posible la creación de software que hasta hace pocos años era inimaginable, incluso para los visionarios más arriesgados.

Hoy en día, es posible encontrar aplicativos en casi cualquier ámbito de la industria, como pueden ser: finanzas, comunicaciones, educación, medicina, entretenimiento, etc. El software, se ha integrado a la vida diaria del común de las personas, variando desde aplicaciones cotidianas como el correo electrónico o la consulta de información en Internet, hasta la manipulación de millones de datos en aplicaciones empresariales robustas.

Se debe aceptar que el software esta en evolución constante. Cada nueva tecnología incrementa las expectativas de los usuarios, induciendo una nueva búsqueda de metodologías de desarrollo que se adapten mejor a los cambios y que mejoren el desempeño, la compatibilidad de los sistemas y la productividad de los equipos de desarrollo. Por consiguiente, seguir el ritmo acelerado de desarrollo que impone la tecnología, es una tarea difícil. Los desarrolladores de software deben tener conocimientos de plataformas y tecnologías específicas, además de habilidades para ofrecer soluciones a los problemas planteados por las reglas del negocio particular.
Para atacar el problema de la complejidad del desarrollo de software, se han ideado nuevas propuestas, dentro de las cuales sobresalen la ingeniería basada en modelos y las líneas de productos de software.

La arquitectura basada en modelos por su parte (MDA Model Driven Architecture), es una propuesta de la OMG (Object Management Group) que brinda un mayor grado de abstracción, separando el mundo del negocio de la plataforma específica de implementación: MDA ofrece una aproximación al desarrollo abierta e independiente de la plataforma. Para esto, se construyen modelos independientes de la plataforma, los cuales expresan la funcionalidad y el comportamiento propios del negocio, luego se integran con los conceptos específicos de la tecnología seleccionada para la implementación, a través de funciones de transformación. Estas transformaciones se encargan de introducir los nuevos conceptos de arquitectura y plataforma específica al modelo de negocio original. Gracias a esta separación propia de MDA, es imaginable partir del mismo modelo de negocio, y conseguir implementaciones en

diferentes tecnologías y plataformas como J2EE, CORBA, o .NET, en teoría, con muy poco esfuerzo adicional.

Por otro lado, las líneas de productos de software (SPL) son una nueva estrategia que busca la optimización de los procesos de desarrollo de software. Por supuesto, el concepto de línea de producto no es nuevo y se ha usado con éxito en otras industrias distintas al software. Sin embargo, el concepto de línea de producto de software es relativamente nuevo, y de acuerdo con los primeros avances en el tema, se espera que los beneficios de su utilización sean equivalentes a los obtenidos en otras áreas con anterioridad. En términos generales, se busca la creación de un conjunto de activos base, entendiendo activo como cualquier artefacto de software, usado para la construcción de más de un producto. La generación de estos activos se puede realizar de dos formas: A partir de productos desarrollados, en cuyo caso, se procede a realizar una identificación de artefactos de software que se puedan aislar y que sean útiles en la producción de otras aplicaciones. La otra forma de generación de activos, es cuando éstos se crean sin ningún producto desarrollado como base, en este caso, cada uno de los activos debe desarrollarse teniendo en cuenta que va a ser utilizado en la producción de distintos productos de software. Cada activo construido debe estar orientado hacia la arquitectura de la línea que describe a la familia de productos por construir, por esto, para que una línea de producto sea útil es necesario delimitar el conjunto de productos. Cada producto miembro de una familia debe compartir algunas características en común con los demás productos.

Esta investigación, busca la materialización de las ideas propuestas tanto por MDA como por las líneas de productos de software. Para esto se propone la creación de una línea de productos de software basada en modelos (MD-SPL). En una MD-SPL los activos son artefactos tales como metamodelos, modelos y transformaciones. Los elementos a partir de los cuales se crean los productos son modelos conformes a metamodelos que especifican conceptos de los diferentes aspectos de los productos. De esta manera se consigue una separación de preocupaciones, permitiendo que los modelos iniciales sean independientes de los conceptos de la arquitectura y tecnologías particulares sobre las cuales se realiza la implementación. Además de los modelos, es necesario, a partir de un modelo inicial hacer un refinamiento y enriquecer dicho modelo con conceptos de arquitectura y tecnología hasta conseguir un modelo destino que representa a un producto de software. Esto es posible gracias a las transformaciones. Una transformación toma un modelo origen y a partir de los conceptos de los metamodelos origen y destino, crea un modelo destino que contiene no sólo la información del modelo origen sino la nueva información especificada en el metamodelo de destino.

Como escenario de experimentación, se han utilizado los ejemplos del proyecto Cupi2 [18]. Este es un proyecto del grupo de construcción de software de la Universidad de los Andes, en el cuál se describe un plan de estudios, con varios niveles, para la enseñanza de la programación de computadores utilizando el lenguaje de programación Java. Los estudiantes deben entender y

dominar los temas mostrados en cada nivel. Para esto, cada nivel está acompañado de una aplicación en donde se ilustran de manera práctica, los temas correspondientes a cada elemento del plan de estudios. Por lo tanto, este proyecto, necesita de muchas aplicaciones semestrales cuyas características de fondo deben ser similares, ya que los temas enseñados son los mismos, pero con cambios en su interfaz y en elementos específicos de cada problema. Construir estas aplicaciones requiere un esfuerzo importante. Por otro lado, la construcción de estos programas, aunque similares, pueden tener defectos que no se desean reflejar en el código fuente que manipula el estudiante.

Nuestra MD-SPL busca automatizar el proceso de creación de estas aplicaciones. Sin embargo, esta propuesta no busca la generación de ejemplos completos. Existen elementos propios de cada ejemplo particular que son específicos al mundo del problema y no pueden ser generados a partir de un conjunto de activos comunes. Estos elementos deben ser creados manualmente por el usuario para completar el ejemplo que está siendo construido.

Este documento está organizado de la siguiente manera. El capítulo 2 presenta el planteamiento del problema. El capítulo 3 muestra una reseña sobre el estado del arte en temas como MDA y líneas de productos de software.
La propuesta de trabajo de la sección 4, describe el proceso, a través del cual, mediante la especificación de un modelo de negocio y la selección de rasgos de tecnología e interfaz se genera una aplicación funcional.
Los capítulos 5 y 6 describen los resultados en términos de activos, como son los metamodelos de mundo, arquitectura y plataforma tecnológica. También las transformaciones necesarias para enriquecer el modelo original con los conceptos de tecnología y finalmente la generación de código fuente a partir de un modelo y un conjunto de plantillas debidamente identificadas y codificadas.
El capítulo 7, describe brevemente el conjunto de herramientas utilizadas en nuestra investigación. Por último se muestran las conclusiones del trabajo realizado y los posibles trabajos futuros.

2 Planteamiento del problema y objetivos

Construir aplicaciones de la manera tradicional, es un proceso que presenta cada vez más inconvenientes en términos de costos de personal, tiempo de desarrollo y calidad de los productos. Los desarrolladores invierten una gran cantidad de tiempo en procesos manuales de codificación y depuración de artefactos de software. Incluso si los artefactos tienen características en común, los esquemas de reutilización no siempre son claros y el hecho de reutilizar elementos desarrollados con anterioridad puede resultar más problemático que desarrollarlos de nuevo.

Para afrontar estos problemas, han surgido nuevas propuestas en donde se busca la disminución del esfuerzo de los desarrolladores en tareas como codificación para abrir paso al modelado y abstracción de alto nivel de los sistemas.

Con MDA por ejemplo, se logra un mayor grado de abstracción, separando el mundo del problema de la plataforma específica. Para esto, se construyen modelos independientes de la plataforma, los cuales expresan la funcionalidad y el comportamiento propios del negocio, luego, estos modelos se integran con los conceptos específicos de la tecnología seleccionada para la implementación. Además, utilizando MDA se pueden optimizar los procesos de combinación de modelos, escritura y generación de código por medio de las transformaciones de modelo a modelo y de modelo a código.

Sin embargo, MDA cuenta con vacíos en su definición, ya que no existen metodologías claras de implementación, hay una gran cantidad de opiniones y puntos de vista diferentes, y la gran mayoría de herramientas están en desarrollo.

Por otro lado, si el problema particular involucra la construcción de un conjunto de aplicaciones con características en común, es imaginable la implementación de una SPL. Las SPL´s, buscan obtener beneficios en tiempo de desarrollo y calidad, a partir de las características en común que comparten los productos. Esto se logra mediante la identificación de un alcance, dentro del cual todos los productos aunque diferentes, comparten elementos en común. De esta manera, se construyen activos, los cuales pueden ser usados en la construcción de distintos productos.

Sin embargo, la implementación de una SPL, abarca grandes retos como por ejemplo la definición de un alcance bien delimitado y un mecanismo de expresión y manejo de variabilidad entre los diferentes productos.

Esta investigación se enfoca en la integración de los dos esquemas antes mencionados, mediante la construcción e implementación de una línea de productos de software basada en modelos (MD-SPL). Nuestros activos son

metamodelos, modelos y transformaciones. Cada modelo conforme a un metamodelo especifica un dominio diferente de los productos para lograr la separación de preocupaciones, es decir, los conceptos del mundo del problema son independientes de los conceptos de la arquitectura y la plataforma tecnológica. Además de los modelos, las transformaciones se utilizan como mecanismo mediante el cual se enriquecen los modelos de negocio con los conceptos de arquitectura y tecnología.

Con el fin de hacer una implementación de nuestra propuesta en un caso del mundo real, hemos seleccionado algunos ejemplos del proyecto Cupi2 como escenario de trabajo.
Cupi2 [18], es el proyecto del grupo de construcción de software de la Universidad de los Andes, para la enseñanza de la programación de computadores utilizando el lenguaje Java. Cupi2 abarca tres cursos: Algorítmica y Programación 1 (APO1), Algorítmica y Programación 2 (APO2) y Estructuras de Datos. A su vez, cada curso se divide en seis niveles para un total de 18 niveles. Cada nivel esta orientado a la enseñanza de un conjunto de objetivos pedagógicos específicos.

La estrategia de enseñanza se basa en el hecho de que los estudiantes utilizan para su aprendizaje ejemplos y ejercicios de aplicaciones completas relacionadas con cada nivel.
Como se describe en [21], una aplicación completa consiste de la descripción del problema, de los requerimientos funcionales, los requerimientos de interfaz usuario, el diseño, las pruebas unitarias y del código que implementa la solución. Para cada nivel se seleccionan los conceptos que se quieren enseñar, los conceptos están alineados con uno o varios ejes temáticos. Por ejemplo, el nivel 7 incluye, entre otros elementos, la enseñanza de algoritmos sobre colecciones como búsquedas y ordenamientos. Así, el objetivo de una aplicación de nivel 7 es facilitar el aprendizaje de estas estructuras. Significa que el código que resuelve un problema de nivel 7 sólo incluye los conceptos de los niveles anteriores y los propios del nivel 7.
El proyecto necesita la construcción semestral de varias aplicaciones ejemplo por cada nivel. El proceso de construcción de estas aplicaciones presenta un conjunto de características que hacen de éste un dominio potencial para MD-SPL. Estas características son:

- Es necesario un gran esfuerzo en tiempos de desarrollo y personal para la construcción de todos los ejemplos para cada semestre.
- Los ejemplos son aplicaciones que comparten un grupo importante de elementos comunes.
- Los ejemplos son utilizadas por un conjunto grande de usuarios, es deseable que la calidad y los estándares se mantengan a lo largo de todos los ejemplos.
- Durante la construcción manual de cada ejemplo, es posible que se inyecten defectos, que se reflejan en el código que manipulan los estudiantes.

No obstante, Cupi2 propone un desarrollo incremental de habilidades en los estudiantes, por lo tanto es un proyecto que abarca diversas temáticas que van desde los conceptos básicos de programación orientada a objetos, hasta temas avanzados como manejo de bases de datos o aplicaciones Cliente/Servidor. Esta diversidad en los temas implica grandes diferencias entre los ejemplos usados para cada nivel.
Por esta razón y para efectos prácticos de la investigación, hemos limitado el alcance de nuestra línea a un grupo reducido de objetivos pedagógicos presentes en los ejemplos de los niveles 3 al 9. De esta manera, se busca incrementar el conjunto de elementos en común de las aplicaciones, que es la base para una línea de producto exitosa.

Partiendo de este subconjunto de ejemplos, mediante la integración de los conceptos establecidos por MDA y las SPL, es imaginable la construcción de un MD-SPL, que a partir de la identificación de elementos comunes y variables, esté en la capacidad de generar los ejemplos, manteniendo los estándares de calidad establecidos y reduciendo el esfuerzo y los recursos empleados en la actualidad.

Por otro lado, uno de los compromisos principales en una MD-SPL es el manejo de las características comunes y diferentes para generar productos que se ajusten a las preferencias del usuario. La expresión de la variabilidad a lo largo de los ejemplos, es un proceso complejo. En nuestro caso, se ha realizado el trabajo en conjunto con Kelly Garcés[12]. En su trabajo se describe el proceso completo de descripción y utilización de la variabilidad, mediante la identificación de rasgos, y sus implicaciones mediante la utilización de distintos tipos de transformaciones entre los modelos.

El objetivo general de nuestra investigación es la construcción de una línea de producto con activos de ingeniería basada en modelos, capaz de construir artefactos relacionados con los objetivos pedagógicos de los niveles 3 a 9 del proyecto Cupi2

Este objetivo involucra un conjunto de objetivos específicos como son:

1. Crear los diferentes metamodelos que describen los conceptos relacionados con:
    - Mundo del problema
    - Arquitectura
    - Plataforma específica de desarrollo (Java).
2. Construir las transformaciones entre los metamodelos y las plantillas de generación de código fuente.
3. Integrar un esquema que permita definir y manejar la variabilidad entre los productos.
4. Implementar la solución sobre un conjunto de herramientas apropiadas para el desarrollo de software mediante MDA y SPL.

3 Estado del Arte

3.1 Arquitectura Basada En Modelos (MDA)

La arquitectura basada en modelos es un marco de trabajo para el desarrollo de software definido por la OMG. Como se manifiesta en [13], la clave de MDA es la importancia que se da a los modelos en el proceso de desarrollo de software. Por consiguiente, las actividades centrales se enfocan en crear modelos del sistema que se desea construir. La utilización de estos modelos ofrece mayor flexibilidad en los siguientes campos:

- Implementación: La implementación de nuevas tecnologías puede ser realizada bajo los diseños existentes.
- Integración: Tomando como base el diseño de los modelos y no sólo su implementación, es posible automatizar la generación de esquemas de integración.
- Mantenimiento: Tener un diseño que sea entendido por una máquina brinda a los desarrolladores acceso directo a la especificación de los sistemas, haciendo del mantenimiento de las aplicaciones una tarea más simple.
- Pruebas y Simulación: Como los modelos desarrollados pueden usarse para generar código, pueden ser usados también para hacer validaciones con respecto a los requerimientos.

3.1.1 Modelos

La piedra angular de toda la propuesta de MDA son los modelos. Un modelo de un sistema es una descripción o especificación de ese sistema y su entorno para cierto propósito Un modelo se presenta con frecuencia como una combinación de dibujos y texto. A su vez, el texto pude estar en un lenguaje de modelamiento o en lenguaje natural. Al igual que los sistemas, los modelos pueden ser estáticos o dinámicos. Cuando un modelo es estático es posible razonar sobre él [5]. Una característica que se espera que cumpla un modelo que represente un sistema es que éste pueda responder a determinadas preguntas, de forma similar a como lo haría el mismo sistema (esto es lo que se conoce como sustitución contextual) [5]. Los modelos también, de acuerdo con la forma como sean construidos, pueden ser prescriptivos o descriptivos. Los primeros se refieren a los modelos que se generan con el objeto de guiar la construcción del sistema que representan (es decir, el sistema que especifican aún no existe). Los descriptivos, por su parte, son generados a partir de sistemas ya existentes, y se usan principalmente para el entendimiento de los mismos [5].

Un factor que da solidez a la propuesta de MDA, es el hecho de que el modelado es una actividad que los seres humanos realizan constantemente, en diversas áreas (tales como física, economía, medicina) y no sólo en computación. Esto se da principalmente debido a que con los modelos es más fácil el entendimiento de los fenómenos que se presentan en la realidad.

3.1.2 Clasificación de Modelos

La definición de modelo dada en la sección anterior, es muy amplia y podría abarcar modelos de muchas clases. Es por esto que en MDA se hace una clasificación de los tipos de modelos, de acuerdo con el grado de descripción que tenga del sistema que representa y a la inclusión o no, de elementos concernientes a arquitectura y plataforma.

3.1.2.1 Plataforma

Una plataforma es un grupo de subsistemas y tecnologías que proveen un conjunto coherente de funcionalidades a través de interfaces y patrones específicos. Una aplicación soportada por alguna plataforma, puede hacer uso de sus servicios sin que importen los detalles de cómo se implementa la funcionalidad subyacente. De esta manera, surge una nueva propiedad de los modelos, esta es la independencia de la plataforma. Un modelo es independiente de la plataforma cuando el modelo no incluye las características especificadas por la plataforma para que el sistema modelado pueda hacer uso de los servicios provistos por esta última. La independencia de la plataforma puede tener una escala. Así, un modelo puede asumir solamente la disponibilidad de características de cualquier plataforma en general, como la invocación remota de servicios. A la vez, otro modelo diferente puede asumir la existencia de un conjunto de herramientas específicas para una tecnología particular como por ejemplo CORBA que permite la invocación remota. Esto significa que mientras algunos modelos pueden asumir la existencia de una característica general, otros modelos pueden estar ligados a una tecnología particular. [13]

3.1.2.2 Modelo independiente de la computación (CIM)

Un modelo independiente de la computación es una vista del sistema modelado que no hace referencia a los detalles de la estructura del sistema. El CIM también es conocido como el modelo del dominio y se expresa en un vocabulario que es familiar a los expertos del dominio. El modelo independiente de la computación juega un papel importante en la comunicación entre los expertos en el dominio del problema y sus requerimientos, y los expertos en el diseño y construcción de artefactos que satisfacen los requerimientos de ese dominio particular.

3.1.2.3 Modelo Independiente de la Plataforma (PIM)

Un modelo independiente de la plataforma es aquel que muestra un grado específico de independencia de plataforma de tal manera que puede ser usado por diferentes plataformas de un tipo similar. Una técnica común para crear un modelo independiente de la plataforma es enfocar el modelo del sistema hacia una maquina virtual. Una maquina virtual se define como un conjunto de partes y servicios que son definidos independientemente de una plataforma especifica y que se concretizan en diferentes formas en cada plataforma especifica, Una maquina virtual es una plataforma, por lo tanto un modelo de esta clase, es

especifico a esa plataforma, pero ese modelo a la vez es independiente de las diferentes plataformas en las cuales la maquina virtual se puede implementar.

3.1.2.4 Modelo de Plataforma Específica (PSM)

Un modelo específico de la plataforma combina la especificación del PIM con los detalles que especifican como ese sistema usa un tipo particular de plataforma.

3.1.3 Metamodelos y Metametamodelos

En MDA se establece la existencia de dos relaciones: "representado por" (representedBy) y "conforme a" (conformsTo) [5]. La primera de ellas se aplica a la relación entre un sistema y un modelo con el que se encuentre representado (es decir, un sistema esta representado por un modelo).
La segunda relación esta en un nivel más alto de abstracción e indica que un modelo es conforme a un metamodelo. Esto quiere decir que un modelo se encuentra especificado por un metamodelo. El metamodelo a su vez debe estar descrito en un lenguaje, representado en un metametamodelo. Este debe ser único en el espacio técnico en el que se este trabajando para poder efectuar operaciones entre modelos, tal como transformaciones, combinaciones y comparaciones. [5]

3.1.3.1 MOF (Meta-Object Facility)

En el caso de MDA, el único meta-metamodelo definido es MOF (Meta Object Facility). MOF es un estándar de la OMG que especifica un lenguaje abstracto para describir otros lenguajes. El propósito de MOF es definir los conceptos básicos para el modelamiento y la estandarización del diseño de meta-modelos y de sus modelos resultantes. Como se muestra en [4] MOF también es conocido como el meta-meta-modelo en el marco de trabajo de MDA en donde se describe una arquitectura de cuatro capas como se muestra a continuación:

- M0: Es el nivel de objetos de usuario y comprende la información que se quiere describir (El mundo real). Esta información es denominada comúnmente como "datos".
- M1: Es el nivel de los modelos, y se compone de los meta-datos que describen la información del nivel anterior (M0).
- M2: Es el nivel de los meta-modelos. En este nivel se encuentran las descripciones (meta-meta-datos) que definen la estructura y la semántica de los meta-datos. Un meta-modelo puede también ser visto como un lenguaje para describir diferentes tipos de datos.
- M3: Es la capa del meta-meta-modelo. Aquí se definen la estructura y la semántica de los meta-meta-datos. En otras palabras es el lenguaje para definir diferentes tipos de meta-datos.

3.1.4 Transformaciones

De forma general, en [7] se describe una transformación como el proceso de convertir un modelo en otro modelo del mismo sistema. Un problema que presenta MDA, es que no establece fundamentos para transformar los modelos independientes de la plataforma (PIM) a modelos dependientes de la plataforma (PSM). Por esta razón la OMG realizó una convocatoria para recibir propuestas a este respecto [7]. Esta convocatoria arrojó una serie de enfoques que se clasifican en [7], mediante la definición de los elementos que deberían estar presentes en un modelo de transformaciones. Entre estos elementos, se encuentran las reglas de transformación, la relación entre el modelo fuente y el modelo destino, y el sentido de la transformación (unidireccional o bidireccional). De las reglas también se establece su alcance de aplicabilidad (tanto en el modelo fuente como en el modelo destino), la estrategia de aplicación (que puede ser interactiva, determinística o no determinística), el orden en que son aplicadas, y la forma en las que son organizadas (por medio de mecanismos de reutilización, modularidad o dándoles una estructura de acuerdo a un lenguaje) [7].

Los enfoques de transformación que existen actualmente, pueden agruparse en dos grandes categorías [7]: "modelo a código" y "modelo a modelo". Tales categorías tienen a su vez subcategorías. Los enfoques basados en *visitors* (que proveen un mecanismo para recorrer los diferentes elementos que conforman un modelo e imprimir en un flujo de texto el código relacionado con estos) y los enfoques basados en *templates* (donde las plantillas contienen el código al que se desea realizar la transformación con espacios que son llenadas a partir de información obtenida del modelo fuente) pertenecen a la categoría modelo a código [7].

Dentro de la categoría de transformaciones modelo a modelo se encuentran los enfoques de manipulación directa (en los cuales los usuarios tienen que hacer prácticamente todo, y se les brinda poca o ninguna ayuda para la definición de las reglas así como el orden en que son aplicadas), los enfoques relacionales (en los que se utilizan relaciones matemáticas para establecer el tipo de relación entre los distintos elementos del modelo fuente y el modelo destino), los enfoques basados en transformaciones gráficas (que siendo los más poderosos también son los más complejos), los enfoques manejados por estructuras (en los que el usuario define las reglas de transformación y las herramientas se encargan de determinar el orden y estrategia de aplicación para estas), y los enfoques híbridos (en los que se usa una combinación de los enfoques anteriores) [7].

3.1.4.1 Reglas de transformación

Un modelo de transformación se conforma de un conjunto de reglas de transformación, las cuales especifican la forma en que los elementos del origen deben ser creados en términos del destino.

De acuerdo con [7], una regla de transformación tiene dos partes: izquierda y derecha. La parte izquierda se encarga de acceder al modelo origen, mientras la parte derecha se encarga de hacer las operaciones necesarias sobre el

modelo destino. Tanto la parte izquierda como la derecha pueden ser representadas mediante la utilización de las siguientes características:
Variables: Las variables contienen elementos del modelo origen o destino (o de elementos intermedios).
Lógica: La lógica expresa cálculos y restricciones sobre los elementos de los modelos. La lógica puede ser ejecutable o no ejecutable. Si es no ejecutable se usa para especificar una relación entre los modelos. Por otro lado, cuando la lógica es ejecutable, puede ser de forma declarativa o imperativa.

En [14] se describe la diferencia entre los lenguajes declarativos y los imperativos. Un lenguaje es declarativo si sus reglas de transformación especifican las relaciones entre los elementos de los modelos origen y destino, sin involucrar un orden de ejecución. En contraste, cuando un lenguaje de transformación es imperativo, éste especifica una secuencia implícita de pasos para ser ejecutados con el fin de producir un resultado. Puede existir una categoría intermedia, conocida como hibrida. En esta categoría los lenguajes tienen una mezcla de instrucciones declarativas e imperativas. Como se menciona en [14], una transformación escrita en este tipo de lenguajes puede tener un conjunto de reglas declarativas que describen las relaciones entre los modelos origen y destino. Adicionalmente cada regla, puede tener fragmentos de código imperativo, que realizan acciones adicionales para producir el resultado deseado.

3.1.4.2 Kermeta

En MDA los meta-lenguajes como MOF, EMOF o Ecore son cada vez más utilizados, sin embargo, estos meta lenguajes se enfocan en la estructura y aunque abarcan conceptos como meta operaciones, no ofrecen un marco de trabajo en el cual se pueda hacer una definición de las acciones de estas operaciones. Kermeta [20] propone la extensión de MOF para integrar conceptos que soporten la descripción del comportamiento de los metamodelos. Fue creado por el grupo Triskell (IRISA). Algunas características de Kermeta son:

1. Es un lenguaje imperativo de dominio específico orientado a objetos. Su enfoque es la manipulación de metamodelos.
2. Es una extensión de EMOF que permite la adición de comportamiento.

3.1.4.3 Kermeta como lenguaje de Transformaciones

A pesar de la motivación presente en la creación de Kermeta, en donde se busca la adición de comportamiento a los metamodelos, sus capacidades se extienden a otros aspectos útiles en los procesos de MDA como son las transformaciones de modelos e incluso el entretejido de aspectos.

Kermeta ofrece las posibilidades de manipulación de modelos conformes a metamodelos. De esta manera, es posible crear nuevos modelos “destino” a

partir de modelos “origen” de acuerdo a las necesidades particulares del desarrollador.

Su sintaxis es muy similar a la de lenguajes de programación actuales como Java. Esto, sumado a su naturaleza imperativa, hace que sea un lenguaje familiar a la mayoría de desarrolladores de software y conlleva a un rápido aprendizaje.

No obstante, cuando se usa Kermeta para realizar transformaciones, las responsabilidades de navegación de los modelos junto con el orden de ejecución de la transformación están en manos del desarrollador. Esto hace que el código de la transformación sea difícil de entender, especialmente cuando se manipulan modelos con muchos elementos. Además un cambio en un metamodelo puede tener un gran impacto en la transformación pues hay entender la forma en que se realiza el recorrido de los modelos origen.

3.1.5 Entrelazado

Además de las transformaciones, el entrelazado es otra operación importante en MDE. El objetivo del entrelazado es especificar relaciones tipadas entre elementos de distintos modelos [9]. Las relaciones son capturadas en un modelo de entrelazado que es conforme a un metamodelo, el cual define los tipos de relaciones. El entrelazado se puede realizar a nivel de M1 o a nivel de M2, es decir, a nivel de modelos o de metamodelos. A diferencia de las transformaciones, las cuales son ejecutadas por un mecanismo de transformación, el entrelazado es hecho de forma manual por el usuario [8]. Cuando la operación de entrelazado es realizada se genera un tercer modelo, llamado modelo de entrelazado, el cual es conforme a un metamodelo de entrelazado.

3.2 Líneas De Producto

Las líneas de producto son un esquema que ya se ha utilizado en otras industrias a la del software con bastante éxito, permitiendo la optimización de procesos y recursos mediante el aprovechamiento de los elementos en común que comparten algunos tipos de productos.
Por ejemplo, come se menciona en [19], la empresa norteamericana Boeing construyó una línea de producto para fabricar dos de sus aviones más populares, el 757 y el 767. A pesar de que son aeronaves diferentes, según Boeing, estas máquinas comparten un 60% de partes en común, y gracias a la estrategia basada en líneas de producto, se logró una reducción de los costos en distintas etapas del proceso como: fabricación de partes, ensamblaje y mantenimiento. De la misma manera, otras industrias, se han visto beneficiadas por la utilización de este tipo de estrategias.

Con respecto a la ingeniería de software, construir sistemas a partir de un conjunto de partes previamente construidas representaría un avance notable, pues los beneficios en tiempos de construcción, utilización de recursos y calidad de los productos serían considerables. Sin embargo, las implementaciones reales de este esquema son limitadas y aún se continúa en fase de exploración e investigación. Ya existe una base bibliográfica sobre el tema, en donde se destaca el marco de trabajo para las líneas de productos de software [19], creado por los investigadores del Instituto de Ingeniería de Software (SEI) de Carnegie Mellon University.

Como se describe en [19], una línea de productos de software (SPL) es un conjunto de sistemas que comparten un grupo de características administrables, las cuales satisfacen las necesidades específicas de un mercado o segmento particular y que son desarrolladas a partir de un conjunto común de activos base. En el ámbito de SPL, un activo base es un artefacto de software que se usa en la producción de más de un producto. Un activo base puede ser un componente de software, un modelo de proceso, un plan, un documento o cualquier otro elemento útil en la construcción de un sistema.

Uno de los activos más importantes en una línea de producto es la arquitectura de software para la línea. Esta debe distribuir las necesidades del conjunto completo de productos y proveer un contexto en el cual otros activos como plantillas de código y artefactos de pruebas puedan ser desarrollados con la flexibilidad necesaria para satisfacer las necesidades de los distintos productos de la línea. Otro activo importante es el alcance de la línea. Este delimita al conjunto de productos que los activos base son capaces de producir. El alcance define las características en común y la variabilidad (formas en las que difieren los productos). De esta manera surgen las familias de productos. Como se menciona en [22], familia de producto hace referencia al grupo de productos de software que pueden ser construidos a partir de un conjunto común de activos. Los productos en una familia por lo general comparten elementos de diseño, componentes y normas para la integración del sistema. El tamaño de la familia depende de la capacidad de la línea para unificar los activos base en un sistema funcional en el que se administran conceptos relativos a reglas de negocio, arquitectura y plataforma de desarrollo.

Existen tres actividades principales involucradas en el desarrollo de una línea de productos de software. El desarrollo de activos, el desarrollo de productos usando activos y la administración de la línea. Tanto del desarrollo de activos como el desarrollo de productos son actividades complementarias y no tienen un orden específico. Por ejemplo es posible crear activos partir de productos desarrollados, en cuyo caso, se procede a realizar una identificación de artefactos de software que se puedan aislar y que sean útiles en la producción de otras aplicaciones. La otra forma de generación de activos, es cuando estos se crean sin ningún producto desarrollado como base, en este caso, cada uno de los activos debe desarrollarse teniendo en cuenta que va a ser utilizado en la producción de distintos productos de software.

4 Propuesta

Como se mencionó en las secciones previas, este trabajo de investigación se centra en la utilización de activos MDA para la implementación de una Línea de productos de Software.

El desarrollo de una línea de productos se basa en dos actividades fundamentales, la creación de activos y la construcción de productos a partir de los activos. Sin embargo, en nuestro caso, cada una de estas actividades tiene componentes especiales debido a la naturaleza de los activos y a la definición y manejo de la variabilidad sobre los productos.

4.1 Identificación y Creación de Activos

En las SPL, se pueden crear activos de dos formas diferentes. Haciendo un análisis de un conjunto de productos previamente construidos, en donde se buscan componentes de cada producto que puedan ser reutilizados en la creación de nuevos productos. La otra forma es cuando los activos se construyen de cero, sin productos desarrollados, teniendo en cuenta las características comunes y variables de la familia de productos que se desean crear.

Nosotros basamos la creación de activos a partir de productos existentes. Para esto, identificamos elementos en común y elementos variables a lo largo de un subconjunto de ejemplos del proyecto Cupi2. Estos elementos dan paso a la creación de activos que corresponden a metamodelos, transformaciones, modelos de rasgos y plantillas de código como se describe a continuación.

4.1.1 Metamodelos

Los metamodelos describen un dominio particular de los sistemas que se están modelando. En nuestro caso, se propone una división en tres dominios diferentes para la expresión de las diferentes características de los ejemplos Cupi2. Estos dominios son el dominio del mundo, el dominio de la arquitectura que incluye la interfaz gráfica de usuario (GUI), y el dominio de la plataforma tecnológica. Esta separación nos permite administrar la variabilidad de las líneas de producto de manera independiente para cada dominio. Es decir, podemos expresar las características comunes y variables de cada dominio, para luego construir aplicaciones con características variables en cada uno de los dominios permitiendo una mayor flexibilidad y simplicidad.

4.1.2 Transformaciones

Nuestra estrategia general para la creación de diferentes aplicaciones de la línea, de acuerdo con el enfoque MDA, se basa en la transformación

automática de modelos hasta obtener aplicaciones completas. Así, se parte de un modelo origen en el dominio del mundo, y se transforma en un modelo destino en el dominio de la arquitectura; luego, a partir del modelo generado en el dominio de la arquitectura, se genera un modelo destino en el dominio de la plataforma tecnológica; finalmente, se transforma a código fuente el modelo de plataforma tecnológica que ya incluye las características de lógica de negocio y de arquitectura. Para llevar a cabo este proceso se utilizan dos tipos de transformaciones descritas en el capítulo 3. Las transformaciones de tipo Modelo-Modelo y las transformaciones de tipo Modelo-Texto. Las transformaciones de tipo Modelo-Modelo se usan para transformar el modelo de la lógica de negocio a través de los diferentes dominios. Por último, la transformación Modelo-Texto se usa para generar el código fuente del producto a partir del modelo de la plataforma tecnológica, junto con una serie de plantillas de código.

4.1.3 Modelos de Rasgos

En nuestra propuesta, es necesario describir el esquema mediante el cuál se identifican y manejan los componentes variables de los distintos ejemplos de la familia de productos.

La meta es lograr una mejor descripción del sistema a partir del modelo inicial.

Para poder desarrollar aplicaciones siguiendo esta estrategia, se crean modelos de rasgos para expresar las características comunes y variables de cada dominio destino. Así, creamos modelos de rasgos para el dominio de la arquitectura, y el dominio de la plataforma tecnológica.

Los rasgos se relacionan con un conjunto de reglas de transformación que luego son usadas para ejecutar transformaciones personalizadas, de acuerdo a las selecciones del usuario.

Para esto, las reglas de transformación se clasifican en tres tipos diferentes: (1) reglas base, (2) reglas de control, y (3) reglas específicas. Las primeras reglas corresponden a reglas de transformación escritas de forma declarativa, y son utilizadas para generar las características comunes de los productos. Las reglas de control se implementan de forma mixta (declarativa e imperativa), y se encargan de escoger qué reglas específicas se deben ejecutar. Las reglas de transformación específicas se implementan de forma imperativa y son las encargadas de generar los elementos necesarios de acuerdo con las características esperadas en el modelo destino de la transformación.

4.1.4 Plantillas de Código

Las plantillas, representan pedazos de código reutilizables. Se crean a partir de los productos desarrollados y se modifican de acuerdo al esquema de la línea para lograr la reutilización en otros productos. En nuestro caso, cada plantilla corresponde a un método. Los datos que necesita la plantilla se obtienen del modelo conforme al metamodelo de la plataforma tecnológica obtenido a través de las transformaciones entre modelos.

4.2 Construcción del Producto

La figura 1 ilustra el proceso mediante el cual se construyen nuevos productos. La estrategia general para la creación de aplicaciones de la línea se basa en la transformación de modelos hasta obtener aplicaciones completas. Así, nosotros partimos de un modelo fuente en el dominio del mundo, y lo transformamos hasta obtener código fuente con características de la lógica de negocio, de arquitectura, y de plataforma tecnológica.

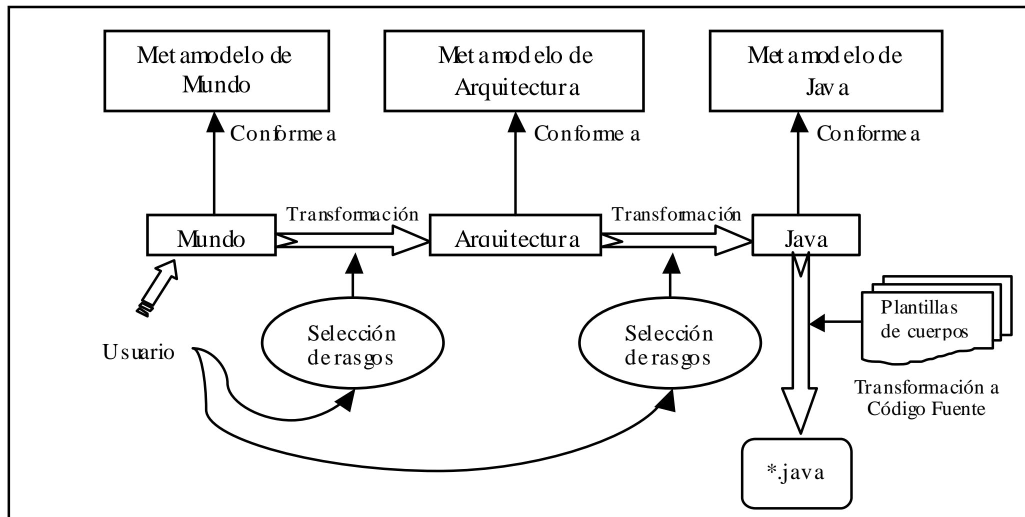


Figura 1 - Proceso de generación

4.3 Implementación

Para la construcción del esquema antes mencionado, es necesario contar con un grupo de herramientas para el desarrollo de cada uno de los activos.
La primera herramienta necesaria es un manejador de modelos. Esta herramienta debe permitir la creación de metamodelos para los diferentes dominios. Además debe ser posible la creación de distintos modelos conformes a los metamodelos previamente creados para cada sistema particular que se desee modelar.
La selección de rasgos, se hace a nivel de modelos. De esta manera, se necesita una herramienta que permita entrelazar modelos.
Para las transformaciones, es necesario usar un lenguaje de transformación que permita hacer correspondencias entre un dominio de origen y un dominio destino. Además se requiere otro lenguaje que permita transformar un modelo en texto. Para finalizar, es ideal contar con un entorno de desarrollo integrado en donde estas herramientas se integren de forma adecuada.

4.4 Alcances y Limitaciones

El proyecto Cupi2 abarca temáticas importantes de un curso de programación. Las temáticas están organizadas en niveles, y cada nivel tiene asociados ejemplos y ejercicios para facilitar la enseñanza/aprendizaje de dichas temáticas. Para nuestra investigación nos basamos en la familia de productos conformada por los distintos ejemplos del proyecto Cupi2.

Todos los ejemplos tienen en común que son aplicaciones monousuario sin requerimientos no funcionales complejos. Todos los ejemplos son escritos en java y se estructuran por tres componentes: el mundo, la interfaz y las pruebas. El componente mundo implementa los conceptos del negocio, sus atributos y relaciones. El componente interfaz usuario implementa la visualización de la información y la interacción entre el usuario y componente del mundo. El componente de las pruebas implementa pruebas unitarias de la funcionalidad ofrecida por el componente mundo.

En el estudio de los distintos ejemplos, hemos encontrado una gran cantidad de características para los diferentes niveles, por esta razón, en nuestra línea de producto hemos dejado de lado el componente de pruebas unitarias y hemos limitado el posible conjunto de aplicaciones que queremos desarrollar, con el fin de identificar un número manejable de características comunes y variables. Nuestra línea sólo contempla las características descritas a continuación, encontradas en los ejemplos para los niveles 3 al 9.

4.4.1 Características del componente Mundo

4.4.1.1 Estructuras de Datos

Las estructuras de datos están presentes en los ejemplos seleccionados y conforman el núcleo de la aplicación. El mundo se representa mediante estructuras de datos de tipo agrupaciones.

Las estructuras de datos que representan las agrupaciones pueden ser de distintos tipos. A continuación se listan los tipos de estructuras de datos incluidos en el alcance de nuestro proyecto.

1. Contenedores de tamaño fijo
2. Contenedores de tamaño variable
3. Listas encadenadas
4. Listas doblemente encadenadas

Cada tipo de estructura de datos implica un conjunto de servicios asociados para su manipulación. Por ejemplo, los algoritmos necesarios para recorrer o insertar un elemento en una lista encadenada son diferentes a los usados para realizar las mismas operaciones sobre una colección de objetos.

4.4.1.2 Persistencia

Persistir la información de los elementos de mundo puede lograrse a través de archivos planos o de técnicas propias de la tecnología En nuestra línea cubrimos los aspectos relacionados con manipulación de archivos y serialización de objetos. En la persistencia sobre archivos planos las operaciones de entrada/salida son codificadas en métodos que cargan y guardan la información en archivos con un formato predeterminado. Por otro lado, la serialización de objetos se basa en la utilización de los servicios que ofrece Java para tal fin.

4.4.2 Características de la interfaz de Usuario

En Cupi2, no existe una forma predeterminada de representar los elementos del mundo en términos de componentes gráficos. La interfaz se forma de un conjunto de vistas. El usuario puede seleccionar una o varias vistas para representar la información de un elemento del mundo. Con propósitos pedagógicos, cada una de las vistas posibles que se utilizan en los ejemplos, tienen definido un conjunto de responsabilidades particulares que el desarrollador puede utilizar para representar su mundo particular.

A continuación se lista el conjunto de vistas que hemos identificado y seleccionado para el ámbito de nuestra línea de productos

4.4.2.1 Vista Principal

Es la vista que agrupa a las demás vistas. Además, a través de esta vista se realiza la comunicación entre el mundo y la interfaz.

4.4.2.2 Vista de Extensión

Es una vista que se utiliza para agregar funcionalidad al aplicativo, consiste en un conjunto de botones y métodos asociados. Sin embargo, en estos métodos sólo se despliega un mensaje de notificación. Las nuevas funcionalidades o extensiones que se hagan al ejercicio, se realizan manualmente por el estudiante a partir de las instrucciones del profesor.

4.4.2.3 Vista de Conjunto

Esta vista se encarga de visualizar un conjunto de elementos. Utiliza componentes como listas o tablas y puede ofrecer adicionalmente servicios que manipulen el orden de presentación de los elementos dentro del conjunto.

4.4.2.4 Vista de Información

La vista de información se encarga de mostrar los datos particular es de un elemento del mundo del problema. Además de información alfanumérica, esta vista también puede mostrar imágenes asociadas a los elementos.

4.4.2.5 Vista de Agregación

La vista de agregación se utiliza para ingresar nueva información de elementos. Sus componentes son similares a los de la vista de información, pero ésta utiliza cuadros de texto y botones de confirmación para recopilar la información ingresada por el usuario de la aplicación.

Cada vista puede incluir elementos como listas, listas desplegables, etiquetas, cuadros de texto, botones, imágenes, botones de radio, y cajas de chequeo. Otros tipos de vistas y elementos gráficos distintos a los ya mencionados no se incluyen en el alcance de nuestro trabajo.

5 Metamodelos de la MD-SPL de Cupi2

Los metamodelos se usan en MDA, para lograr la separación de preocupaciones a través de la representación de dominios particulares. En nuestra investigación, hemos identificado tres dominios diferentes a saber: mundo, arquitectura y plataforma tecnológica. Cada dominio describe un conjunto de características de los miembros de la familia de productos (ver figura 2) a través de metamodelos. La separación de cada dominio, permite modelar los conceptos del mundo del problema independientemente de los conceptos de la arquitectura y de la plataforma tecnológica.

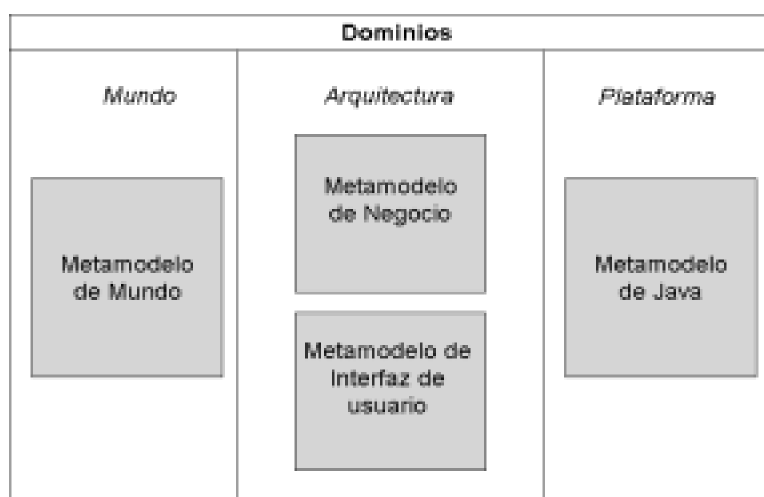


Figura 2 - Dominios de la MD-SPL Cupi2

Los metamodelos de los diferentes dominios involucrados y sus principales conceptos se presentan a continuación.

5.1 Metamodelo de Mundo

El metamodelo del mundo, es el dominio donde se describen los elementos del mundo del problema de un ejemplo Cupi2. Se ha realizado una abstracción para delimitar un grupo de conceptos que permitan modelar cualquier mundo imaginable.
En términos generales, estos ejemplos describen un conjunto de elementos relacionados entre sí a través de una estructura de agrupamiento.

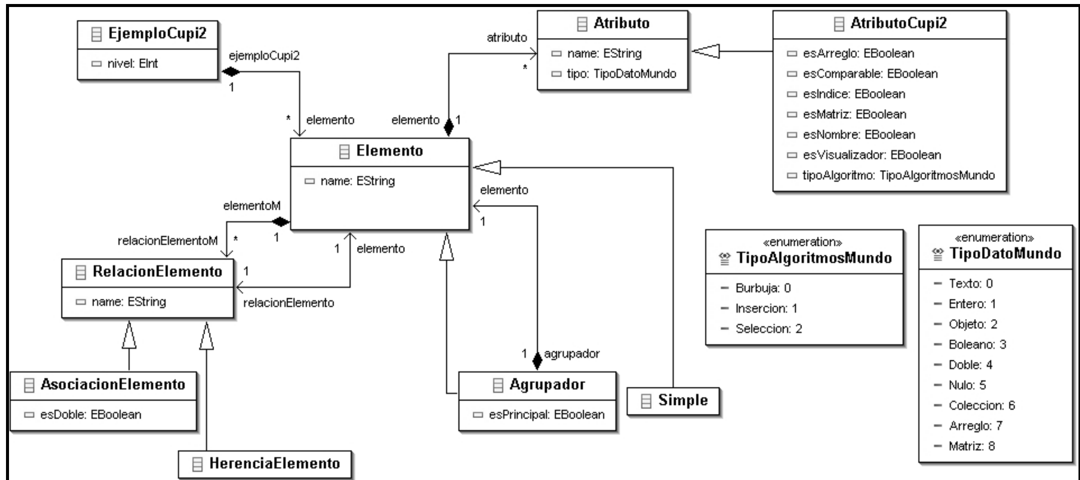


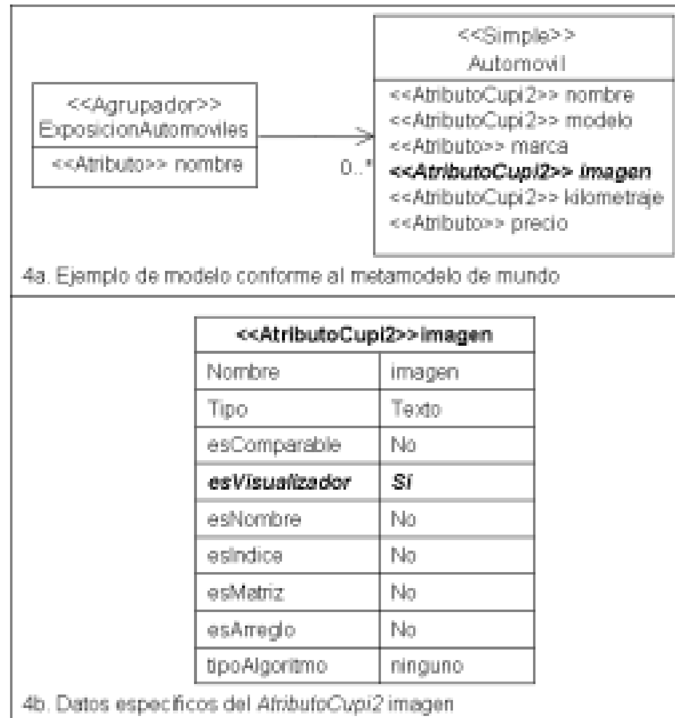
Figura 3 - Metamodelo de Mundo

La figura 3 muestra el metamodelo de mundo. Se pueden observar conceptos como *Agrupador* y *Elemento* y la relación de agregación entre ellos. Si el elemento es *Simple* entonces no puede estar asociado con otros elementos. Los conceptos de relaciones de *Asociación* y *Herencia*, se usan para poder expresar relaciones diferentes a las agrupaciones directas entre el *Agrupador* y el *Elemento*. Por ejemplo, la *Asociación* se usa para crear estructuras lineales como listas enlazadas, mientras que la herencia define una relación de especialización entre dos elementos.
También se tiene el concepto de *AtributoCupi2* como una especialización del *Atributo*. Sus propiedades caracterizaran al *Elemento* del mundo de acuerdo con los requerimientos particulares de la aplicación que se va a modelar. Por ejemplo, se tiene la propiedad *escomparable* que en los modelos conformes a este metamodelo, indica que el atributo puede ser comparado y por lo tanto tendrá un tratamiento especial en el momento de construir los algoritmos de búsqueda o de ordenamiento.

5.1.1 Modelos Conformes al Metamodelo de Mundo

Los modelos conformes al metamodelo de mundo, utilizan sus conceptos para expresar las características propias del dominio que representan. La Figura 4a presenta el modelo para un mundo particular, en este caso, una exposición de automóviles. Aquí se utilizan los estereotipos *Agrupador*, *Simple, Atributo y AtributoCupi2* para indicar la relación de conformidad entre los elementos del modelo y los del metamodelo del mundo. La clase *ExposicionAutomoviles* es conforme al elemento *Agrupador* que fue definido en el metamodelo. De acuerdo con la definición de Agrupador, un concepto de este tipo debe agrupar un conjunto de otros elementos. En el caso del ejemplo, estos elementos son de la clase *Automóvil*. Cada *Automóvil* a su vez contiene atributos y atributos de Cupi2. Si los atributos son de Cupi2, estos presentan un conjunto de datos adicionales. En la figura 4b, se muestran los datos específicos del atributo imagen perteneciente al elemento Automóvil. La imagen tiene seleccionada la

opción esVisualizador, de esta manera, se logra que la imagen se manipule de una forma diferente y se muestre en la vista correspondiente de la interfaz de forma correcta.



<<AtributoCupi2>>imagen	
Nombre	imagen
Tipo	Texto
esComparable	No
esVisualizador	***Sí***
esNombre	No
esIndice	No
esMatriz	No
esArreglo	No
tipoAlgoritmo	ninguno

Figura 4 - Modelo para el ejemplo Exposición de Automóviles conforme con el metamodelo de mundo.

5.2 Metamodelo de Arquitectura

El dominio de la arquitectura, abarca los temas de diseño de los ejemplos Cupi2, sin embargo no se incluyen decisiones de implementación sobre una plataforma tecnológica específica. Para facilitar su representación, se han creado dos metamodelos diferentes, uno para los conceptos relacionados con la lógica (metamodelo de negocio) y otro con los conceptos de la interfaz de usuario (metamodelo de interfaz).

5.2.1 Metamodelo de Negocio

El dominio del negocio es una extensión del metamodelo del mundo original, no obstante, en el metamodelo de negocio se introduce el concepto de servicio y todas las características asociadas al mismo.

En la figura 5 se muestra el metamodelo de negocio de la arquitectura, en donde se observan varios elementos del metamodelo del mundo como son *Agrupador*, *Simple*, *Elemento*, *Atributo* y *AtributoCupi2*. Además, en este metamodelo se introducen nuevos conceptos como son: *Clase*, *Servicio*, *Parámetro* y *DatoEspecial*. El concepto *Clase* se utiliza para generalizar a

todos los *Elementos* y relacionarlos con sus *Servicios*, sus *Atributos* y con el punto de entrada del metamodelo. Los servicios son incluidos puesto que son necesarios para manipular las estructuras de datos presentes en los ejemplos.

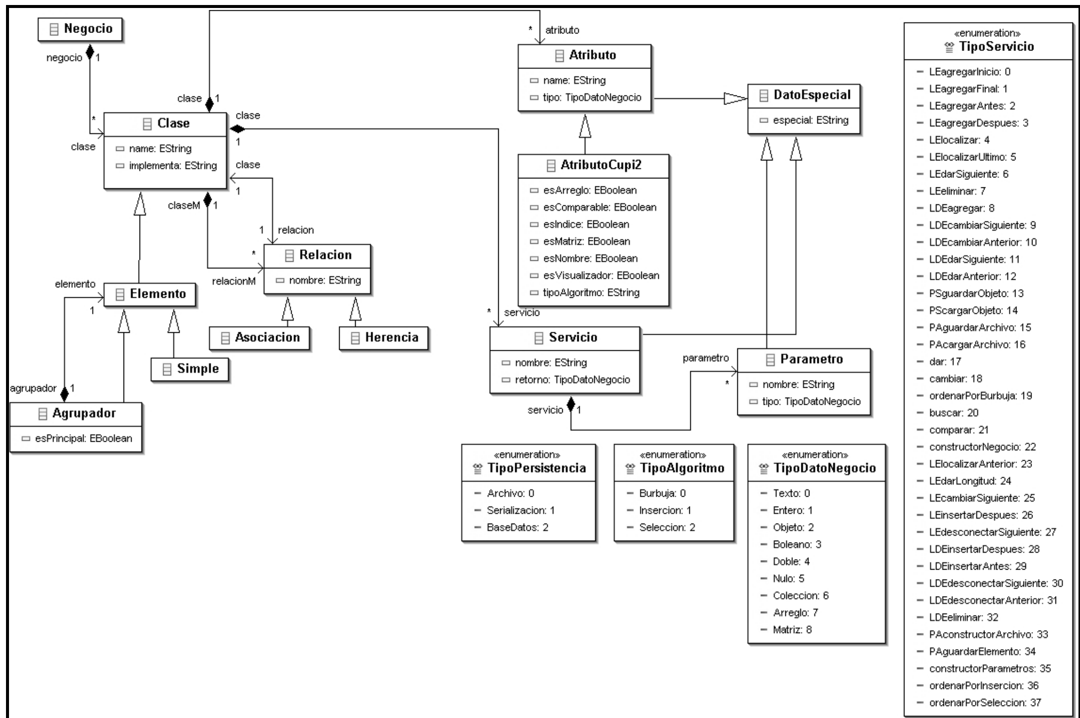


Figura 5 - Metamodelo de Negocio (Arquitectura)

5.2.2 Metamodelo de Interfaz

La interfaz, contempla los conceptos de elementos gráficos y de interacción que serán parte de la interfaz gráfica de usuario del producto generado. En la figura 6 se muestra el metamodelo de interfaz. Este metamodelo busca la generalización de los conceptos encontrados en los distintos tipos de vistas identificadas para los ejemplos de Cupi2. Como elemento principal se tiene el concepto de *Vista*, una vista a la vez pude tener un conjunto de *componentes* y un conjunto de *servicios*. Los componentes se dividen en dos tipos, componentes de *interacción* y de *visualización*. Los componentes de interacción se refieren a componentes visuales mediante los cuales el usuario puede realizar invocar algún tipo de funcionalidad. Algunos ejemplos de componentes de interacción pueden ser botones, listas o listas desplegables. Por otro lado, los componentes de visualización se usan para mostrar la información propia del ejemplo al usuario. Algunos ejemplos de estos componentes son las etiquetas los cuadros de texto o los mensajes.

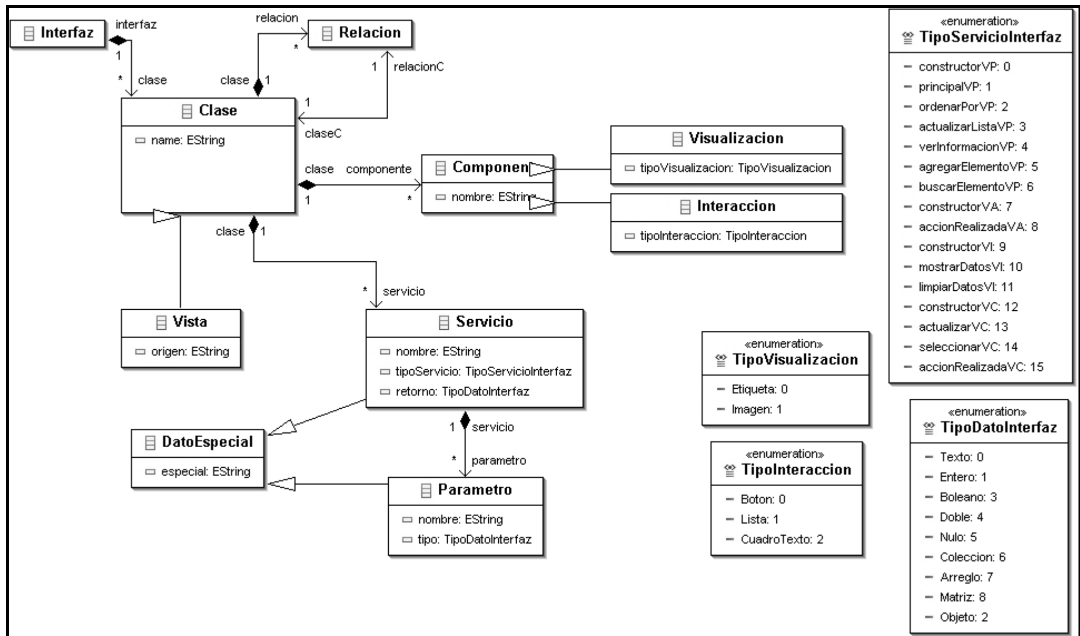


Figura 6 - Metamodelo de Interfaz de usuario (Arquitectura)

5.2.3 Creación de modelos conformes al metamodelo de arquitectura

Los modelos conformes a la arquitectura se dividen en dos tipos. Los conformes al metamodelo de negocio y los conformes al metamodelo de interfaz. De acuerdo con el esquema para el desarrollo de productos mostrado en el capítulo 4, estos modelos se generan a partir de la ejecución de las transformaciones. Sin embargo, con el fin de ilustrar la forma en que se pueden usar los conceptos de los metamodelos de arquitectura para expresar los ejemplos, se presentan a continuación los modelos del ejemplo de la exposición de automóviles visto en la sección previa. De esta manera, se pueden ver los cambios en los modelos cuando se enriquecen con los conceptos del negocio y la interfaz de usuario en el dominio de la arquitectura.

5.2.3.1 Modelo conforme al metamodelo de Negocio (Arquitectura)

En la figura 7 se muestra el modelo del ejemplo de Exposición de Automóviles conforme al metamodelo de negocio. De la misma manera que en la figura 4, se utilizan estereotipos para mostrar la relación de conformidad de los elementos del modelo con el metamodelo. Los elementos presentes en este modelo son los mismos del modelo del mundo. Es decir, se tiene una exposición de automóviles que agrupa a un conjunto de automóviles. Exposición es conforme a *Agrupador* y Automóvil es conforme a *Simple*. Los datos de los elementos conformes a *Atributo* y *AtributoCupi2* también se mantienen. Sin embargo, los nuevos elementos se representan con la aparición de los servicios. En este ejemplo particular, se tienen dos atributos comparables, estos son, modelo y precio. Cada uno de estos atributos conlleva a la aparición de los servicios ordenarPorModelo y ordenarPorPrecio en la exposición y a compararPorModelo y compararPorPrecio en el Automóvil.

Además, se crea un servicio de búsqueda basado en el atributo nombre del Automóvil. Por otro lado, se crean los servicios asociados con la relación de agrupación entre exposición y automóvil como son agregarAutomovil, buscarAutomovil, cambiarAutomóvil y eliminarAutomovil. Por último se agregan métodos necesarios para el funcionamiento de la aplicación como son los métodos para obtener y cambiar los atributos, y los constructores de las clases.

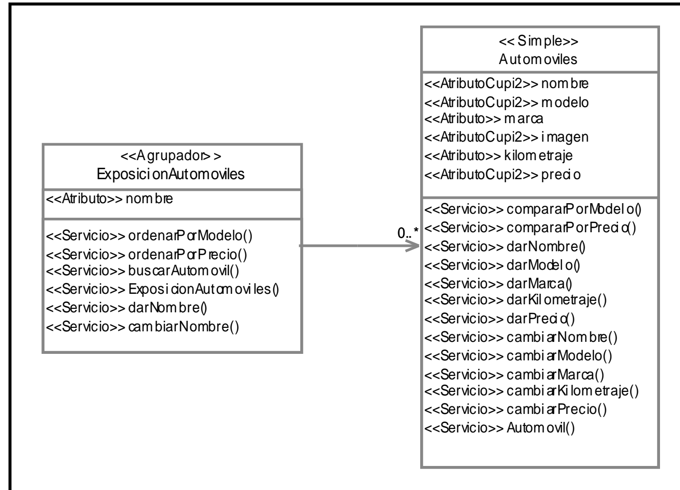


Figura 7 - Modelo conforme al metamodelo de Negocio (Arquitectura)

5.2.3.2 Modelo conforme al metamodelo de Interfaz (Arquitectura)

La figura 8 presenta un modelo conforme al metamodelo de interfaz de la arquitectura. En este caso se tienen cuatro clases diferentes conformes al elemento Vista del metamodelo. Cada vista a su vez tiene un conjunto de componentes específicos de acuerdo a sus responsabilidades. El elemento *InterfazExposicion* es la vista principal y contiene a las demás vistas. La vista *ConjuntoAutomoviles* se encarga de mostrar al grupo de automóviles contenidos por la estructura de datos del ejemplo. Esta vista tiene una componente de interacción de tipo lista para mostrar el conjunto de automóviles. Los botones *ordenarPorModelo y ordenarPorPrecio* están asociados con los servicios de ordenamiento del negocio sobre la lista de automóviles. La vista *BusquedaAutomóvil* brinda al usuario la posibilidad de buscar un automóvil en específico a partir del nombre. Por último, la vista *InformaciónAutómovil*, tiene un conjunto de etiquetas para visualizar cada uno de los atributos del Automóvil.

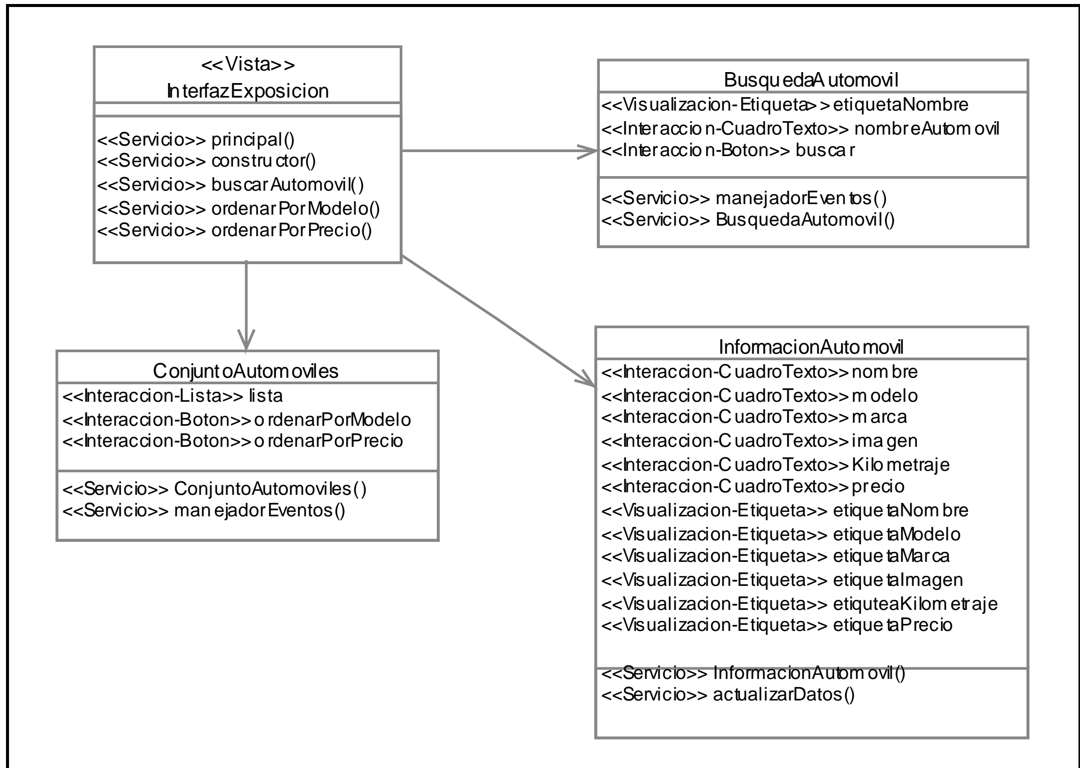


Figura 8 - Modelo para el ejemplo Exposición de Automóviles conforme al metamodelo de Interfaz (Arquitectura).

5.3 Metamodelo de Tecnología (Java)

El último dominio definido es para la tecnología, en este dominio se involucran todos los detalles propios del lenguaje de implementación de las aplicaciones que en nuestro caso particular es Java.

El metamodelo de Java (Figura 9) representa los conceptos de éste lenguaje de programación. El punto de entrada corresponde a la metaclase *JavaApp*. Esta meta-clase agrupa a todos los paquetes (*Package*) que forman parte de la aplicación. A su vez, cada paquete agrupa a las clases (*JavaClass*). Cada clase se compone de atributos (*Field*) y métodos (*Method*). Cada método en el metamodelo de Java tiene un conjunto de parámetros (*Parameter*) y un cuerpo (*Body*). Este cuerpo tiene un identificador, el cuál se utiliza para relacionar cada método del modelo con la plantilla especial correspondiente al cuerpo del método, lo que permite la generación del código funcional.
Adicionalmente, este metamodelo contiene un conjunto de especializaciones relacionadas directamente con el dominio de Cupi2. Las clases del metamodelo de Java se especializan en clases de negocio (*Business*) y clases de interfaz (*Interface*). A su vez las clases de negocio se especializan en agrupadoras (*Container*) y simples (*Simple*). Los atributos también se especializan en atributos de cupi2 (*Cupi2Field*).

Con esta integración entre los conceptos de los ejemplos Cupi2 y los conceptos de la plataforma tecnológica, se busca que los modelos conformes a este metamodelo tengan no sólo los detalles de la implementación, sino que también tengan toda la semántica propia de los dominios del mundo y la arquitectura.

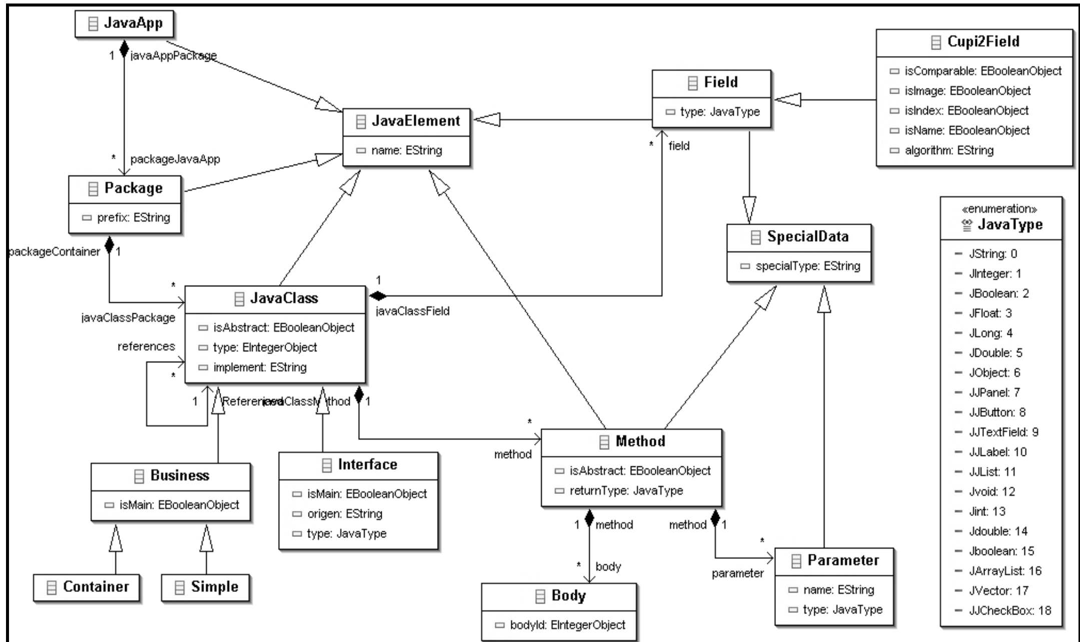


Figura 9 - Metamodelo de Java

5.3.1 Modelo conforme al metamodelo de Java

Los modelos conformes al metamodelo de java, tienen todos los elementos de negocio e interfaz de usuario que describen al sistema que representan. Además, estos modelos utilizan los tipos de datos y las características del lenguaje de programación Java.

En la figura 10 se muestra el modelo del componente mundo para la exposición de automóviles conforme al metamodelo de java. Existe un punto de entrada o raíz que representa a la aplicación. Dentro de la raíz, se agrupan los paquetes. En este caso existe un paquete para el negocio. Aquí se encuentran las clases correspondientes al modelo de negocio en la arquitectura. Cada clase se conforma de métodos y campos de acuerdo a los servicios y atributos existentes previamente en el modelo de negocio pero especificados con las características propias de la plataforma tecnológica

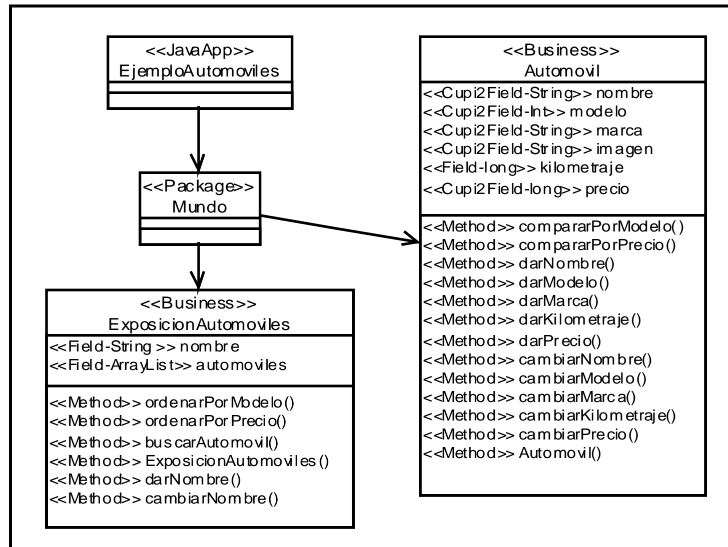


Figura 10 - Modelo del componente Mundo conforme al metamodelo de Java

Para una mejor comprensión de los metamodelos mostrados en este capítulo, el Anexo A presenta una descripción detallada de cada una de las meta clases, meta atributos, relaciones y enumeraciones de los metamodelos de mundo, negocio, interfaz de usuario y tecnología.

6 Transformaciones de la MD-SPL de Cupi2

Como se explicó en el capítulo 4, mediante el uso de transformaciones, los modelos se enriquecen con las características propias de cada dominio. Así, las transformaciones son las herramientas mediante las cuales se convierte un modelo de un dominio particular a otro.

En nuestra línea de productos utilizamos transformaciones para enriquecer un modelo del mundo del problema inicial, de forma secuencial a través de los dominios definidos. Dado que contamos con tres dominios diferentes (mundo, arquitectura y plataforma tecnológica), son necesarias dos transformaciones. La primera transformación va del dominio del mundo al dominio de la arquitectura. La segunda transformación que va del dominio de la arquitectura al dominio de la plataforma. Adicionalmente es necesaria una última transformación que tome un modelo del dominio de la tecnología y genere el código fuente funcional. Como se mostró en el capítulo anterior, el dominio de la arquitectura se divide en dos metamodelos, el de negocio y el de interfaz de usuario. Por esta razón, se tienen dos transformaciones independientes entre estos dominios. En total se necesitan cuatro transformaciones como se ilustra en la figura 11.

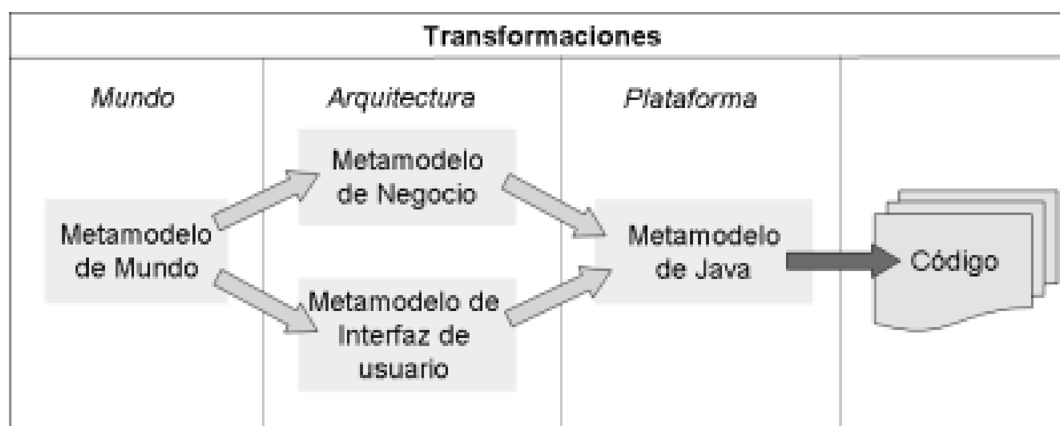


Figura 11 - Transformaciones entre los dominios de Cupi2

De las cuatro transformaciones mostradas en la figura 10, tres son transformaciones de tipo modelo a modelo y la restante es de tipo modelo a código. Para el primer tipo de transformaciones se utilizó el lenguaje ATL, que permite la transformación de modelos a partir de los conceptos de sus metamodelos. Para la transformación de la regla modelo a texto, se utilizó Acceleo, un lenguaje diseñado para este tipo de transformaciones que a través de la integración de los modelos con plantillas, permite la generación del código fuente de las aplicaciones. El capítulo 7 muestra una reseña más detallada de cada uno de los lenguajes utilizados, así como también de las demás tecnologías necesarias para la construcción de la línea de productos.

6.1 Transformaciones de Tipo Modelo a Modelo

Las transformaciones entre modelos son unidireccionales. A partir de modelos origen, se crea un modelo destino. Durante la ejecución de la transformación los modelos origen pueden ser navegados, sin embargo, no se permite la modificación de su contenido. Adicionalmente, el modelo destino que está siendo creado, no puede ser navegado. La definición de una transformación en ATL se conoce como módulo. Un módulo contiene un encabezado, una sección para referenciar otras reglas (*import section*) y un conjunto de *helpers* y reglas de transformación [15].

El encabezado tiene el nombre del módulo y especifica los nombres de los metamodelos y modelos de origen y destino para la transformación. En la figura 12 se muestra el encabezado de la transformación que va del metamodelo del mundo al dominio de la arquitectura, específicamente al metamodelo de negocio. El primer dato corresponde al nombre de la transformación. Luego se especifican los nombres del modelo de salida, en este caso el negocio y los modelos de entrada que en este caso corresponden al modelo del mundo y al modelo de entrelazado.

```
module mundo2Negocio;
create OUT : negocio from IN2 :weaving, IN : mundo;
```

Figura 12 - Encabezado de las transforamaciones ATL

Los *helpers* se usan para realizar operaciones sobre los modelos, y pueden ser invocados desde diferentes puntos en un módulo ATL. Pueden verse como los equivalentes en ATL a los métodos de Java. Sin embargo, estos *helpers* no se pueden utilizar para generar elementos en el modelo destino, por lo cual su uso se limita a realizar procesos intermedios que ayuden a establecer valores de inicialización dentro de las reglas de transformación. En la figura 13 se muestra el *helper hay Serializacion*. Este *helper* se utiliza para identificar si sobre algún elemento del mundo se deben implementar los servicios de serialización para lograr la persistencia.

```
helper def: haySerializacion(elementoMundo: mundo!Elemento) : Boolean=
        weaving!Link.allInstances()->asSequence()->iterate(e; ret:Boolean= false |
                if e.right.name.startsWith('Serializacion') then
                        if elementoMundo.name.startsWith(e.leftname) then
                                true
                        else
                                ret
                        endif
                else
                        ret
                endif
        );
```

Figura 13 - Ejemplo de un *helper* en ATL

Las reglas de transformación son el componente esencial de ATL y se usan para expresar la lógica de la transformación. En nuestro caso, usamos tres tipos de reglas: reglas de base, reglas de control y reglas especificas.

6.1.1 Reglas de correspondencia

Las reglas base son declarativas y especifican para que elementos del origen se crean elementos en el destino, y la forma en que los elementos del destino deben ser inicializados. Con una regla base se establece un tipo dado de elemento en el modelo de origen y se generan uno o más tipos de elementos en el modelo destino. El motor de ATL recorre los modelos origen y si encuentra una regla que cumple con las características de correspondencia del modelo, la ejecuta. Estas reglas no especifican un orden de navegación del modelo origen o de ejecución.

En la figura 14 se ilustra una regla base que convierte todos los atributos del mundo en atributos del negocio. En la primera instrucción se observa el patrón de origen. En el patrón de origen se especifican los elementos del modelo de origen que van a ser transformados, en este caso, los únicos elementos manejados por esta regla son los que sean conformes al elemento *Atributo* del metamodelo del mundo. El patrón destino describe los elementos que deben ser creados en el modelo destino a partir de la información del elemento origen. En este caso el nombre y el tipo del atributo del modelo origen se asignan a los elementos correspondientes del modelo destino

```
rule Atributo
{
        from atributo          : mundo!Atributo(atributo.oclIsTypeOf(mundo!Atributo))
        to   atributoNegocio   : negocio!Atributo(
                                        nombre<-atributo.nombre,
                                        tipo<-atributo.tipo
                                       )
}
```

Figura 14 - Regla Base

6.1.2 Reglas de Control

Este tipo de reglas también son similares a las reglas base pero tienen un componente imperativo de control. Por ejemplo, en la figura 15 se muestra la misma regla usada en la figura 14 pero en este caso se añaden instrucciones imperativas. Esto significa que una vez el motor de ejecución encuentre las correspondencias que describe esta regla, no sólo crea los elementos especificados en el patrón de destino, sino que además ejecuta las instrucciones imperativas. Estas instrucciones pueden ser usadas para modificar los elementos de salida o para invocar otras reglas, modificando de esta manera el orden de ejecución de la transformación. En este caso particular, la instrucción imperativa invoca a una regla que se encarga de crear los servicios cambiar y dar (getters y setters) para cada atributo.

```
rule Atributo
{
        from atributo        : mundo!Atributo(atributo.oclIsTypeOf(mundo!Atributo))
        to   atributoNegocio : negocio!Atributo(
                                    nombre<-atributo.nombre,
                                    tipo<-atributo.tipo
                                   )
        do
        {
                thisModule.crearSetterGetter(atributoNegocio);
        }
}
```

Figura 15 - Regla de Control

6.1.3 Reglas específicas

Las reglas específicas brindan mecanismos imperativos para solucionar problemas específicos. Son reglas que sólo se ejecutan cuando son invocadas y pueden recibir parámetros. Estas reglas a diferencia de los *helpers*, pueden generar elementos en el modelo destino.

En la figura 16 se muestra una regla específica. Esta regla se encarga de adicionar los métodos dar y crear para cada atributo transformado en la regla de la figura 15. Adicionalmente, la sentencia imperativa se usa para relacionar los nuevos servicios creados con el atributo que se recibe como parámetro.

```
rule crearSetterGetter(atributo: Negocio!Atributo)
{
        to
        getter    : negocio!Servicio(nombre <- 'dar'+atributo.nombre, retorno<-atributo.tipo ,
                                    tipoServicio<-#dar),
        setter    : negocio!Servicio(nombre <- 'cambiar'+atributo.nombre, retorno<-#Nulo,
                                    tipoServicio<-#cambiar)
        do
        {
                atributo.clase.servicio<- atributo.clase.servicio->union(Sequence{getter,setter});
        }
}
```

Figura 16 - Regla específica

6.1.4 Expresión de variabilidad a través de las reglas de transformación

En nuestra línea de productos, se integra un esquema para el manejo de la variabilidad de cada uno de los miembros de la familia que se quieren construir. Este trabajo se realizó en conjunto con Kelly Garcés [12].

La variabilidad se expresa a través de modelos de rasgos. Cada rasgo representa una característica que puede tener o no una aplicación generada por la línea de productos. A su vez los rasgos se relacionan con reglas imperativas que adicionan elementos al modelo que se está transformando. Así

una selección de rasgos implica la ejecución de un conjunto de reglas que a su vez generan nuevos elementos en el modelo destino.
Para lograr este objetivo, se crea un modelo de entrelazado. El modelo de entrelazado es el resultado de la selección de un conjunto de rasgos sobre un modelo origen.

Cada rasgo esta relacionado con una o varias reglas de transformación que se encargan de construir los elementos necesarios en el modelo, que satisfacen el rasgo.
En [12] se describen en detalle los rasgos de la línea de productos y el proceso de entrelazado y la relación con las reglas de transformación.

6.2 Transformaciones Modelo a Modelo para la línea de Cupi2

Como se ilustró en la figura 11, nuestra línea utiliza tres transformaciones de tipo modelo a modelo. A su vez cada transformación tiene reglas generales y específicas para el manejo de la variabilidad. A Continuación se explican los conceptos más importantes de cada una de ellas. Adicionalmente en el Anexo B, se encuentra el código ATL en donde se pueden observar los detalles técnicos de implementación de cada uno de los conceptos que se mencionan en este capítulo.

6.2.1 Transformación Mundo a Negocio

Esta transformación toma los elementos del mundo y los convierte en elementos de la arquitectura. Además, teniendo en cuenta las características de las agrupaciones del mundo y de los rasgos seleccionados, crea los servicios necesarios para manipular las agrupaciones y persistir la información del mundo.

6.2.1.1 Regla Entrada

Esta regla se usa para corresponder los puntos de entrada de los modelos. En este caso, para cada elemento conforme a la metaclase *ElementoCupi2* del metamodelo del mundo, se crea un elemento conforme a la metaclase *Negocio* del metamodelo de negocio en la arquitectura.

6.2.1.2 Regla Agrupador

La regla Agrupador transforma a todos los elementos que sean conformes a la metaclase *Agrupador* del metamodelo de mundo. Para cada elemento agrupador se crea un elemento en la arquitectura con los mismos datos. Además, la regla Agrupador verifica si el agrupador que se está transformando tiene asociado alguno de los rasgos de persistencia. Si es el caso, invoca a otra regla de transformación encargada de generar los servicios propios de este rasgo.

6.2.1.3 Regla Simple

La regla simple transforma todos los elementos del modelo del mundo que sean conformes a la metaclase Simple, mediante de la creación de otro elemento conforme a la metaclase Simple del metamodelo de negocio en la arquitectura.

6.2.1.4 Regla Atributo

La regla atributo se encarga de transformar los elementos del modelo origen que sean conformes a la metaclase *Atributo*. Los atributos de estos elementos como son nombre y tipo, se asignan al nuevo elemento del mismo tipo en el modelo de negocio. Además, mediante sentencias imperativas, se invoca a una regla especial, encargada de crear los métodos dar y cambiar (getter y setter). Esto se hace para todos los atributos encontrados dentro de los distintos elementos del metamodelo de origen.

6.2.1.5 Regla AtributoCupi2

Los atributos de tipo Cupi2 del mundo, se replican con la misma información a través de la metaclase *AtributoCupi2* del metamodelo de negocio. Además esta transformación crea los servicios necesarios a partir de la información de los atributos. Por ejemplo, si el atributo es comparable, se invoca una regla que genera los servicios de ordenamiento y comparación del atributo.

6.2.1.6 Regla AsociacionElemento

La regla *AsociacionElemento*, es la regla más compleja de todas las reglas que hacen parte de la transformación mundo a negocio. En esta regla se transforman las asociaciones entre elementos.
De acuerdo con la descripción del metamodelo, las relaciones de asociación se usan para describir estructuras de datos de los tipos listas y listas doblemente encadenadas. Así lo primero que se hace es verificar si la lista es simple o doble. Esto se logra a través del atributo *esDoble* de la metaclase Asociación. Si la relación es simple, se invoca a la regla *crerMetodosListas*, en donde se crean los servicios y atributos necesarios para la manipulación de los elementos dentro de una estructura de datos de tipo lista. Por otro lado si la lista es doble, se invoca a la regla *crearMetodosListasDobles*, en donde se crean los servicios y los atributos propios de una lista doblemente encadenada.

6.2.1.7 Regla PersistenciaArchivo2Negocio

Esta regla es específica, y su función es crear los servicios para persistir la información del elemento a través de archivos. Los servicios creados son guardar y cargar.

6.2.1.8 Regla PersistenciaSerializacion2Negocio

Esta regla es específica, y su función es crear los servicios para persistir la información del elemento a través de la serialización de objetos. Aquí se crean los servicios guardar y cargar, pero además se crea una relación de herencia. Esto se hace para garantizar que los elementos que se quieran persistir sean serializables, es decir, que sus atributos cumplan con las condiciones

necesarias que impone la tecnología particular para que se puedan almacenar en archivos que representan a los objetos.

6.2.1.9 Regla crearMetodosListas

Esta regla se invoca para crear los servicios para la manipulación de listas doblemente encadenadas. Los servicios creados a través de esta regla son: *agregar*, *agregarAlFinal*, *agregarAntesDe*, *agregarDespuesDe*, *localizar*, *localizarUltimo*, *eliminar* y *darSiguiente*. Además de los servicios antes mencionados también se crea el atributo siguiente. Tanto los servicios como el atributo creado, se asocian a los elementos sobre los que se crea la asociación que representa la lista encadenada.

6.2.1.10 Regla crearMetodosListasDobles

Esta regla es similar a la anterior, pero en este caso se crean los servicios y atributos que se requieren para las listas doblemente encadenadas, estos son: *agregar*, *agregarAntesDe*, *agregarDespuesDe*, *localizar*, *darSiguiente* y *darAnterior*. Además de los servicios se crean los atributos siguiente y anterior. Todos los servicios y atributos creados, se asocian a los elementos sobre los que se crea la asociación que representa la lista doblemente encadenada.

6.2.1.11 Regla crearServiciosComparables

Esta regla recibe un atributo de Cupi2 que tiene la propiedad de ser comparable. Así, esta regla crea los servicios *ordenarPor* y *compararPor* y los asocia con el elemento que contenga al atributo.

6.2.1.12 Regla crearSetterGetter

De la misma forma que la regla anterior, esta regla crea los servicios dar y cambiar para el atributo que recibe por parámetro y los asocia a la clase del elemento que contiene al atributo.

6.2.2 Transformación Mundo a Interfaz

Esta transformación crea un modelo conforme al metamodelo de interfaz a partir del mundo y de los rasgos seleccionados por el usuario.

6.2.2.1 Regla Entrada

Esta regla base se usa para corresponder los puntos de entrada de los modelos del mundo y la interfaz. En este caso, para cada elemento conforme a la metaclase *ElementoCupi2* del metamodelo del mundo, se crea un elemento conforme a la metaclase Interfaz del metamodelo de Interfaz en la arquitectura.

6.2.2.2 Regla agregarVistaInterfaz

La regla de control *agregarVistaInterfaz* transforma los elementos del modelo de entrelazado hagan referencia a las vistas de la interfaz. Para cada correspondencia, crea un elemento conforme a Vista del metamodelo de interfaz. Dependiendo del tipo de interfaz, se agregan componentes de interacción o visualización que sean necesarios.

6.2.2.3 Regla agregarComponenteVisualizacionVista

Esta es una regla especifica que agrega un componente de visualización a una vista. Los datos del componente que se crea, así como de la vista sobre la que se debe agregar, se reciben como parámetros de entrada de la transformación.

6.2.2.4 Regla agregarComponenteInteraccionVista

Esta regla cumple con una función similar a la anterior, pero en este caso los componentes que se crean y se agregan a las vistas son de interacción.

6.2.2.5 Regla agregarConstructor

La regla *agregarConstructor*, crea y agrega el servicio constructor a la vista que se recibe como parámetro de entrada a la regla. El constructor es el servicio que inicializa y ordena todos los elementos que hacen parte de la interfaz.

6.2.2.6 Regla agregarManejador

Cuando las vistas tienen componentes de interacción, estos generan eventos producidos por las acciones de los usuarios. El servicio *manejador* captura los eventos y realiza las acciones necesarias para cada caso. Esta regla se encarga crear y agregar este tipo de servicios a las vistas que lo necesiten.

6.2.3 Transformación Arquitectura a Java

La transformación de arquitectura a Java tiene tres modelos origen. El modelo de negocio, el modelo de la interfaz, y el modelo de rasgos de tecnología. A partir de estos tres modelos, se crea un único modelo que expresa todos los conceptos de la aplicación en términos de los elementos y tipos de datos propios de Java. A continuación se describen las diferentes reglas de esta transformación.

6.2.3.1 Regla Entrada

Esta regla base se usa para crear el punto de entrada y los paquetes de la aplicación a partir de las entradas del modelo de interfaz y del modelo de negocio. En términos de los metamodelos, se crea un elemento conforme a la metaclase *JavaApp* y dentro de éste, se crean dos elementos conformes a la metaclase *Package*, uno para la interfaz y otro para el negocio.

6.2.3.2 Regla Agrupador2Clase

Esta regla base se encarga de crear clases a partir de los elementos agrupadores que existan en el modelo de origen. Todas las clases creadas a partir de esta regla se agregan al paquete del mundo.

6.2.3.3 Regla Simple2Clase

Esta regla base se encarga de crear clases a partir de los elementos de tipo Simple que existan en el modelo de origen. Todas las clases creadas a partir de esta regla se agregan al paquete del mundo.

6.2.3.4 Regla Vista2Clase

Esta regla de control se encarga de crear clases a partir de los elementos de tipo Vista que existan en el modelo de origen. Todas las clases creadas a partir de esta regla se agregan al paquete de interfaz. Además basada en los rasgos de tecnología decide el tipo de implementación que tendrá cada vista.

6.2.3.5 Reglas Servicio2Method y Regla ServicioInterfaz2Method

Estas reglas base convierten los servicios existentes en el modelo de origen, en métodos del modelo de Java.

6.2.3.6 Reglas AtributoNegocio2Field y AtributoCupi2Negocio2Field

Estas reglas de control crean campos en las clases de java, para todos los elementos de tipo *Atributo* y *AtributoCupi2* encontrados en el modelo origen. Los tipo de datos seleccionados en Java se seleccionan a partir de los rasgos de tecnología que escoja el usuario.

6.2.3.7 Reglas Interaccion2Field y Visualizacion2Field

Estas reglas crean campos en las clases de java, para todos los elementos de tipo *Interacción* y *Visualización* encontrados en el modelo origen. Los tipos de datos propios de java como cajas de texto, listas, etiquetas, etc., se seleccionan a partir de los rasgos de tecnología que escoja el usuario.

6.3 Transformación Modelo a Código Fuente

Una vez se ha creado un modelo que representa a la aplicación y los conceptos se expresan en términos de la tecnología particular, el último paso es generar el código fuente. Para esto se utiliza la transformación de modelo a código. En está transformación a partir de la información del modelo de la aplicación, y de un conjunto de plantillas de código, se crean los artefactos que integran el producto final. Para nuestros productos, los artefactos generados corresponden a paquetes, clases, atributos y métodos. Los paquetes son directorios que agrupan un conjunto de clases con características similares. En el caso de los ejemplos de nuestra línea, siempre existen dos paquetes uno para el mundo del problema y otro para la interfaz de usuario. La información de estos paquetes se obtiene del modelo de la aplicación. Las clases junto con sus atributos y sus métodos se crean a partir de las clases existentes en el modelo. Sin embargo, la información que contiene el modelo sobre los métodos se refiere sólo a su signatura. En este punto se utilizan las plantillas de cuerpos de los métodos. Cada método ha sido previamente clasificado de acuerdo a su funcionalidad, de esta manera se tienen plantillas para cada tipo de método. En la figura 17, se muestra la plantilla para el método encargado de cargar una colección a partir de un archivo. Las instrucciones resaltadas acceden al

modelo. Con la información obtenida del modelo, se completa la plantilla y se crean métodos completos y funcionales dentro de la clase a la que pertenecen.

```
ruta = param;
try
{
    BufferedReader br = new BufferedReader( new FileReader( param ) );
    String linea = "";
    int numeroLinea = 0;
    linea = br.readLine( ).trim();
    int numero = 0;

    while( linea.equals( "" ) )
    {
        linea = br.readLine( ).trim();
        numeroLinea++;
    }
    numero = Integer.parseInt( linea );
    for( int i = 0; i < numero; i++ )
    {
        numeroLinea++;
        <%.javaClassMethod.name%> p = new <%.javaClassMethod.name%>( br);
        //Verificar que tipo de estructura y la forma de agregacion
        <%for (javaClassMethod.field){%>
        <%if (.type.equalsIgnoreCase("JArrayList")){%>
            <%.name%>.add( p );
        <%}%>
    }
    br.close();
}
catch (FileNotFoundException e) {
      e.printStackTrace();
}
catch (IOException e) {
      e.printStackTrace();
}
```

Figura 17 - Plantilla para método que carga una estructura de datos

7 Implementación

A medida que crece la aceptación de enfoques como el propuesto con MDA, se espera que la utilización de lenguajes para transformación de modelos, se convierta en una tarea tan común como puede ser hoy en día la programación de computadores en lenguajes como Java o C++. Para soportar estos nuevos procesos de desarrollo, se han desarrollado un conjunto de lenguajes y herramientas para trabajar en la ingeniería basada en modelos. A continuación se presenta una breve reseña de algunos de estos lenguajes y herramientas que fueron estudiados durante la creación de la línea de productos de software basada en modelos para el proyecto Cupi2

7.1 Entorno de desarrollo Eclipse

Como ambiente de desarrollo se utilizó Eclipse [10]. Eclipse ofrece una plataforma de desarrollo abierta. Una de las características principales de Eclipse es la utilización de módulos (plugins). Esto permite extender la funcionalidad del entorno para incluir soporte a diversas tecnologías de acuerdo a las necesidades de cada usuario particular. En nuestro caso, hemos utilizado un conjunto de herramientas para la creación de metamodelos y modelos, creación y ejecución de transformaciones y entrelazado de modelos.

7.2 Creación de Metamodelos EMF y Omondo

EMF (Eclipse Modelling Framework) [11], es un marco de trabajo para la generación de herramientas y aplicaciones basado en modelos de clases.
Gracias a EMF, se pueden crear metamodelos que definen para expresar los conceptos de los dominios de las aplicaciones. EMF ofrece una interfaz de tipo árbol, mediante la cual se crean las metaclases, meta-atributos y relaciones de los metamodelos. Sin embargo, la creación de metamodelos y en especial de las relaciones entre meta-clases utilizando esta interfaz, es complicada. Por esta razón, se utilizó Omondo [17]. Omondo se integra a Eclipse y permite la creación de metamodelos mediante una interfaz gráfica, similar a la ofrecida por las herramientas de diseño UML tradicionales.
En la figura 18, se muestra la interfaz de Omondo para la creación de metamodelos de manera gráfica.

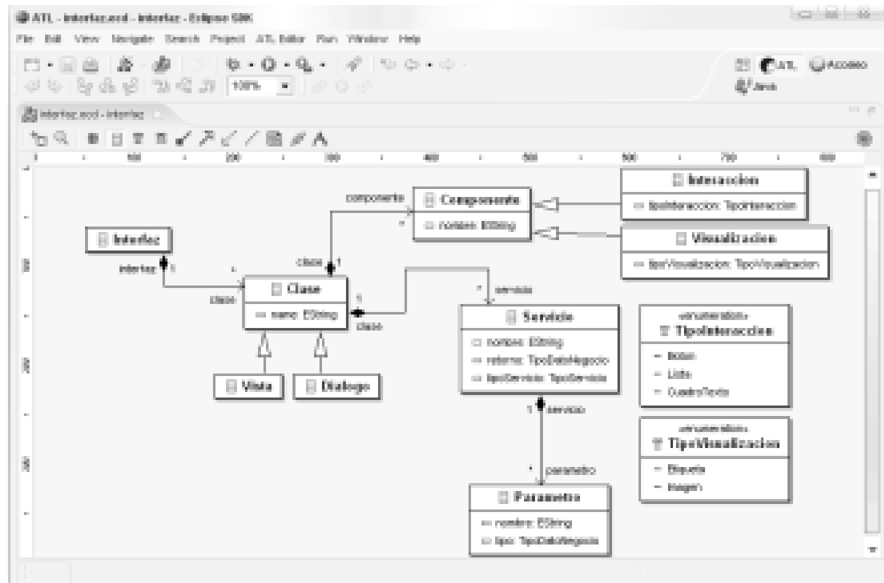


Figura 18 - Interfaz de creación de metamodelos de Omondo

7.3 Lenguajes de Transformación

7.3.1 ATL

ATL(ATLAS Transformation Language) es un lenguaje con el que se pueden realizar transformaciones dentro de un framework MDA. ATL se desarrolló como parte de la plataforma AMMA(ATLAS Model Management Architecture). [3]

Es un lenguaje hibrido, es decir, sus instrucciones pueden ser declarativas o imperativas de acuerdo a las necesidades de los desarrolladores.

ATL puede ser usado en un esquema de transformación como el mostrado en la figura 19, En este esquema, un modelo origen conforme a un metamodelo origen se transforma en un modelo destino conforme a un metamodelo destino mediante la transformación ATL. Los metamodelos a su vez son conformes a MOF.

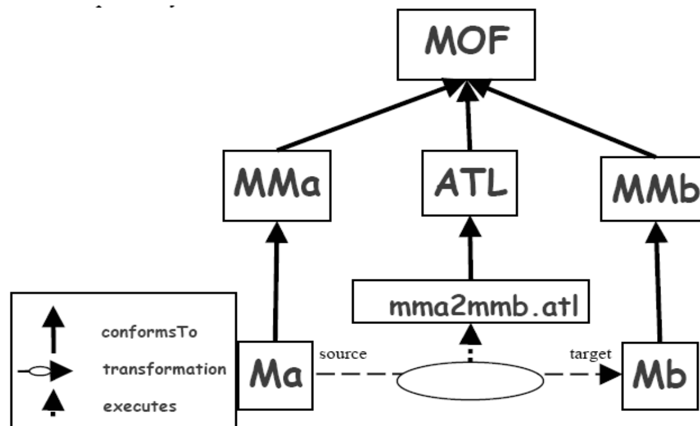


Figura 19 - Esquema de utilización de ATL [15]

ATL también se integra a Eclipse a través de un plugin denominado ADT (ATL Development Tools). Este plugin integra un conjunto de herramientas como son, motor de ejecución de reglas, un editor de reglas con sintaxis resaltada, y otras funcionalidades como un depurador y un generador de configuraciones de ejecución. En la figura 20 se muestra el entorno de desarrollo Eclipse utilizando el plugin de ATL para el desarrollo de reglas

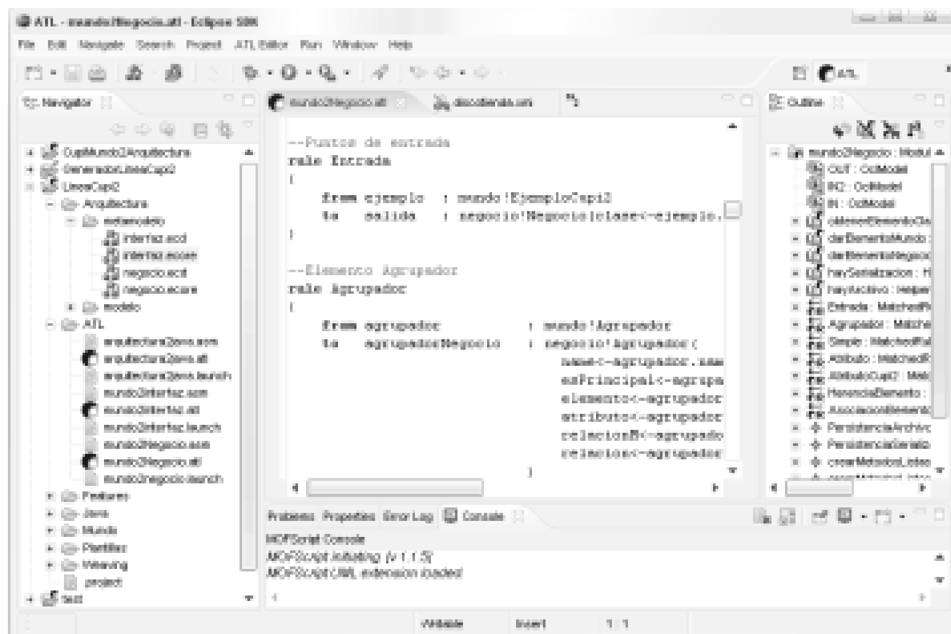


Figura 20 - Integración de ATL y Eclipse

7.3.2 Acceleo

Acceleo [1] es una herramienta de generación de código fuente, basada en Eclipse y EMF. Para la generación, se crean módulos que representan un conjunto de plantillas que describen la información requerida para generar código fuente en términos de los metamodelos. Las plantillas se integran con la información de los modelos que se navegan con base en la estructura de su metamodelo. Además las plantillas pueden tener embebidos fragmentos de código definidos por el usuario, esto significa que los usuarios pueden modificar el código generado y volver a generar la aplicación sin que su código sea modificado por el proceso de generación. En la figura 21 se muestra el entorno de desarrollo que ofrece Acceleo gracias a su integración con Eclipse

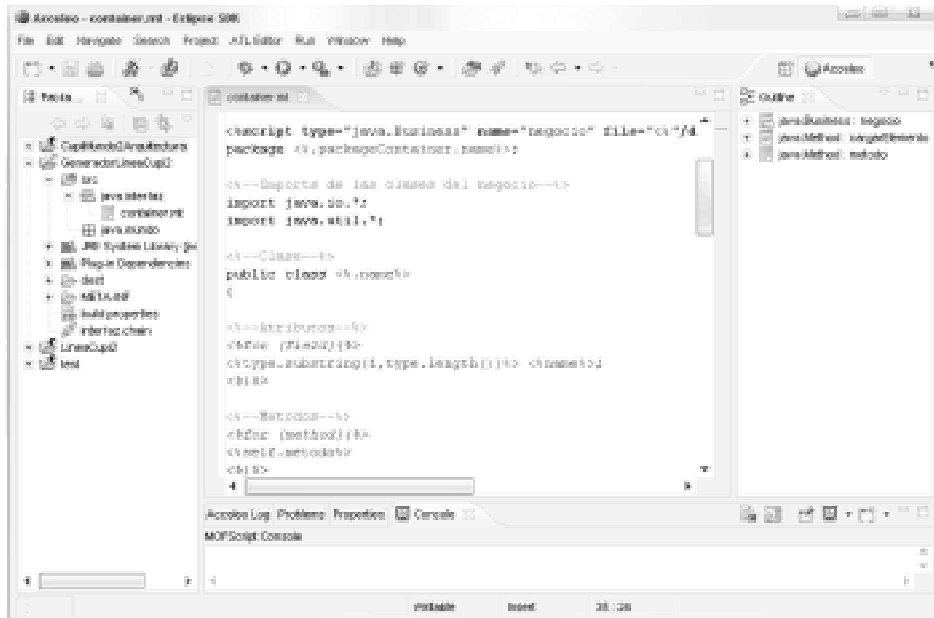


Figura 21 - Acceleo y Eclipse

7.4 Entrelazado de Modelos AMW

AMW (Atlas Model Weaver) [2] es una herramienta para establecer relaciones entre modelos. La idea principal de la implementación es tener una interfaz de usuario simple que pueda adaptarse al metamodelo de entrelazado escogido por el usuario. La interfaz por defecto se ajusta al metamodelo de entrelazado base y tiene tres paneles: izquierdo, derecho y central. En el panel izquierdo se visualiza el modelo origen, en el panel derecho el modelo destino y en el panel central el modelo de entrelazado. El modelo de entrelazado se construye adicionando enlaces y arrastrando elementos de los modelos origen y destino que sean correspondientes. La figura 22 muestra el entorno de creación de los modelos de entrelazado a través AMW.

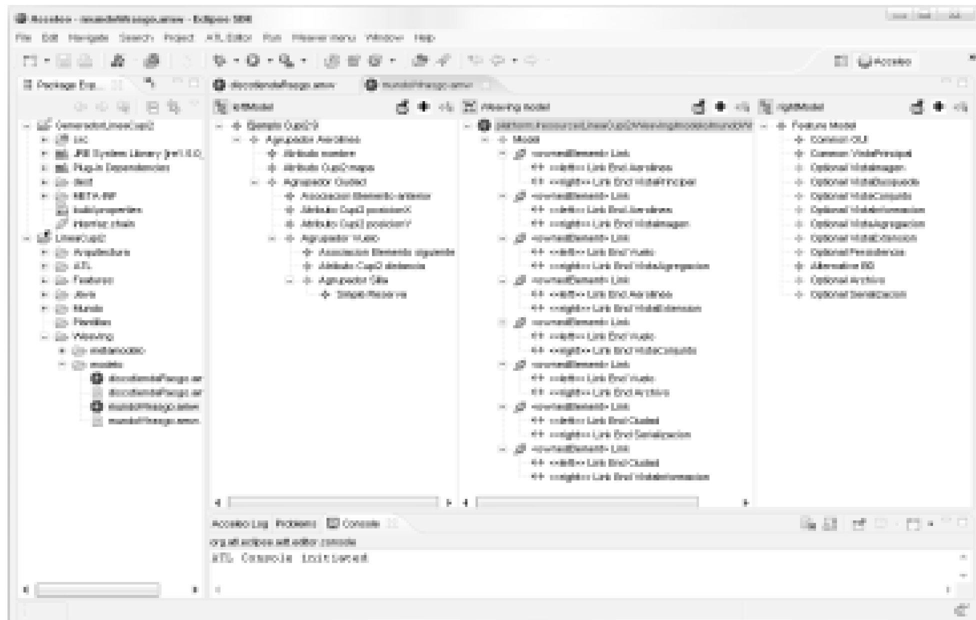


Figura 22 - Entrelazado a través de AMW

8 Conclusiones y aportes de la Investigación

La industria del software, presenta nuevos retos relacionados con las expectativas de los usuarios y la implementación de nuevas tecnologías. Automatizar los procesos de desarrollo es una necesidad que se hace más explicita con el paso del tiempo.

Las estrategias propuestas en MDA y SPL son ambiciosas y prometen facilitar estos procesos a través de esquemas generativos y composicionales. Sin embargo, las implementaciones exitosas en casos del mundo real son todavía limitadas. Esto principalmente porque este tipo de estrategias implican cambios en los esquemas tradicionales de desarrollo de software, dando una gran importancia a las abstracciones de alto nivel de los sistemas a través de modelos y a la definición de estrategias de composición de productos a través de artefactos que incrementen la productividad y el tiempo de desarrollo.

Esta investigación ha intentado dar un paso adelante en este dominio, mediante la construcción de un sistema que se basa en los principios establecidos tanto por MDA como por las SPL.

Se han identificado y descrito tres dominios que describen las características más importantes de aplicaciones de baja complejidad, a través de los metamodelos. Además se ha llegado a la especificación de las transformaciones necesarias para generar productos funcionales. La utilización de metamodelos para cada dominio particular del proceso de desarrollo, provee un esquema de separación de preocupaciones, en donde el esfuerzo inicial se focaliza en las reglas del negocio, para luego especificar los conceptos de arquitectura y plataforma. Utilizando transformaciones, los modelos se enriquecen en cada etapa del proceso de generación de la línea, hasta llegar a un modelo que representa un producto de software. De esta forma, sólo se construye manualmente el modelo inicial de la lógica del negocio usando conceptos con un alto nivel de abstracción. Posteriormente se generan de forma automática otros modelos por medio de reglas de transformación hasta obtener aplicaciones completas. Así, se construyen y usan reglas de trasformación más sencillas y flexibles que las necesarias para transformar en un solo paso un modelo de lógica de negocio en una aplicación completa.

Además, esta investigación va más allá, introduciendo el manejo de la variabilidad de una familia de productos. Este es tal vez uno de los temas más interesantes para la construcción de esquemas de línea de producto que cumplan efectivamente con las expectativas de los usuarios.

Se ha mostrado, que mediante una especificación de los rasgos comunes y variables de un grupo de aplicaciones, es posible integrar al usuario en un proceso de personalización del producto. Las decisiones del usuario se reflejan mediante modelos de entrelazado que relacionan aspectos particulares del

mundo del problema, con características especificas de los productos de la línea.

Finalmente, los modelos de entrelazado que representan las preferencias de los usuarios, se acoplan al proceso de generación a través de las reglas de transformación que por un lado, crean los elementos comunes a todos los productos y por otro, crean los elementos específicos a cada rasgo particular seleccionado por el usuario. De esta manera, es imaginable la construcción de productos que comparten características en común, pero que están diferenciados para satisfacer las necesidades de cada usuario.

8.1 Aportes de la investigación

8.1.1 Integración de MDA y SPL

El aporte principal de este trabajo se presenta a través de la combinación de los esquemas composicional y generativo. Para esto se han creado los metamodelos, modelos y transformaciones necesarios para la generación de código fuente de aplicaciones de baja complejidad. Adicionalmente, se han creado activos basados en plantillas de código que se componen a través de los modelos dando como resultado el código fuente de las aplicaciones.

8.1.2 Definición de los metamodelos

Los metamodelos permiten plasmar los conceptos de un conjunto de aplicaciones a través de la abstracción de sus características principales. En nuestro caso, se definieron tres dominios como son: mundo, arquitectura y plataforma tecnológica. Para cada dominio se crearon los metamodelos que fueron validados mediante la construcción de modelos conformes y transformaciones.

8.1.3 Definición de las transformaciones

Se crearon las transformaciones necesarias para transformar un modelo con los conceptos del mundo en un modelo con todas las características de arquitectura y plataforma tecnológica. Además, cada transformación incluyó un conjunto de reglas de distintos tipos: base, de control y específicas. Estos tipos de reglas permitieron la integración de un esquema de variabilidad basado en rasgos, mediante el cual, los usuarios de la línea hacen parte de un proceso de personalización de las aplicaciones generadas.

8.1.4 Creación de activos basados en plantillas

Las plantillas de código se crearon a partir de los métodos comunes encontrados en los distintos tipos de ejemplos de la familia de productos. Cada plantilla de método tiene un conjunto de vacíos que se deben llenar con la información de los modelos de la aplicación que se esta generando. Las plantillas garantizan que el código generado es consistente con los estándares.

Además su integración con los modelos supone un alto grado de reutilización a lo largo de los distintos productos.

8.1.5 Automatización del proceso de construcción de productos

La construcción de productos, ha sido automatizada mediante la creación de un plugin que se encarga de mostrar una interfaz amigable a los usuarios. A la vez, este plugin se encarga de invocar cada proceso necesario en la construcción del producto, desde la creación del modelo origen hasta las transformaciones finales, pasando por el proceso de entrelazado de rasgos y las transformaciones intermedias. El proceso detallado de utilización del plugin se describe en [12].

9 Trabajos Futuros

Aún cuando los resultados hasta ahora obtenidos como parte de la implementación de nuestra propuesta son motivadores, existen un conjunto de áreas en las que se debe profundizar el trabajo.

Por ejemplo, la creación de metamodelos sigue siendo una tarea compleja, no hay ninguna metodología o parámetros que indiquen si un metamodelo es correcto o no con respecto al dominio que representa. Nuestras vías de verificación de los metamodelos se han basado en la construcción de transformaciones o en la creación de modelos conformes. Además, los metamodelos se enfocan en la estructura del dominio que representan. La expresión de comportamiento a través de los metamodelos no es del todo clara. La mayoría de herramientas actuales no ofrecen un marco de trabajo en el cual se pueda hacer una definición de las acciones de las posibles meta-operaciones. Trabajos más profundos en esta dirección podrían dar soluciones a los problemas relacionados con la expresión y transformación de conceptos relativos a requerimientos funcionales de las aplicaciones que se construyen.

Es necesario profundizar en el tema de separación de preocupaciones, explorando alternativas para separar dominios y para modelar conceptos de cada dominio particular. También es necesario trabajar sobre la construcción de reglas de control y reglas específicas más modulares con base en patrones de transformación mejor definidos. Como parte complementaria a los procesos de transformación, y en general a los procesos de generación de aplicaciones, en esta propuesta la administración de la trazabilidad es un campo completo por explorar.

A continuación se presentan algunos de los posibles trabajos que se podrían desarrollar sobre la MD-SPL de Cupi2.

9.1 Implementación de un esquema de trazabilidad

La MD-SPL Cupi2 genera ejemplos alineados con los temas de los módulos de Cupi2, sin embargo sería deseable la generación de los ejercicios. Un ejercicio es un proyecto completo de programación, en el cual hay partes que el estudiante debe completar y partes que están dadas. Las partes a completar están relacionadas con los temas que el ejercicio busca desarrollar. La construcción y el mantenimiento manual de los ejercicios son propensos a defectos. Por ejemplo cuando un ejercicio se está editando se pueden agregar elementos innecesarios u omitir otros esenciales para los temas del ejercicio. Es importante entonces controlar la propagación de los cambios de un modelo sobre los demás modelos, esto se relaciona con el manejo de la trazabilidad.

9.2 Extender la capacidad de la línea de productos

Para el desarrollo de este trabajo, hemos limitado el número de artefactos generados. Sin embargo, existen elementos de los ejemplos de Cupi2, ausentes en nuestra línea pero de gran importancia para la productividad del proyecto Cupi2 en general. Algunos de estos elementos son:

9.2.1 Componente de Pruebas

Las pruebas son el tercer componente de los ejemplos, junto con la interfaz y el mundo. Para incluir este tipo de artefactos, es necesario definir un nuevo metamodelo de pruebas a nivel de la arquitectura y crear las transformaciones necesarias que generen las clases y métodos necesarios de este componente.

9.2.2 Elementos gráficos

Las interfaces gráficas de los ejemplos de Cupi2 se basan en un conjunto de componentes como botones, etiquetas, cuadros de texto etc. De esta manera, agregar nuevos componentes como tablas, botones especiales, indicadores, etc., enriquecería las interfaces provistas por la línea y se ampliaría el conjunto de ejemplos posibles a través de la selección de los usuarios. No obstante, cada componente nuevo tiene un comportamiento especial y un conjunto de servicios que deben ser tenidos en cuenta en la construcción de los activos y en la integración con las interfaces actuales.

9.2.3 Características de otros niveles

Otros aportes que incrementaría la utilidad de nuestra línea podrían estar relacionados con la extensión de los metamodelos y transformaciones para incluir temas de diferentes niveles del proyecto Cupi2 como por ejemplo: bases de datos, gráficos en dos dimensiones, estructuras de datos de tipo árbol, etc.

9.3 Crear dominios para otros lenguajes

Esta investigación se ha centrado en un lenguaje de programación: Java. Esto debido a que es el lenguaje utilizado en el ámbito del proyecto Cupi2, para los cursos de programación de computadores. Sin embargo, es imaginable construir este tipo de aplicaciones en otros lenguajes de programación. Para esto sería necesario construir un nuevo metamodelo de plataforma tecnológica y crear las transformaciones necesarias. Como marco de referencia se pueden tomar los metamodelos y transformaciones propuestos en [6] y [23], en donde además de Java, se describen artefactos MDA para la construcción de aplicaciones empresariales utilizando la plataforma .NET.

9.4 Implementar soluciones para problemas reales con características similares

En nuestro proyecto hemos mostrado una estrategia mediante la cual creamos los artefactos propios de MDA como son los metamodelos, modelos de rasgos y transformaciones, y los utilizamos como activos de una línea de productos variables. Nuestra familia de productos objetivo son un conjunto de

aplicaciones de baja complejidad con un número importante de elementos comunes y elementos variables. Basados en los resultados obtenidos, pensamos que es posible construir sistemas similares en otros ámbitos en los que las condiciones de características comunes variables provean un marco de trabajo amigable y una oportunidad de aplicar un esquema integrado de líneas de productos de software basadas en modelos.

10 Bibliografía

[1]. Acceleo. En Línea: http://www.acceleo.org, Ultima visita 2006

[2]. AMW Home Page. En Línea: http://www.eclipse.org/gmt/amw/, Ultima visita 2006

[3]. ATL Home Page. En Línea: http://www.eclipse.org/gmt/atl/, Ultima visita 2006

[4]. A. Belangour, J. Bézivin, M. Fredj. Towards platform independence : a MDA organization. Workshop on Information Technology. Rabat, Morocco, 2003.

[5]. J. Bézivin On the Unification Power of Models. *Software and Systems Modeling, 4* (2). 171-188. 2005.

[6]. J. Bohórquez. Framework De MDA para aplicaciones empresariales hacia plataformas J2EE Y .Net - Metamodelos. Grupo de Construcción de Software: Universidad de los Andes, 2006.

[7]. K. Czarnecki, S. Helsen. (2003). Calssification of Model Transformation Approches. Canadá: University of Waterloo.

[8]. M. Didonet Del Fabro, J. Bézivin, F. Jouault, E. Breton y G. Gueltas, AMW: a generic model weaver. en *Journées sur l'Ingénierie Dirigée par les Modèles (IDM'05)*, Paris, France, 2005.

[9]. M. Didonet Del Fabro, F. Jouault, K. Van den Berg, Model Transformation and Weaving in the AMMA Platform.

[10]. Eclipse Home. En Línea: http://www.eclipse.org/ , Ultima visita 2006

[11]. EMF Home. Disponible en: http://www.eclipse.org/emf/ - Ultima visita Enero 2007.

[12]. K. Garcés. Expresión de Variabilidad en una línea de productos basada en modelos. Grupo de Construcción de Software: Universidad de los Andes, 2007

[13]. A. Kleppe, J. Warmer, W. Bast. MDA Explained: The Model Driven Architecture: Practice and Promise, Addisson Wesley (2003)

[14]. I. Kurtev. Adaptability of Model Transformations. University of Twente, 2005.

[15]. I. Kurtev, F. Jouault. Transforming Models with ATL. ATLAS Group(INRIA & LINA) University of Nantes.

[16]. J. Miller, J. Mukerji. OMG. MDA Guide Versión (2003)

[17]. Omondo. Disponible en: http://www.omondo.com/ - Ultima visita Enero 2007

[18]. Proyecto Cupi2. Disponible en http://cupi2.uniandes.edu.co. Ultima visita Diciembre 2005

[19]. S.E.I. A Framework for Software Product Line Practice, Version 4.2, Software Engineering Institute En linea: http://www.sei.cmu.edu/productlines/framework.html

[20]. The Triskell Metamodeling Language Kermeta. En línea: http://www.kermeta.org/ Ultima visita Diciembre 2006

[21]. J. Villalobos, R. Casallas, K. Marcos. El reto de diseñar un primer curso de programación de computadores. Departamento de ingeniería de Sistemas y Computación. Universidad de los Andes.

[22]. J. Withey. Investment Analysis of Software Assets for Product Lines, CMU Technical Report (1996)

[23]. A. Yie. Framework De MDA para aplicaciones empresariales hacia plataformas J2EE Y .Net - Transformaciones. Grupo de Construcción de Software: Universidad de los Andes, 2006.

11 Anexo A - Metamodelos de la MD-SPL Cupi2

A continuación se presenta una descripción detallada de cada una de las metaclases presentes en los metamodelos de la MD - SPL Cupi2

11.1 Metamodelo de Mundo

Nombre	**EjemploCupi2**
Tipo	MetaClase
Descripción	Punto de entrada del metamodelo del mundo del problema. Su función principal es agrupar a todos los elementos que puedan existir en el mundo.
Atributos	• *nivel*: Información del nivel específico de Cupi2, al cual pertenece el Ejemplo
Clases Asociadas	Elemento: El EjemploCupi2 contiene a todos los elementos que hacen parte del mundo del problema.
Restricciones	Debe ser único para cualquier modelo y debe contener todos los elementos existentes.

Nombre	**Elemento**
Nombre MetaClase	MetaClase
Descripción	Representa un concepto del mundo del problema que agrupa un conjunto de atributos. Por ejemplo, si se modela una Aerolínea, ejemplos de Elementos serían Aerolínea, Ciudad, Vuelo, Etc..
Atributos	*Nombre:* Información del nombre del elemento
Clases Asociadas	*Atributo*: Un Elemento puede tener uno o varios Atributos, que caracterizan aún más al elemento *Agrupador*: Un Elemento puede especializarse en Elemento de tipo Agrupador, es decir, que contiene a otros elementos. Simple: Un elemento puede especializarse en Elemento de tipo Simple, es decir que no contiene a ningún elemento
Restricciones	Ninguna

Nombre	**Agrupador**
Tipo	MetaClase
Descripción	Representa a un Elemento, que además puede contener otros elementos.
Atributos	*esPrincipal:* Indica si el elemento está o no contenido por

	otro elemento agrupador, en el segundo caso, se considera como elemento principal pues contiene a los demás elementos que hacen parte del mundo del problema.
Clases Asociadas	Elemento: Un elemento agrupador, puede contener uno o más e Elementos.
Restricciones	Si su atributo esPrincipal es falso, debe estar contenido en otro Agrupador

Nombre	**Simple**
Tipo	MetaClase
Descripción	Representa a un elemento que no contiene a otros elementos
Atributos	
Clases Asociadas	
Restricciones	Un elemento simple siempre debe estar contenido por el elemento EjemploCupi2 o por el elemento Agrupador.

Nombre	**Atributo**
Tipo	MetaClase
Descripción	Representa una característica propia de un elemento
Atributos	• *Nombre*: Nombre del Atributo • *Tipo:* Tipo de dato del atributo, está limitado a los tipos encontrados en la Enumeración TipoDatos.
Clases Asociadas	Elemento: Un atributo hace parte de un Elemento.
Restricciones	Ninguna

Nombre	**AtributoCupi2**
Tipo	MetaClase
Descripción	Especialización de la metaclase Atributo, mediante la cual se incluye nueva información específica de los Ejemplos de Cupi2, por ejemplo si el atributo es usado como índice en algún tipo de búsqueda.
Atributos	• *esComparable*: Indica si el atributo es usado en algoritmos que utilicen las funciones de comparación y ordenamiento de Elementos. • *esIndice*: Indica si el atributo es usado en algoritmos de búsqueda. • *esNombre*: Indica si el atributo hace parte del nombre del Elemento. • *esVisualizador*: Indica si el atributo está asociado a una imagen que representa al Elemento.
Clases Asociadas	Elemento: Un atributo hace parte de un Elemento.

Restricciones	Ninguna

Nombre	**RelacionElemento**
Tipo	MetaClase
Descripción	Esta metaclase se usa para expresar una relación entre dos elementos, distinta a la relación de agrupación entre el Agrupador y el Elemento
Atributos	
Clases Asociadas	Elemento: La relación se establece entre dos Elementos
Restricciones	Ninguna

Nombre	**AsociacionElemento**
Tipo	MetaClase
Descripción	Especialización de la metaclase Relación, mediante la cual se representa la relación necesaria para crear listas encadenadas y doblemente encadenadas.
Atributos	• *esDoble*: Indica si la lista es doblemente encadenada o sencilla
Clases Asociadas	Elemento: La relación se establece entre dos Elementos
Restricciones	Ninguna

11.2 Metamodelo de Arquitectura Negocio

Nombre	**Negocio**
Tipo	MetaClase
Descripción	Punto de entrada del metamodelo de Arquitectura (Negocio). Su función principal es agrupar a todos los elementos que puedan existir en el negocio
Atributos	
Clases Asociadas	Clase: Un Negocio contiene a las clases que hacen parte de la arquitectura.
Restricciones	Debe ser único para cualquier modelo y debe contener a todos los elementos del negocio.

Nombre	**Clase**
Tipo	MetaClase
Descripción	Metaclase usada para agrupar a todos los tipos de Elementos presentes en el mundo del problema de un ejemplo de Cupi2.
Atributos	• *name*: Nombre de la Clase
Clases Asociadas	*Servicio*: Asociación que permite relacionar una clase con sus servicios ofrecidos. *Atributo*: Asociación que permite relacionar una clase con sus atributos. *Relacion*: Asociación que permite a una clase o una especialización de la clase contener a otras clases.
Restricciones	Ninguna

Nombre	**Relación**
Tipo	MetaClase
Descripción	Metaclase utilizada para expresar una relación entre clases
Atributos	
Clases Asociadas	*Clase*: Una Relación está asociada a una clase *ClaseM*: La clase contiene una o muchas relaciones, que a su vez se asocian con una clase.
Restricciones	Ninguna

Nombre	**Asociación**
Tipo	MetaClase
Descripción	Metaclase que extiende *Relacion* y se utiliza para expresar una relación de especialización entre dos elementos.
Atributos	
Clases Asociadas	*Clase*: Una Relación está asociada a una clase *ClaseM*: La clase contiene una o muchas relaciones, que a su vez se asocian con una clase.
Restricciones	Ninguna

Nombre MetaClase	**Elemento**
Descripción	Extiende a la a metaclase Clase y representa un concepto del mundo del problema que agrupa un conjunto de atributos y servicios. Por ejemplo, si se modela una Aerolínea, ejemplos de Elementos serían

	Aerolínea, Ciudad, Vuelo, etc.
Atributos	Name: Nombre del elemento
Clases Asociadas	*Atributo*: Un Elemento puede tener uno o varios Atributos, que caracterizan aún más al elemento *Agrupador*: Un Elemento puede especializarse en Elemento de tipo Agrupador, es decir, que contiene a otros elementos. Simple: Un elemento puede especializarse en Elemento de tipo Simple, es decir que no contiene a ningún elemento
Restricciones	Ninguna

Nombre	**Agrupador**
Tipo	MetaClase
Descripción	Representa a un Elemento, que además puede contener otros elementos.
Atributos	• *esPrincipal:* Indica si el elemento está o no contenido por otro elemento agrupador, en el segundo caso, se considera como elemento principal.
Clases Asociadas	Elemento: Un elemento agrupador, puede contener uno o más Elementos.
Restricciones	Si su atributo esPrincipal es falso, debe estar contenido en otro Agrupador

Nombre	**Simple**
Tipo	MetaClase
Descripción	Representa a un elemento que no contiene a otros elementos
Atributos	
Clases Asociadas	
Restricciones	Un elemento simple siempre debe estar contenido por el Negocio o por elemento Agrupador.

Nombre	**Atributo**
Tipo	MetaClase
Descripción	Representa una característica propia de un elemento
Atributos	• *nombre:* Nombre del Atributo • *tipo:* Tipo de dato del atributo, está limitado a los tipos encontrados en la Enumeración TipoBasico
Clases Asociadas	Elemento: Un atributo hace parte de un Elemento.
Restricciones	Ninguna

Nombre MetaClase	**AtributoCupi2**

Descripción	Especialización de la metaclase Atributo, mediante la cual se incluye nueva información específica de los Ejemplos de Cupi2, por ejemplo si el atributo es usado como índice en algún tipo de búsqueda.
Atributos	• *esComparable*: Indica si el atributo es usado en algoritmos que utilicen las funciones de comparación y ordenamiento de Elementos. • *esIndice*: Indica si el atributo es usado en algoritmos de búsqueda. • *esNombre*: Indica si el atributo hace parte del nombre del Elemento. • *esVisualizador*: Indica si el atributo está asociado a una imagen que representa al Elemento.
Clases Asociadas	Elemento: Un atributo hace parte de un Elemento.
Restricciones	Ninguna

Nombre	**Servicio**
Tipo	MetaClase
Descripción	Representa un servicio prestado por la clase a la que pertenece.
Atributos	• *nombre:* Nombre del Atributo • *retorno:* Tipo de dato de la salida de una operación, está limitado a los tipos encontrados en la Enumeración TipoDato. • *tipoServicio:* Asocia al servicio de acuerdo a su funcionalidad con uno de los servicios encontrados en la enumeración TipoServicio
Clases Asociadas	Clase: Un servicio hace parte de una clase Parámetro: Un Servicio tiene un conjunto de parámetros
Restricciones	Ninguna

Nombre	**Parámetro**
Tipo	MetaClase
Descripción	Representa a cada uno de los datos de entrada de una Operación.
Atributos	• *nombre:* Nombre del parámetro • *tipo:* Tipo de dato del parámetro, está limitado a los tipos encontrados en la Enumeración TipoDato
Clases Asociadas	Operación: Un parámetro hace parte de una Operación
Restricciones	Ninguna

Nombre MetaClase	**DatoEspecial**
Descripción	Se utiliza para manejar los tipos de datos propios del

	ejemplo. Por ejemplo cuando el retorno de un servicio corresponde al tipo de dato de un elemento del mundo.
Atributos	• *nombre:* Nombre del dato especial que debe corresponder con el nombre de algún elemento del mundo.
Clases Asociadas	Servicio: El servicio puede tener un retorno te tipo especial. Atributo: El atributo de una clase puede ser de tipo especial. Parámetro: El parámetro de un servicio puede ser de tipo especial.
Restricciones	Ninguna

Nombre	**TipoServicio**
Tipo	Enumeración
Descripción	Se utiliza para limitar el número de posibles servicios presentes en al arquitectura. Abarca métodos relacionados con estructuras de datos, persistencia y comportamiento de las vistas de la interfaz.
Atributos	
Clases Asociadas	Servicio: El tipo de servicio esta limitado al grupo listado en esta enumeración.
Restricciones	Ninguna

Nombre	**TipoDato**
Tipo	Enumeración
Descripción	Se utiliza para limitar los tipos de datos que se pueden usar con los atributos y parámetros de los modelos del mundo y la arquitectura.
Atributos	
Clases Asociadas	Servicio: El retorno de un servicio esta limitado a la lista de tipos de la enumeración TipoDato Parámetro: El tipo de dato del Parametro de un servicio esta limitado a la lista de tipos de la enumeración TipoDato Atributo: El tipo de dato de un Atributo esta limitado a la lista de tipos de la enumeración TipoDato
Restricciones	Ninguna

11.3 Metamodelo de Arquitectura (Interfaz)

Nombre	**Interfaz**
Tipo	MetaClase
Descripción	Punto de entrada del metamodelo de Arquitectura (Interfaz). Su función principal es agrupar a todos los elementos que puedan existir en la interfaz.
Atributos	
Clases Asociadas	Clase: La interfaz contiene a las clases que hacen parte de la interfaz.
Restricciones	Debe ser único para cualquier modelo y debe contener todos los elementos existentes.

Nombre	**Clase**
Tipo	MetaClase
Descripción	Metaclase usada para agrupar a todos los tipos de Vistas presentes en la interfaz de un ejemplo de Cupi2.
Atributos	• *nombre:* Nombre de la Clase
Clases Asociadas	*Operacion*: Asociación que permite relacionar una clase con sus servicios ofrecidos. *Atributo*: Asociación que permite relacionar una clase con sus atributos. *Relacion*: Asociación que permite a una clase o una especialización de la clase contener a otras clases.
Restricciones	Ninguna

Nombre	**Vista**
Tipo	MetaClase
Descripción	Metaclase que representa los diferentes tipos de interfaces posibles para los ejemplos de cupi2, se especializa en diferentes tipos de vistas con características específicas de acuerdo a la función de cada vista.
Atributos	
Clases Asociadas	*Clase*: Una vista es una especialización de una clase. *Relacion*: Asociación que permite a una clase o una especialización de la clase contener a otras clases.
Restricciones	Ninguna

Nombre	**Atributo**
Tipo	MetaClase

Descripción	Representa una característica propia de una Clase
Atributos	• *nombre:* Nombre del Atributo • *tipo:* Tipo de dato del atributo, está limitado a los tipos encontrados en la Enumeración TipoDato
Clases Asociadas	Clase: Un atributo hace parte de una Clase
Restricciones	Ninguna

Nombre	**Componente**
Tipo	MetaClase
Descripción	Un componente es un elemento de la interfaz, el cual hace parte de una Vista y tiene un objetivo particular, se puede especializar en componente de interacción o componente de visualización.
Atributos	Nombre: nombre del componente
Clases Asociadas	Atributo: El componente es una especialización de un atributo.
Restricciones	No debe ser contenido por una VistaPrincipal

Nombre	**Interacción**
Tipo	MetaClase
Descripción	Representa a un componente mediante el cual, se permite la interacción con el usuario final del ejemplo, de esta manera se disparan eventos que invocan la funcionalidad ofrecida por la aplicación.
Atributos	• *tipoInteraccion:* Tipo de elemento de interfaz del componente, está limitado a los tipos encontrados en la Enumeración TipoInteraccion.
Clases Asociadas	Componente: Interacción es una especialización de un Componente.
Restricciones	No debe ser contenido por una VistaPrincipal No debe ser contenido por una VistaInformacion

Nombre	**Visualización**
Tipo	MetaClase
Descripción	Representa a un componente mediante que no permite interacción con el usuario. Sólo se usa para mostrar información respectiva a algunos elementos del mundo del problema.
Atributos	• *tipoVisualizacion:* Tipo de elemento de visualización del componente, está limitado a los tipos encontrados en la Enumeración TipoVisualizacion.
Clases Asociadas	Componente: Visualización es una especialización de un Componente.
Restricciones	No debe ser contenido por una VistaPrincipal

Nombre	**Servicio**
Tipo	MetaClase
Descripción	Representa un servicio prestado por la clase a la que pertenece.
Atributos	• *nombre:* Nombre del Atributo • *retorno:* Tipo de dato de la salida de una operación, está limitado a los tipos encontrados en la Enumeración TipoDato
Clases Asociadas	Clase: Un servicio hace parte de una clase Parámetro: Una operación tiene un conjunto de parámetros
Restricciones	Ninguna

Nombre	**Parámetro**
Tipo	MetaClase
Descripción	Representa a cada uno de los datos de entrada de una Operación.
Atributos	• *nombre:* Nombre del parámetro • *tipo:* Tipo de dato del parámetro, está limitado a los tipos encontrados en la Enumeración TipoDato
Clases Asociadas	Operación: Un parámetro hace parte de una Operación
Restricciones	Ninguna

Nombre	**TipoInteracción**
Tipo	Enumeración
Descripción	Se utiliza para limitar los tipos de componentes de interaccion que se pueden utilizar en las vistas.
Atributos	
Clases Asociadas	*Interaccion*: El tipode un componente de interacción esta limitado a la lista de tipos de la enumeración TipoInteraccion
Restricciones	Ninguna

Nombre	**TipoVisualización**
Tipo	Enumeración
Descripción	Se utiliza para limitar los tipos de componentes de visualización que se pueden utilizar en las vistas.
Atributos	
Clases Asociadas	Visualización: El tipo de un componente de visualización esta limitado a la lista de tipos de la enumeración TipoVisualización
Restricciones	Ninguna

11.4 Metamodelo de Plataforma Tecnológica (Java)

Nombre	**JavaElement**
Tipo	Metaclase
Descripción	La metaclase JavaElement es una abstracción que representa a todos los posibles elementos de java. Se usa para incluir un nombre a todas las metaclases que la extienden.
Atributos	*name*: El nombre del elemento, todas las metaclases del metamodelo que sean un elemento de java tienen un nombre.
Clases Asociadas	JavaApp: La metaclase que representa la aplicación es un elemento de Java. JavaClass: La metaclase JavaClass es un elemento de java. Field: Los atributos son elementos de java. Method: Los métodos son elementos de java.
Restricciones	Ninguna

Nombre	**JavaApp**
Tipo	Metaclase
Descripción	Representa a una aplicación de java y es el punto de entrada del metamodelo.
Atributos	
Clases Asociadas	JavaElement: La aplicación es un elemento de java. Package: La aplicación contiene un grupo de paquetes.
Restricciones	Ninguna

Nombre	**Package**
Tipo	Metaclase
Descripción	El paquete de java representa una agrupación de clases de acuerdo a sus responsabilidades o características particulares.
Atributos	*prefix*: Un paquete puede tener un prefijo que se antepone al nombre.
Clases Asociadas	JavaElement: Los paquetes son elementos de java. JavaApp: El paqute debe estar contenido dentro de una aplicación de java. JavaClass: Cada paquete contiene clases de java.
Restricciones	Ninguna

Nombre	**JavaClass**
Tipo	Metaclase
Descripción	Es la metaclase que representa a las clases de Java

Atributos	*extends*: indica si la clase del ejemplo extiende a una clase propia de la tecnología. En nuestro caso es posible por ejemplo que la clase extienda Serializable para lograr guardar los objetos en archivos, o si extiende a JPanel puede ser visualizada e incluir componentes de interfaz de usuario.
Clases Asociadas	Package: Cada clase pertenece a un Paquete Method: Cada clase puede tener un conjunto de métodos. Field: Cada clase puede tener un conjunto de atributos. Business: La clase se puede especializar en una clase para el negocio. Interface: La clase se puede especializar en una clase para la interfaz. Modifier: Los modificadores de la clase agregan las características de visibilidad de java. JavaElement: La clase es un elemento de java y tiene un nombre.
Restricciones	Ninguna

Nombre	**Business**
Tipo	Metaclase
Descripción	Representa al conjunto de clases referentes al núcleo o negocio de la aplicación.
Atributos	
Clases Asociadas	JavaClass: Es una especialización de una clase de Java.
Restricciones	Ninguna

Nombre	**Container**
Tipo	Metaclase
Descripción	Representa al conjunto de clases referentes al núcleo o negocio de la aplicación que agrupan a otras clases.
Atributos	
Clases Asociadas	JavaClass: Es una especialización de una clase de negocio
Restricciones	Ninguna

Nombre	**Simple**
Tipo	Metaclase
Descripción	Representa al conjunto de clases referentes al núcleo o negocio de la aplicación, que son agrupadas por otras clases contenedoras.
Atributos	
Clases Asociadas	Business: Es una especialización de una clase de negocio.

Restricciones	Ninguna

Nombre	**Interface**
Tipo	Metaclase
Descripción	Representa al conjunto de clases referentes a la interfaz de usuario.
Atributos	
Clases Asociadas	JavaClass: Es una especialización de una clase de Java.
Restricciones	Ninguna

Nombre	**Field**
Tipo	Metaclase
Descripción	Es la metaclase que representa a los atributos de las clases en Java
Atributos	*type*: Tipo de dato del atributo, esta limitado a los tipos presentes en la enumeración JavaType *specialType:* Es usado cuando el tipo del atributo no propio de java sino que corresponde a una clase de la aplicación.
Clases Asociadas	JavaClass: Cada atributo hace parte de una clase de java Cupi2Field: Los atributos pueden ser especializados en atributos Cupi2. JavaElement: Los atributos son elementos de java y tienen un nombre.
Restricciones	Ninguna

Nombre	**Cupi2Field**
Tipo	Metaclase
Descripción	Esta metaclase extiende los atributos de java e incluye las características propias de los atributos de los ejemplos Cupi2.
Atributos	• *isComparable*: Indica si el atributo es usado en algoritmos que utilicen las funciones de comparación y ordenamiento de elementos. • *isIndex*: Indica si el atributo es usado en algoritmos de búsqueda. • *esName*: Indica si el atributo hace parte del nombre del Elemento. • *esDisplay*: Indica si el atributo está asociado a una imagen que representa al Elemento.
Clases Asociadas	Field: Los atributos de cupi2 son una especialización de los atributos de Java.
Restricciones	Ninguna

Nombre	**Method**

Tipo	Metaclase
Descripción	Es la metaclase que representa a los métodos de las clases en Java
Atributos	*returnType:* indica el tipo de retorno del método. Está limitado al conjunto de datos presentes en la enumeración JavaType.
Clases Asociadas	JavaClass: Un método pertenece a una clase de Java. Parameter: Un método puede tener un conjunto de parámetros. Body: Un método tiene un cuerpo asociado.
Restricciones	Ninguna

Nombre	**Body**
Tipo	Metaclase
Descripción	Es la metaclase que representa el cuerpo de un método.
Atributos	*bodyId:* Identificador del tipo de cuerpo que corresponde al método al que pertenece.
Clases Asociadas	Method: Un cuerpo pertenece a un método.
Restricciones	Ninguna

Nombre	**Parameter**
Tipo	Metaclase
Descripción	Es la metaclase que representa los parámetros que puede recibir un métod en Java.
Atributos	*name:* Nombre del parámetro. *type:* Tipo de dato del parámetro, esta limitado a los tipos presentes en la enumeración JavaType.
Clases Asociadas	Method: Un parámetro pertenece a un método.
Restricciones	Ninguna

Nombre	**JavaType**
Tipo	Enumeración
Descripción	En esta enumeración se listan todos los tipos de datos de la tecnología que pueden ser usados para construir los ejemplos.
Atributos	
Clases Asociadas	
Restricciones	Ninguna

12 Anexo B - Reglas de Transformación

A continuación se presentan las diferentes reglas de transformación en lenguaje ATL, creadas para la MD-SPL de Cupi2

12.1 Transformación Mundo A Negocio

```
--**************************************************--
--Transformacion Mundo a Negocio      --
--Carlos Andres Parra A.              --
--c-parra1@uniandes.edu.co            --
--MDSPL Cupi2 - Qualdev Software Group--
--Universidad de los Andes 2007        --
--**************************************************--
module mundo2Negocio;
create OUT : negocio from IN2 :weaving, IN : mundo;


--Encontrar el Elemento con base en la clase
helper def: obtenerElementoClase(clazz : negocio!Clase) : negocio!Elemento=
        negocio!Elemento.allInstances()->asSequence()->iterate(e; ret : negocio!Elemento = Set(negocio!Elemento) |
                if e.name.startsWith(clazz.name) then
                        e
                else
                        ret
                endif
        );

--Encontrar el elemento del mundo con base en el link
helper context weaving!Link def: darElementoMundo() :
        mundo!Elemento =
        mundo!Elemento.allInstances()->asSequence()->iterate(e; ret : mundo!Elemento = Set(mundo!Elemento)|
                if e.name.startsWith(self.left.name) then
                        e
                else
                  ret
                endif
        );

--Encontrar el elemento del negocio con base en el link
helper context weaving!Link def: darElementoNegocio() :
        negocio!Elemento=
        negocio!Elemento.allInstances()->asSequence()->iterate(e; ret : negocio!Elemento = Set(negocio!Elemento)|
                                if e.name.startsWith(self.left.name)
                                then
                                        e
                                else
                                  ret
                                endif
                                );

--Buscar si existe el Rasgo Persistencia-Serializacion
helper def: haySerializacion(elementoMundo : mundo!Elemento) : Boolean =
        weaving!Link.allInstances()->asSequence()->iterate(e; ret:Boolean = false |
                if  e.right.name.startsWith('Serializacion') then
                        if elementoMundo.name.startsWith( e.left.name) then
                                true
                        else
                                ret
                        endif
                else
                        ret
                endif
        );
```

```
--Busca si existe el rasgo Persistencia - Archivo
helper def: hayArchivo(elementoMundo : mundo!Elemento) : Boolean =
	weaving!Link.allInstances()->asSequence()->iterate(e; ret:Boolean = false |
		if e.right.name.startsWith('Archivo') then
			if elementoMundo.name.startsWith( e.left.name) then
				true
			else
				ret
			endif
		else
			ret
		endif
	);

--Retorna el tipo de algoritmo como texto
helper def : obtenerAlgoritmoTexto(tipoAlgoritmo : mundo!TipoAlgoritmosMundo): String =
	if tipoAlgoritmo = #Burbuja
	then 'Burbuja'
		else if tipoAlgoritmo = #Insercion
		then 'Insercion'
			else if tipoAlgoritmo = #Seleccion
			then 'Seleccion'
				else
					'Burbuja'
			endif
		endif
	endif;


helper def : obtenerServicioOrdenador(atributo : mundo!AtributoCupi2): negocio!TiposServicio=
	if atributo.tipoAlgoritmo = #Burbuja
	then #ordenarPorBurbuja
		else if atributo.tipoAlgoritmo = #Insercion
		then #ordenarPorInsercion
			else if atributo.tipoAlgoritmo = #Seleccion
			then #ordenarPorSeleccion
				else
					#ordenarPorBurbuja
			endif
		endif
	endif;

-----------Reglas declarativas para la transformacion del mundo no variable a la arquitectura-----------------

--Puntos de entrada
rule Entrada
{
	from ejemplo : mundo!EjemploCupi2
	to salida : negocio!Negocio(clase<-ejemplo.elemento)
}

--Elemento Agrupador
rule Agrupador
{
	from agrupador		: mundo!Agrupador
	to agrupadorNegocio	: negocio!Agrupador(
						name<-agrupador.name,
						esPrincipal<-agrupador.esPrincipal,
						elemento<-agrupador.elemento,
						atributo<-agrupador.atributo,
						relacionM<-
agrupador.relacionElementoM,
						relacion<-agrupador.relacionElemento
					)
	do
	{
		--Rasgo Persistencia Serializable
		if(thisModule.haySerializacion(agrupador))
		{
			--Metodos Guardar, Cargar, atributo ruta y extender el agrupado de serializable

	thisModule.PersistenciaSerializacion2Negocio(agrupadorNegocio.elemento, agrupadorNegocio);
		}
```

```
--Rasgo Persistencia Archivo
if(thisModule.hayArchivo(agrupador))
{
    thisModule.PersistenciaArchivo2Negocio(agrupadorNegocio); --Metodos Guardar, Cargar
y atributo ruta
    thisModule.PersistenciaArchivo2NegocioSimple(agrupador.elemento,
agrupadorNegocio.elemento);
}
--Crear el atributo para la agrupacion
thisModule.agregarAtributoAgrupador(agrupador.elemento, agrupadorNegocio);

--Constructor con parametros por defecto
thisModule.agregarConstructorParametros(agrupadorNegocio, agrupador);
        }
}

--ElementoSimple
rule Simple
{
        from simple                    : mundo!Simple
        to   simpleNegocio  : negocio!Simple(
                                                        name<-simple.name,
                                                        atributo<-simple.atributo,
                                                        relacionM<-simple.relacionElementoM,
                                                        relacion<-simple.relacionElemento
                                                )
        do
        {
                thisModule.agregarConstructorParametros(simpleNegocio, simple);
        }
}

--Atributo
rule Atributo
{
        from atributo                  : mundo!Atributo(atributo.oclIsTypeOf(mundo!Atributo))
        to   atributoNegocio  : negocio!Atributo(
                                                        name<-atributo.name,
                                                        tipo<-atributo.tipo
                                                )
        do
        {
                thisModule.crearSetterGetter(atributoNegocio);
        }
}

--AtributoCupi2
rule AtributoCupi2
{
        from atributoCupi2                      :
mundo!AtributoCupi2(atributoCupi2.oclIsTypeOf(mundo!AtributoCupi2))
        to   atributoCupi2Negocio      : negocio!AtributoCupi2(
                                                        name<-atributoCupi2.name,
                                                        tipo<-atributoCupi2.tipo,
                                                        esComparable<-
atributoCupi2.esComparable,
                                                        esIndice<-atributoCupi2.esIndice,
                                                        esMatriz<-atributoCupi2.esMatriz,
                                                        esNombre<-atributoCupi2.esNombre,
                                                        esVisualizador<-
atributoCupi2.esVisualizador,
                                                        esArreglo<-atributoCupi2.esArreglo,
                                                        tipoAlgoritmo<-
thisModule.obtenerAlgoritmoTexto(atributoCupi2.tipoAlgoritmo)
                                                )
        do
        {
                --Rasgo ordenamientos sobre estructuras de datos
                if(atributoCupi2Negocio.esComparable)
                {
                        let elemento : negocio!Elemento =
thisModule.obtenerElementoClase(atributoCupi2Negocio.clase) in
```

```
				thisModule.crearServiciosComparables(elemento.agrupador, atributoCupi2Negocio,
atributoCupi2);
			}
			thisModule.crearSetterGetter(atributoCupi2Negocio);
		}
}

--Herencia
rule HerenciaElemento
{
	from herencia			: mundo!HerenciaElemento
	to   herenciaNegocio : negocio!Herencia(
					clase<-herencia.elemento,
					claseM<-herencia.elementoM
				)
}

--Asociacion
rule AsociacionElemento
{
	from	asociacion				: mundo!AsociacionElemento
	to	asociacionNegocio : negocio!Asociacion(
					nombre<-asociacion.name,
					clase<-asociacion.elemento,
					claseM<-asociacion.elementoM
				)
				--Una Asociacion indica listas enlazadas o doblemente enlazadas por lo tanto
se crean nuevos servicios
				--Si es una herencia, se replica en la Arquitectura para la segunda
transformacion a Java
				do
				{
					--Si es lista doblemente encadenada
					if(asociacion.esDoble)
					{
						--Agregar los metodos para el manejo de listas doblemente
encadenadas
						let elemento : negocio!Elemento =
thisModule.obtenerElementoClase(asociacionNegocio.claseM) in
						thisModule.crearMetodosListasDobles(elemento.agrupador,
asociacionNegocio.claseM);
					}
					--Si es lista simple
					else
					{
							--Si es una relacion entre el mismo elemento

	if(asociacionNegocio.claseM.name.startsWith(asociacionNegocio.clase.name))
							{
								--Relacion entre el mismo elemento, los
servicios se crean en su agrupador
								let elemento : negocio!Elemento =
thisModule.obtenerElementoClase(asociacionNegocio.claseM) in

	thisModule.crearMetodosListas(elemento.agrupador, asociacionNegocio.clase);--asociacionNegocio.claseM
							}
							--Si es una relacion entre elementos diferentes
							else
							{
								--Relacion entre diferentes elementos,
se crean en claseM y clase respectivamente

	thisModule.crearMetodosListas(asociacionNegocio.claseM, asociacionNegocio.clase);
							}
					}
				}
}
-----------------------Reglas Imperativas que crean los elementos necesarios para cada Rasgo----------------------------

--Regla imperativa que cumple el rasgo Persistencia Archivo
rule PersistenciaArchivo2Negocio(clase : negocio!Clase)
{
	to
```

```
		--Por ahora como el nivel 7
		guardarArchivo	:	negocio!Servicio(nombre <- 'guardar'+clase.name, retorno<-#Nulo, parametro<-
Sequence{parametro1}, tipoServicio<-#PAguardarArchivo),
		parametro1		:	negocio!Parametro(nombre<- 'param',tipo<-#Texto),
		cargarArchivo	:	negocio!Servicio(nombre <- 'cargar'+clase.name, retorno<-#Nulo, tipoServicio<-
#PAcargarArchivo),
		atributoRuta	:	negocio!Atributo(name<-'archivo'+clase.name, tipo<-#Texto)--Agrupadora

		do
		{
			clase.servicio<-clase.servicio->union(Sequence{guardarArchivo, cargarArchivo});
			clase.atributo<-clase.atributo->union(Sequence{atributoRuta});
		}
}
--Regla imperativa que cumple el rasgo Persistencia Archivo
rule PersistenciaArchivo2NegocioSimple( mundo : mundo!Elemento, clase : negocio!Elemento)
{
		to
		servicioConstructorArchivo : negocio!Servicio(nombre<-mundo.name, parametro<-parametro1, retorno<-
#Nulo, tipoServicio<-#PAconstructorArchivo),
		parametro1: negocio!Parametro(nombre<-'param', tipo<-#Objeto, especial<-'BufferedReader' ),
		guardar : negocio!Servicio(nombre<-'guardar'+mundo.name, parametro<-parametro2, retorno<-#Nulo,
tipoServicio<-#PAguardarElemento),
		parametro2: negocio!Parametro(nombre<-'param', tipo<-#Objeto, especial<-'PrinterWriter' )
		do
		{
			clase.servicio<-clase.servicio->union(Sequence{servicioConstructorArchivo, guardar});
		}
}

--Regla imperativa que cumple el rasgo Persistencia Serializacion
rule PersistenciaSerializacion2Negocio(agrupado : negocio!Clase, agrupador : negocio!Clase)
{
		to
		guardar		:	negocio!Servicio(nombre <- 'guardar'+agrupador.name, retorno<-#Nulo,
tipoServicio<-#PSguardarObjeto),
		cargar		:	negocio!Servicio(nombre<-'cargar'+agrupador.name, parametro<-
Sequence{parametro1}, retorno<-#Nulo,tipoServicio<-#PScargarObjeto),
		parametro1	:	negocio!Parametro(nombre<-'param',tipo<-#Texto),
		atributoRuta:	negocio!Atributo(name<-'archivo'+agrupador.name, tipo<-#Texto)--Agrupadora
		--herencia	:	negocio!Herencia(nombre<-'Serializable')
		do
		{
			agrupador.servicio<-agrupador.servicio->union(Sequence{guardar, cargar});
			agrupador.atributo<-agrupador.atributo->union(Sequence{atributoRuta});
			agrupado.implementa<-'Serializable';
		}
}

--Regla imperativa que cumple con el rasgo Lista Encadenada, agrega los metodos y atributos:
rule crearMetodosListas(claseAgrupadora:negocio!Clase, claseAgrupada : negocio!Clase)
{
		to
		agregaralinicio			:		negocio!Servicio(nombre <-
'agregar'+claseAgrupada.name+'AlInicio',	retorno<-#Nulo, parametro<-Sequence{parametro1}, tipoServicio<-
#LEagregarInicio),
		parametro1			:	negocio!Parametro(nombre <-'elemento' , tipo<-#Objeto,
especial<-claseAgrupada.name),
		agregaralfinal			:		negocio!Servicio(nombre <-
'agregar'+claseAgrupada.name+'AlFinal',	retorno<-#Nulo, parametro<-Sequence{parametro2}, tipoServicio<-
#LEagregarFinal),
		parametro2			:	negocio!Parametro(nombre <-'param' , tipo<-#Objeto,
especial<-claseAgrupada.name),
		agregarantesde			:		negocio!Servicio(nombre <-
'agregar'+claseAgrupada.name+'AntesDe',	retorno<-#Nulo, parametro<-Sequence{parametro3,parametro4},
tipoServicio<-#LEagregarAntes),
		parametro3			:	negocio!Parametro(nombre <-'elemento' , tipo<-#Objeto,
especial<-claseAgrupada.name),
		parametro4			:	negocio!Parametro(nombre <-'indice' , tipo<-#Entero),
		agregardespuesde	:		negocio!Servicio(nombre <-
'agregar'+claseAgrupada.name+'DespuesDe',	retorno<-#Nulo, parametro<-Sequence{parametro5,parametro6},
tipoServicio<-#LEagregarDespues),
```

```
		parametro5			:	negocio!Parametro(nombre <-'elemento' , tipo<-#Objeto,
especial<-claseAgrupada.name),
		parametro6			:	negocio!Parametro(nombre <-'indice' , tipo<-#Entero),
		localizar			:		negocio!Servicio(nombre <-
'localizar'+claseAgrupada.name,		retorno<-#Objeto, parametro<-Sequence{parametro7}, especial<-
claseAgrupada.name, tipoServicio<-#LElocalizar),
		parametro7			:			negocio!Parametro(nombre <-'indice' ,
tipo<-#Entero),
		localizarAnterior		:		negocio!Servicio(nombre <-
'localizarAnterior'+claseAgrupada.name,		retorno<-#Objeto, parametro<-Sequence{parametro9}, especial<-
claseAgrupada.name, tipoServicio<-#LElocalizarAnterior),
		parametro9			:			negocio!Parametro(nombre <-'indice' ,
tipo<-#Entero),
		localizarUltimo			:		negocio!Servicio(nombre <-'localizarUltimo',
		retorno<-#Objeto, especial<-claseAgrupada.name, tipoServicio<-#LElocalizarUltimo),
		elminar				:			negocio!Servicio(nombre <-'eliminar',
		retorno<-#Nulo, parametro<-Sequence{parametro10}, tipoServicio<-#LEeliminar),
		parametro10			:			negocio!Parametro(nombre <-'indice' ,
tipo<-#Entero),
		darLongitud			:			negocio!Servicio(nombre <-
'darLongitud',		retorno<-#Entero, tipoServicio<-#LEdarLongitud),
		darSiguiente			:		negocio!Servicio(nombre <-'darSiguiente',
		retorno<-#Objeto, especial<-claseAgrupada.name, tipoServicio<-#LEdarSiguiente ),
		cambiarSiguiente		:		negocio!Servicio(nombre <-'cambiarSiguiente',	retorno<-
#Nulo, parametro<-Sequence{parametro11}, tipoServicio<-#LEcambiarSiguiente ),
		parametro11			:			negocio!Parametro(nombre <-
'elemento' , tipo<-#Objeto, especial<-claseAgrupada.name),
		insertarDespues			:		negocio!Servicio(nombre <-'insertarDespues',
		retorno<-#Nulo, parametro<-Sequence{parametro12}, tipoServicio<-#LEinsertarDespues ),
		parametro12			:			negocio!Parametro(nombre <-
'elemento' , tipo<-#Objeto, especial<-claseAgrupada.name),
		desconectarSiguiente		:		negocio!Servicio(nombre <-'desconectarSiguiente',
		retorno<-#Nulo,tipoServicio<-#LEdesconectarSiguiente ),
		enlace				:			negocio!Atributo(name<-'lePrimero',
tipo<-#Objeto, especial<-claseAgrupada.name),--Agrupadora
		contador			:		negocio!Atributo(name<-
'leNumero'+claseAgrupada.name, tipo<-#Entero),--Agrupadora
		indice				:			negocio!Atributo(name<-'leIndice',
tipo<-#Entero),--Agrupada
		siguiente			:		negocio!Atributo(name<-'leSiguiente', tipo<-
#Objeto, especial<-claseAgrupada.name)--Agrupada

		do
		{
			claseAgrupadora.servicio <- claseAgrupadora.servicio-
>union(Sequence{agregaralinicio,agregaralfinal, agregarantesde, agregardespuesde,
localizar,localizarAnterior,localizarUltimo, elminar, darLongitud});
			claseAgrupada.servicio<- claseAgrupada.servicio->union(Sequence{ darSiguiente,
cambiarSiguiente, insertarDespues, desconectarSiguiente});
			claseAgrupada.atributo<- claseAgrupada.atributo->union(Sequence{indice, siguiente});
			claseAgrupadora.atributo<- claseAgrupadora.atributo->union(Sequence{enlace, contador});
		}
}

-- Regla imperativa que cumple con el rasgo Lista Doblemente Encadenada, crea los servicios y atributos:
rule crearMetodosListasDobles(claseAgrupadora: negocio!Clase, claseAgrupada: negocio!Clase)
{
		to
		agregar				:			negocio!Servicio(nombre <-
'agregar'+claseAgrupada.name,		retorno<-#Nulo, parametro<-Sequence{parametro1}, tipoServicio<-
#LDEagregar),
		parametro1			:	negocio!Parametro(nombre <-'elemento' , tipo<-#Objeto,
especial<-claseAgrupada.name),
		elminar				:			negocio!Servicio(nombre <-'eliminar',
		retorno<-#Nulo, parametro<-Sequence{parametro2}, tipoServicio<-#LEeliminar),
		parametro2			:			negocio!Parametro(nombre <-'indice' ,
tipo<-#Entero),
		darSiguiente			:		negocio!Servicio(nombre <-'darSiguiente',
		retorno<-#Objeto, especial<-claseAgrupada.name, tipoServicio<-#LDEdarSiguiente),
		darAnterior			:			negocio!Servicio(nombre <-
'darAnterior',		retorno<-#Objeto, especial<-claseAgrupada.name, tipoServicio<-#LDEdarAnterior ),
		cambiarSiguiente		:		negocio!Servicio(nombre <-'cambiarSiguiente',	retorno<-
#Nulo, parametro<-Sequence{parametro3}, tipoServicio<-#LDEcambiarSiguiente ),
```

```
		parametro3				:			negocio!Parametro(nombre <-
'elemento' , tipo<-#Objeto, especial<-claseAgrupada.name),
		cambiarAnterior			:			negocio!Servicio(nombre <- 'cambiarAnterior',
		retorno<-#Nulo, parametro<-Sequence{parametro4}, tipoServicio<-#LDEcambiarAnterior ),
		parametro4				:			negocio!Parametro(nombre <-
'elemento' , tipo<-#Objeto, especial<-claseAgrupada.name),
		insertarDespues			:			negocio!Servicio(nombre <- 'insertarDespues',
		retorno<-#Nulo, parametro<-Sequence{parametro5}, tipoServicio<-#LDEinsertarDespues ),
		parametro5				:			negocio!Parametro(nombre <-
'elemento' , tipo<-#Objeto, especial<-claseAgrupada.name),
		insertarAntes			:			negocio!Servicio(nombre <- 'insertarAntes',
		retorno<-#Nulo, parametro<-Sequence{parametro6}, tipoServicio<-#LEinsertarAntes ),
		parametro6				:			negocio!Parametro(nombre <-
'elemento' , tipo<-#Objeto, especial<-claseAgrupada.name),
		desconectarSiguiente		:		negocio!Servicio(nombre <- 'desconectarSiguiente',
		retorno<-#Nulo,tipoServicio<-#LDEdesconectarSiguiente ),
		desconectarAnterior		:			negocio!Servicio(nombre <- 'desconectarAnterior',
		retorno<-#Nulo,tipoServicio<-#LDEdesconectarAnterior ),
		enlace					:			negocio!Atributo(name<-'IdePrimero',
tipo<-#Objeto, especial<-claseAgrupada.name),--Agrupadora
		contador				:			negocio!Atributo(name<-
'IdeNumero'+claseAgrupada.name, tipo<-#Entero),--Agrupadora
		indice					:			negocio!Atributo(name<-'IdeIndice',
tipo<-#Entero),--Agrupada
		siguiente				:			negocio!Atributo(name<-'IdeSiguiente', tipo<-
#Objeto, especial<-claseAgrupada.name),--Agrupada
		anterior				:			negocio!Atributo(name<-'IdeAnterior', tipo<-
#Objeto, especial<-claseAgrupada.name)–Agrupada

		do
		{
				claseAgrupadora.servicio <- claseAgrupadora.servicio->union(Sequence{agregar,elminar});
				claseAgrupadora.atributo<- claseAgrupadora.atributo->union(Sequence{enlace, contador});
				claseAgrupada.servicio<- claseAgrupada.servicio->union(Sequence{ darSiguiente, darAnterior,
cambiarSiguiente, cambiarAnterior, insertarDespues, insertarAntes, desconectarSiguiente, desconectarAnterior});
				claseAgrupada.atributo<- claseAgrupada.atributo->union(Sequence{indice, siguiente, anterior});
				thisModule.crearGetterSetter(indice);
				thisModule.crearGetterSetter(enlace);
				thisModule.crearGetterSetter(contador);

		}
}
--Metodos que se crean a partir de una atributo comparable
rule crearServiciosComparables(claseAgrupadora : negocio!Clase, atributo: negocio!AtributoCupi2, origen :
mundo!AtributoCupi2)
{
		to
		--ServicioOrdenarPor
		ordenarpor			: negocio!Servicio(nombre <- 'ordenarPor'+atributo.name,	retorno<-#Nulo,
tipoServicio<-thisModule.obtenerServicioOrdenador(origen) ),
		--ServicioBuscarPor
		buscar				: negocio!Servicio(nombre <- 'buscar'+atributo.name,		retorno<-#Nulo,
parametro<-Sequence{parametro1}, tipoServicio<-#buscar ),
		parametro1		: negocio!Parametro(nombre <-'param' , tipo<-#Objeto),			--Tipo de dato del
atributo Indice ( atributo.esNombre.tipo)

		--ServicioCompararPor
		compararpor			: negocio!Servicio(nombre <- 'compararPor'+atributo.name, retorno<-#Entero,
parametro<-Sequence{parametro2}, tipoServicio<-#comparar ),
		parametro2		: negocio!Parametro(nombre <-'param' , tipo<-atributo.tipo)
		do
		{
				claseAgrupadora.servicio <- claseAgrupadora.servicio->union(Sequence{ordenarpor, buscar});
				atributo.clase.servicio<- atributo.clase.servicio->union(Sequence{compararpor});
		}
}

--Regla que crea el constructor con parametros por defecto,
rule agregarConstructorParametros(clase : negocio!Elemento, origen : mundo!Elemento)
{
		to
		constructor			: negocio!Servicio(nombre <- clase.name,	tipoServicio<-#comparar, retorno<-
#Nulo, parametro<-Sequence{}, tipoServicio<-#constructorParametros)
```

```
	do
	{
		for(a in origen.atributo)
		{
			thisModule.agregarParametrosConstructorParametros(constructor, a);
		}
		clase.servicio<- clase.servicio->union(Sequence{constructor});
	}
}
--Regla que crea cada uno de los parametros del constructor por defecto
rule agregarParametrosConstructorParametros(servicio : negocio!Servicio, atributo : negocio!Atributo)
{
	to
	parametroConstructor	: negocio!Parametro(nombre<-'param'+atributo.name, tipo<-
atributo.tipo)
	do
	{
		servicio.parametro<- servicio.parametro->union(Sequence{parametroConstructor});
	}
}

--Regla que crea los getters y setters para los atributos de las clases
rule crearSetterGetter(atributo : Negocio!Atributo)
{
	to
	getter		: negocio!Servicio(nombre <- 'dar'+atributo.name, tipoServicio<-#dar, retorno<-
atributo.tipo), --Retorno del mismo tipo del atributo
	setter		: negocio!Servicio(nombre <- 'cambiar'+atributo.name,	retorno<-#Nulo,
tipoServicio<-#cambiar, parametro<-parametroSetter),
	parametroSetter : negocio!Parametro(nombre <-'param', tipo<-atributo.tipo )

	do
	{
		atributo.clase.servicio<- atributo.clase.servicio->union(Sequence{getter,setter});
	}
}

rule agregarAtributoAgrupador( p : mundo!Elemento, agrupadorN : negocio!Agrupador)
{
	to
	atributoAgrupador	:	negocio!Atributo(name<-'conjunto' + p.name,
tipo<-#Coleccion)
	do
	{
		agrupadorN.atributo<- agrupadorN.atributo->union(Sequence{ atributoAgrupador});
	}
}
```

12.2 Transformación Mundo A Interfaz

```
--****************************************--
--Transformacion Mundo a Interfaz	--
--Carlos Andres Parra A.	--
--c-parra1@uniandes.edu.co	--
--MDSPL Cupi2 - Qualdev Software Group--
--Universidad de los Andes 2007	--
--****************************************--
module mundo2interfaz;
create OUT : interfaz from IN : mundo, IN2 : weaving;

helper def: darFeature(nombreElemento : String) :
	String =
	weaving!Link.allInstances()->asSequence()->iterate(e; ret : String = '' |
				if e.left.description.startsWith(nombreElemento)
				then
					e.right.name
				else
```

```
                                        ret
                                        endif
                                        );

helper context weaving!Link def: darNombreVista() :
        String =
        mundo!Elemento.allInstances()->asSequence()->iterate(e; ret : String = '' |
                                        if e.name.startsWith(self.left.description)
                                        then
                                                self.right.name + e.name
                                        else
                                          ret
                                        endif
                                        );

helper context weaving!Link def: darElemento() :
        mundo!Elemento =
        mundo!Elemento.allInstances()->asSequence()->iterate(e; ret : mundo!Elemento = Set(mundo!Elemento)|
                                        if e.name.startsWith(self.left.name)
                                        then
                                                e
                                        else
                                          ret
                                        endif
                                        );

helper def :obtenerRaiz() : interfaz!Interfaz =
        interfaz!Interfaz.allInstances()->asSequence()->first();

helper def :obtenerVistaPadre() : interfaz!Vista =
        interfaz!Interfaz.allInstances()->asSequence()->first().clase->asSequence()-
>select(a|a.oclIsTypeOf(interfaz!Vista))->iterate(e; ret : interfaz!Vista= Set(interfaz!Vista) |
                if e.name.startsWith('VistaPrincipal')
                then
                                        e
                else
                        ret
                endif
        );

rule Entrada
{
        from ejemplo : mundo!EjemploCupi2
        to      interfaz: interfaz!Interfaz(clase<-Sequence{vistaPrincipal, vistaExtension}),
                vistaPrincipal : interfaz!Vista(name <- 'VistaPrincipal', relacion<-Sequence{relacion}),
                vistaExtension : interfaz!Vista(name<-'VistaExtension'),
                relacion : interfaz!Relacion(claseC<-vistaExtension)
        do
        {
                thisModule.addConstructor(vistaPrincipal, #constructorVP);
                thisModule.addMetodoPrincipal(vistaPrincipal, #principalVP);
                        --<%self.buscarElementoVP%>)
                thisModule.addConstructorVistaHija( vistaExtension, vistaPrincipal, #constructorVP);
        }

}

rule addVistaInterfaz
{
        from    link : weaving!Link(link.right.name.startsWith('Vista'))
        using
        {
                elemento : mundo!Elemento= link.darElemento();
        }

        to      vista : interfaz!Vista(name <- link.right.name + elemento.name, origen<-elemento.name),
        relacionHija : interfaz!Relacion(claseC<- vista)

        do
        {

                if(link.right.name.startsWith('VistaConjunto'))
                {
```

```
				--Componente tipo lista para manejar el conjunto de elementos
				thisModule.addComponenteInteraccionVista( vista, 'Lista', #Lista);
				--Botones de manipulacion de la lista
				for(a in elemento.atributo)
				{
					if(a.oclIsTypeOf(mundo!AtributoCupi2))
					{
						if(a.esComparable)
						{
							thisModule.addComponenteInteraccionVista( vista,
'ordenarPor'+a.name, #Boton);

		thisModule.addOrdenarPorVP(thisModule.obtenerVistaPadre(),a.name, #ordenarPorVP );
						}
					}
				}
				thisModule.addConstructorVistaHija( vista,thisModule.obtenerVistaPadre(),
#constructorVC);
				thisModule.addManejador( vista, #accionRealizadaVC);
				thisModule.addActualizarListaVP(thisModule.obtenerVistaPadre(),#actualizarListaVP );
				thisModule.addActualizarVC(vista, #actualizarVC);
				thisModule.addSeleccionarVC( vista, #actualizarVC);
			}

			if(link.right.name.startsWith('VistaAgregacion'))
			{
				--Cuadros de texto y etiquetas por cada atributo del Elemento
				--Botones para confirmar o borrar el formulario
				thisModule.addComponenteVisualizacionAtrib( vista, elemento,'etiqueta', #Etiqueta);
				thisModule.addComponenteInteraccionAtrib(vista, elemento, 'texto', #CuadroTexto);
			thisModule.addComponenteInteraccionVista( vista, 'aceptar', #Boton);
			thisModule.addComponenteInteraccionVista( vista, 'borrar', #Boton);
			--Agregar el boton examinar si hay una imagen
			for(a in elemento.atributo)
				{
					if(a.oclIsTypeOf(mundo!AtributoCupi2))
					{
						if(a.esVisualizador)
						{
							thisModule.addComponenteInteraccionVista( vista,
'examinar'+a.name, #Boton);
						}
					}
				}

			thisModule.addAgregarElementoVP(thisModule.obtenerVistaPadre(), elemento,
#agregarElementoVP );
				thisModule.addManejador( vista, #accionRealizadaVA);
				thisModule.addConstructorVistaHija( vista,thisModule.obtenerVistaPadre(),
#constructorVA);
			}
			if(link.right.name.startsWith('VistaInformacion'))
			{
			 thisModule.addConstructorVistaHija(vista,thisModule.obtenerVistaPadre(), #constructorVI);
			  thisModule.addVerInformacionVP(thisModule.obtenerVistaPadre(),
elemento.name,#verInformacionVP );
			  thisModule.addComponenteVisualizacionAtrib(vista, elemento, 'etiqueta', #Etiqueta);
			  thisModule.addComponenteInteraccionAtrib( vista, elemento, 'texto', #CuadroTexto);
			  thisModule.addMostrarDatosVI(vista, elemento.name, #mostrarDatosVI);
			  thisModule.addLimpiarDatosVI(vista, #limpiarDatosVI);
			}
			--Asociarlas a la raiz
			thisModule.obtenerRaiz().clase<-thisModule.obtenerRaiz().clase->union(Sequence{vista});
			--relacion con la principal
			thisModule.obtenerVistaPadre().relacion<-thisModule.obtenerVistaPadre().relacion-
>union(Sequence{relacionHija});
		}
}

--Agrega a la vista componentes de visualización que no se mapean desde el mundo
rule addComponenteVisualizacionVista(vi : interfaz!Vista, nombre : String, tipo : interfaz!TipoVisualizacion){
	to
```

```
			comp : interfaz!Visualizacion(
								nombre <- nombre,
								tipoVisualizacion <- tipo

						)
		do
		{
			vi.componente <- vi.componente->union(Sequence{comp});
		}
}

--Agrega a la vista componentes de interacción que no se mapean desde el mundo
rule addComponenteInteraccionVista( vi : interfaz!Vista, nombre : String, tipo : interfaz!TipoInteraccion)
{
		to
			comp : interfaz!Interaccion(
								nombre <- nombre,
								tipoInteraccion <- tipo

						)
		do
		{
			vi.componente <- vi.componente->union(Sequence{comp});
		}
}

--Agrega a la vista componentes de visualización que están relacionados a los atributos
rule addComponenteVisualizacionAtrib( vi : interfaz!Vista, elemento : mundo!Elemento, nombre : String, tipo :
interfaz!TipoVisualizacion)
{
		do
		{
			for(a in elemento.atributo)
			{
						thisModule.addComponenteVisualizacionVista(vi, nombre + a.name, tipo);
			}
		}
}

--Agrega a la vista componentes de interacción que están relacionados a los atributos
rule addComponenteInteraccionAtrib(vi : interfaz!Vista, elemento : mundo!Elemento, nombre: String, tipo :
interfaz!Tipo)
{
		do
		{
			for(a in elemento.atributo)
			{
					thisModule.addComponenteInteraccionVista( vi, nombre + a.name, tipo);
			}
		}

}

--Agrega el servicio constructor Principal
rule addConstructor(clase : interfaz!Vista, tipo : interfaz!TipoServicioInterfaz)
{
		to
		servicio : interfaz!Servicio(nombre <- clase.name, retorno<-#Nulo, parametro<-parametroC, tipoServicio<-tipo
),
		parametroC : interfaz!Parametro(nombre<-'param', tipo<-#Objeto)
		do
		{
			clase.servicio<- clase.servicio->union(Sequence{servicio});
		}
}

--Agrega el servicio constructor Vistas Hijas
rule addConstructorVistaHija(clase : interfaz!Vista, clasePadre : interfaz!Vista, tipo : interfaz!TipoServicioInterfaz)
{
		to
		servicio : interfaz!Servicio(nombre <- clase.name, retorno<-#Nulo, parametro<-parametroC, tipoServicio<-tipo
),
		parametroC : interfaz!Parametro(nombre<-'param', tipo<-#Objeto, especial<-clasePadre.name)
```

```
		do
		{
			clase.servicio<- clase.servicio->union(Sequence{servicio});
		}
}

--Agrega el servicio de manejo de eventos
rule addManejador(clase : interfaz!Vista, tipo : interfaz!TipoServicioInterfaz)
{
		to
		servicio	: interfaz!Servicio(nombre <-'actionPerformed',	retorno<-#Nulo, parametro<-parametro1,
tipoServicio<-tipo),
   parametro1  : interfaz!Parametro( nombre<-'param' , tipo<-#Objeto, especial<-'ActionEvent')
		do
		{
				clase.servicio<- clase.servicio->union(Sequence{servicio});
		}
}

--Agrega el main
rule addMetodoPrincipal(clase : interfaz!Vista, tipo : interfaz!TipoServicioInterfaz)
{
		to
		servicio	: interfaz!Servicio(nombre <-'main',	retorno<-#Nulo, parametro<-parametro1, tipoServicio<-tipo
),
   parametro1  : interfaz!Parametro( nombre<-'param' , tipo<-#Texto, especial<-'args')
		do
		{
				clase.servicio<- clase.servicio->union(Sequence{servicio});
		}
}
--Agrega los ordenar por a la interfaz principal
rule addOrdenarPorVP(clase : interfaz!Vista, nombre : String, tipo : interfaz!TipoServicioInterfaz)
{
		to
		servicio	: interfaz!Servicio(nombre <-'ordenarPor'+nombre,		retorno<-#Nulo, tipoServicio<-tipo)
		do
		{
				clase.servicio<- clase.servicio->union(Sequence{servicio});
		}
}
--Agrega metodo actualizar al main para el rasgo de vista conjunto
rule addActualizarListaVP(clase : interfaz!Vista, tipo : interfaz!TipoServicioInterfaz)
{
		to
		servicio	: interfaz!Servicio(nombre <-'actualizarLista',	retorno<-#Nulo, tipoServicio<-tipo )
		do
		{
				clase.servicio<- clase.servicio->union(Sequence{servicio});
		}
}
--Agrega metodo agregar a la vista principal  para el rasgo de vista agregacion
rule addAgregarElementoVP(clase : interfaz!Vista, elemento: mundo!Elemento, tipo : interfaz!TipoServicioInterfaz)
{
		to
		servicio	: interfaz!Servicio(nombre <-'agregar'+elemento.name,	parametro<-Sequence{}, retorno<-
#Nulo, tipoServicio<-tipo )
		do
		{
			for(a in elemento.atributo)
				{
					thisModule.agregarParametroAgregar(servicio, a);
				}
			clase.servicio<- clase.servicio->union(Sequence{servicio});
		}
}

--Agrega parametros al servicio agregar
rule agregarParametroAgregar(servicio : interfaz!Servicio, a : mundo!Atributo)
{
		to
		parametro		: interfaz!Parametro(nombre<-a.name , tipo<-a.tipo)
		do
```

```
		{
					servicio.parametro<- servicio.parametro->union(Sequence{parametro});
		}
}

--Agrega metodo que actualiza la lista para el rasgo de Vista informacion
rule addVerInformacionVP(clase : interfaz!Vista, elemento: String, tipo : interfaz!TipoServicioInterfaz)
{
		to
		servicio	: interfaz!Servicio(nombre <-'verConjunto',		parametro<- parametro1, retorno<-#Nulo,
tipoServicio<-tipo),
		parametro1 : interfaz!Parametro(nombre<-'param', tipo<- #Objeto, especial<-elemento)
		do
		{
					clase.servicio<- clase.servicio->union(Sequence{servicio});
		}
}


rule addMostrarDatosVI(clase : interfaz!Vista, elemento : String , tipo : interfaz!TipoServicioInterfaz)
{
		to
		servicio	: interfaz!Servicio(nombre <-'mostrarDatos',		parametro<- parametro1, retorno<-#Nulo,
tipoServicio<-tipo),
		parametro1 : interfaz!Parametro(nombre<-'param', tipo<- #Objeto, especial<-elemento)
		do
		{
					clase.servicio<- clase.servicio->union(Sequence{servicio});
		}
}

rule addLimpiarDatosVI(clase : interfaz!Vista, tipo : interfaz!TipoServicioInterfaz)
{
		to
		servicio	: interfaz!Servicio(nombre <-'LimpiarDatos',		retorno<-#Nulo, tipoServicio<-tipo )
		do
		{
					clase.servicio<- clase.servicio->union(Sequence{servicio});
		}
}

rule addActualizarVC(clase : interfaz!Vista, tipo : interfaz!TipoServicioInterfaz)
{
		to
		servicio	: interfaz!Servicio(nombre <-'actualizar', retorno<-#Nulo, parametro<- parametro1, tipoServicio<-tipo
),
		parametro1 : interfaz!Parametro(nombre<-'param', tipo<- #Coleccion)

		do
		{
					clase.servicio<- clase.servicio->union(Sequence{servicio});
		}
}

rule addSeleccionarVC(clase : interfaz!Vista, tipo : interfaz!TipoServicioInterfaz)
{
		to
		servicio	: interfaz!Servicio(nombre <-'seleccionar',		retorno<-#Nulo, parametro<- parametro1,
tipoServicio<-tipo),
				parametro1 : interfaz!Parametro(nombre<-'param', tipo<- #Entero)
		do
		{
					clase.servicio<- clase.servicio->union(Sequence{servicio});
		}
}
```

12.3 Transformación Arquitectura a Java

```
--******************************************--
--Transformacion Arquitectura a Java        --
--Carlos Andres Parra A.                    --
--c-parra1@uniandes.edu.co                  --
--MDSPL Cupi2 - Qualdev Software Group--
--Universidad de los Andes 2007             --
--******************************************--
module arquitectura2java;
create OUT : java from IN1 : negocio, IN2 : interfaz, IN3 : wnt, IN4 : wit;

--Helpers
--Helper que obtiene el tipo de dato en java basado en el tipo de dato de la arquitectura
helper def: tipoDatoJava(tipo : negocio!TipoDatoNegocio, nombre :String ) : java!JavaType =
	if tipo = #Texto
	then #JString
		else if tipo = #Entero
		then #Jint
			else if tipo = #Boleano
			then #Jboolean
				else if tipo = #Doble
				then #Jdouble
					else if tipo = #Objeto
					then #JObject
						else if tipo = #Coleccion
						then
thisModule.obtenerImplementacionColeccion(nombre)--#JArrayList
							else if tipo = #Nulo
							then #Jvoid
								else #Jvoid
							endif
						endif
					endif
				endif
			endif
		endif
	endif;

--Obtiene el tipo de coleccion del entrelazado entre negocio y tecnologia
helper def: obtenerImplementacionColeccion( nombre : String ) : java!JavaType =
		wnt!Link.allInstances()->asSequence()->iterate(e; ret java!JavaType = #Jvoid|
		if (e.left.name.startsWith(nombre)) then
			if e.right.name.startsWith('ArrayList') then
				#JArrayList
			else
				#JVector
		endif
		else
			ret
		endif
	);


--Helpers para los tipos de elementos de visualizacion e interaccion
helper def: tipoComponenteInteraccionJava(tipo : interfaz!TipoInteraccion, nombre :String) : java!JavaType =
	if tipo = #Boton
	then #JJButton
		else if tipo = #CajaVerificacion
		then #JJCheckBox
			else if tipo = #Lista
			then thisModule.obtenerImplementacionLista(nombre)
				else if tipo = #CuadroTexto
				then #JJTextField
					else #Jvoid
				endif
			endif
		endif
	endif;

--Obtiene el tipo de lista del entrelazado entre interfaz y tecnologia
```

```
helper def: obtenerImplementacionLista(nombre : String ) : java!JavaType =
		wit!LinkallInstances()->asSequence()->iterate(e; ret:java!JavaType = #JJList|
		if (e.left.name.startsWith(nombre)) then
			if e.right.name.startsWith('JList') then
				#JJList
			else
				#JJComboBox
		endif
		else
			ret
		endif
	);

helper def: tipoComponenteVisualizacionJava(tipo : interfaz!TipoVisualizacion) : java!JavaType =
	if tipo = #Etiqueta
	then #JJLabel
		else if tipo = #Imagen
		then #JJLabel
			else if tipo = #Mensaje
			then #Jvoid
				else #Jvoid
			endif
		endif
	endif;

--Helper que obtiene el tipo de servicio de negocio
helper def: tipoServicio(tipo : negocio!TipoServicio ) : Integer =
	if tipo = #LEagregarInicio
	then 1
		else if tipo = #LEagregarFinal
		then 2
			else if tipo = #LEagregarAntes
			then 3
				else if tipo = #LEagregarDespues
				then 4
					else if tipo = #LElocalizar
					then 5
						else if tipo = #LElocalizarUltimo
						then 6
							else if tipo = #LEeliminar
							then 7
								else if tipo = #LDEagregar
								then 8
									else if tipo =
#LDEcambiarSiguiente
									then 9
										else if tipo =
#LDEcambiarAnterior
										then 10
											else if
tipo = #LDEdarSiguiente
											then 11

	else if tipo = #LDEdarAnterior

	then 12

	else if tipo = #PSguardarObjeto

	then 13

		else if tipo = #PScargarObjeto

		then 14

			else if tipo = #PAguardarArchivo

			then 15

				else if tipo = #PAcargarArchivo

				then 16
```

```
else if tipo = #dar
then 17
    else if tipo = #cambiar
    then 18
        else if tipo = #ordenarPorBurbuja
        then 19
            else if tipo = #buscar
            then 20
                else if tipo = #comparar
                then 21
                    else if tipo =
#constructorNegocio
                    then 22
                        else if
tipo = #LElocalizarAnterior
                        then 23
else if tipo = #LEdarLongitud
then 24
else if tipo = #LEcambiarSiguiente
then 25
    else if tipo = #LEinsertarDespues
    then 26
        else if tipo = #LEdesconectarSiguiente
        then 27
            else if tipo = #LDEinsertarDespues
            then 28
                else if tipo = #LDEinsertarAntes
                then 29
                    else if tipo = #LDEdesconectarSiguiente
                    then 30
```

```
else if tipo = #LDEdesconectarAnterior
then 31
    else if tipo = #LDEeliminar
    then 32
        else if tipo =
#PAconstructorArchivo
        then 33
            else if tipo =
#PAguardarElemento
            then 34
                elseif
tipo = #constructorParametros
                then 35
    else if tipo = #ordenarPorInsercion
    then 36
    else if tipo = #ordenarPorSeleccion
    then 37
        else 100
    endif
    endif
                endif
            endif
        endif
    endif
```

```
                                                            endif

                                                      endif

                                                endif

                                          endif

                                    endif

                              endif

                        endif

                        endif
                                                                                    endif
                                                                              endif
                                                                        endif
                                                                  endif
                                                            endif
                                                      endif
                                                endif
                                          endif
                                    endif
                              endif
                        endif
                        endif
                                                                                    endif
                                                                              endif
                                                                        endif
                                                                  endif
                                                            endif
                                                      endif
                                                endif
                                          endif
                                    endif
                              endif
                        endif;

--Helper que obtiene el tipo de servicio de negocio
helper def: tipoServicioInterfaz(tipo : negocio!TipoServicio) : Integer =
      if tipo = #constructorVP
      then 1
            else if tipo = #principalVP
            then 2
                  else if tipo =#ordenarPorVP
                  then 3
                        else if tipo =#actualizarListaVP
                        then 4
                              else if tipo =#verInformacionVP
                              then 5
                                    else if tipo =#agregarElementoVP
```

```
then 6
    else if tipo =#buscarElementoVP
    then 7
        else if tipo =#constructorVA
        then 8
            else if tipo
=#accionRealizadaVA
            then 9
                else if tipo
=#constructorVI
                then 10
                    else if
tipo =#mostrarDatosVI
                    then 11
    else if tipo =#limpiarDatosVI
    then 12
    else if tipo =#constructorVC
    then 13
        else if tipo =#actualizarVC
        then 14
            else if tipo =#seleccionarVC
            then 15
                else if tipo =#accionRealizadaVC
                then 16
                    else 100
                endif
            endif
        endif
    endif
    endif
                    endif
                endif
            endif
        endif
    endif
endif
endif
endif
endif
endif
endif;

helper def: obtenerPaqueteMundo() : java!Package =
    java!Package.allInstances()->asSequence()->iterate(e; ret : java!Package = Set(java!Package)|
                if e.name.startsWith('mundo')
                then
                    e
                else
                  ret
                endif
                );


helper def: obtenerPaqueteMundo() : java!Package =
    java!Package.allInstances()->asSequence()->iterate(e; ret : java!Package = Set(java!Package)|
                if e.name.startsWith('mundo')
                then
```

```
                    e
                else
                    ret
                endif
                );


rule Entrada
{
        from interfaz : interfaz!Interfaz
        to
        raiz                : java!JavaApp( packageJavaApp<-Sequence{packNegocio, packInterfaz}),
        packNegocio : java!Package(name<-'mundo', javaClassPackage<-Sequence{}),
        packInterfaz: java!Package(name<-'interfaz', javaClassPackage<-interfaz.clase)
}


rule Agrupador2Clase
{
        from agrupador              : negocio!Agrupador
        to  claseAgrupador : java!Container(name<- agrupador.name, method<- agrupador.servicio, field<-
agrupador.atributo, references<-agrupador.elemento)
        do
        {
                thisModule.obtenerPaqueteMundo().javaClassPackage<-
thisModule.obtenerPaqueteMundo().javaClassPackage->union(Sequence{claseAgrupador});
        }
}

rule Simple2Clase
{
        from simple         : negocio!Simple
        to  claseSimple : java!Simple(name<-simple.name, method<- simple.servicio, field<- simple.atributo,
implement<-simple.implementa)
        do
        {
                thisModule.obtenerPaqueteMundo().javaClassPackage<-
thisModule.obtenerPaqueteMundo().javaClassPackage->union(Sequence{claseSimple});
        }
}

rule Vista2Clase
{
        from vista                  : interfaz!Vista
        to  clase           : java!Interface( name<-vista.name, method<- vista.servicio, origen<-vista.origen, field<-
Sequence{ vista.componente})
        do
        {
                if(vista.name.endsWith('Principal'))
                {
                        for(p in vista.relacion)
                        {
                                thisModule.agregarCampoRelacionH(p, clase);
                        }
                }
                else
                {
                        thisModule.agregarCampoRelacionP(vista.relacionC, clase);
                }
        }


}

rule ServicioNegocio2Method
{
        from servicio : negocio!Servicio
        to  method          :java!Method( name <- servicio.nombre,
                                                        parameter<-servicio.parametro,
                                                        returnType<-
thisModule.tipoDatoJava(servicio.retorno, ''),
                                                        body<-cuerpo,
```

```
                                                                  specialType<-servicio.especial),
        cuerpo                          : java!Body(bodyId<- thisModule.tipoServicio(servicio.tipoServicio))--
servicio.tipoServicio)
}

rule ServicioInterfaz2Method
{
        from servicio  : interfaz!Servicio
        to  method           : java!Method(name <-servicio.nombre, parameter<-servicio.parametro,

        returnType<-thisModule.tipoDatoJava(servicio.retorno, ' '),

        body<-cuerpo,

        specialType<-servicio.especial),
        cuerpo                          : java!Body(bodyId<-  thisModule.tipoServicioInterfaz(servicio.tipoServicio))

}

rule ParametroNegocio2Parameter
{
        from parametro  : negocio!Parametro
        to  method             :java!Parameter(name<-parametro.nombre,

                type<-thisModule.tipoDatoJava(parametro.tipo,''),

                specialType<-parametro.especial)
}

rule ParametroInterfaz2Parameter
{
        from parametro  : interfaz!Parametro
        to  method              : java!Parameter(name<-parametro.nombre,

                        type<-thisModule.tipoDatoJava(parametro.tipo, ''),

                        specialType<-parametro.especial)
}

rule AtributoNegocio2Field
{
        from atributo :  negocio!Atributo(atributo.oclIsTypeOf(negocio!Atributo))
        to                      field :  java!Field(name<-atributo.name, type<-thisModule.tipoDatoJava(atributo.tipo,
atributo.name), specialType<-atributo.especial )
}

rule AtributoCupi2Negocio2Field
{
        from atributo :  negocio!AtributoCupi2(atributo.oclIsTypeOf(negocio!AtributoCupi2))
        to                      field : java!Cupi2Field(name<-atributo.name,
                                                                        type<-
thisModule.tipoDatoJava(atributo.tipo, atributo.name),
                                                                        isIndex<-atributo.esIndice,
                                                                        isName<-atributo.esNombre,
                                                                        isComparable<-
atributo.esComparable,
                                                                        isImage<-
atributo.esVisualizador,
                                                                        specialType<-
atributo.especial,
                                                                        algorithm<-
atributo.tipoAlgoritmo
                                                                        )
}

rule Interaccion2Field
{
        from interaccion : interfaz!Interaccion
        to          field                   : java!Field(name<-interaccion.nombre,
                                                                                  type<-
thisModule.tipoComponenteInteraccionJava(interaccion.tipoInteraccion, interaccion.nombre))
}
```

```
rule Visualizacion2Field
{
		from visualizacion : interfaz!Visualizacion
		to		field			:java!Field(name<-visualizacion.nombre,
											type <-
thisModule.tipoComponenteVisualizacionJava(visualizacion.tipoVisualizacion)	)
}


rule agregarCampoRelacionH(p : interfaz!Relacion, clase : java!Interface)
{
		to	hijo	: java!Field(name<-p.claseC.name.toLower(), type<-#JObject, specialType<-p.claseC.name)
		do
		{
				clase.field<- clase.field->union(Sequence{hijo});
		}
}

rule agregarCampoRelacionP(p: interfaz!Relacion, clase : java!Interface)
{
		to	padre	: java!Field(name<-'padre', type<-#JObject, specialType<-p.clase.name)
		do
		{
				clase.field<- clase.field->union(Sequence{padre});
		}
}
```