Universidad Autónoma Metropolitana

Unidad Iztapalapa

Ciencias Básicas e Ingeniería

Maestría en Ciencias y Tecnologías de la Información

Un enfoque MDA para el desarrollo de aplicaciones basadas en un modelo de componentes orientados a servicios

Idónea comunicación de resultados

Presenta: Néstor A. Riba Zárate.

Asesor: Dr. Humberto Cervantes Maceda

Julio 2007

Dedicado a...

Dios.

Por ser mi guía, abrirme puertas en todo el trayecto y mostrarme su amor en todo momento.

Mi familia, Arce, Genaro, Marce y Emi.

Por ser un ejemplo constante de superación y esfuerzo, por el amor y el apoyo brindado en todos los sentidos para poder realizar este trabajo y así poder continuar con mi desarrollo profesional.

Mi novia, Naeily.

Por ser una inspiración constante, por su amor, por ser una fuerza que me motiva constantemente a lograr todo lo que me proponga, saber que no existen límites en lo que se puede ser.

Mi asesor, Humberto.

Por su invitación a participar en este proyecto, por su paciencia, sus conocimientos y su amistad. Por su guía en este trabajo y sus consejos constantes.

Mi Universidad, la UAM-I.

Por ser la institución que me ha formado tanto en el aspecto profesional como en el personal y por todos los apoyos brindados en mis estudios.

Agradecimientos:

LAFMI (Laboratorio Franco Mexicano de Informática). Por el apoyo para la realización de este proyecto.

Profesores de la UAM-I. Por compartir sus conocimientos que ayudaron en la realización de este proyecto.

Lionel Touseau, Didier Donsez y el equipo ADELE la Universidad Joseph Fourier. Por el trato amable y el apoyo que me brindaron en mi estancia en Grenoble.

Victor Ramos. Por su tutoría y consejos al inicio de mis estudios de maestría.

Indice

1. Introducción

1.1. Componentes y servicios

En épocas recientes el desarrollo de aplicaciones de software ha crecido enormemente. Las aplicaciones que se construyen en la actualidad tienen que diseñarse para soportar distintas situaciones independientes de los aspectos funcionales y estar construidas para cambiar. Esto se debe en parte a que en la industria del software el surgimiento de nuevas tecnologías y plataformas es constante, frecuentemente surgen nuevas plataformas que provocan que las aplicaciones se vuelvan obsoletas en cuestión de pocos meses o años, cuando lo que se desea es todo lo contrario, construir una aplicación capaz de durar más tiempo mediante el mantenimiento o como en el caso de otras industrias sucede, a través de la sustitución de componentes viejos por unos más actualizados y con nueva funcionalidad. Sumado a esto, se encuentra la necesidad de acelerar los procesos de desarrollo software mediante la reutilización de bloques de software en la construcción del software nuevo. Para ayudar a resolver estos problemas se planteó la solución de dividir la aplicación en pequeños bloques de código reutilizables llamados componentes. Un componente empaqueta funcionalidad y ofrece interfaces para conectarse con otros componentes. Así, la aplicación puede ensamblarse de manera más simple a través del ensamblado de componentes que generalmente es realizado por un tercero ajeno al desarrollo de los componentes que ensambla. Esto facilita el mantenimiento de las aplicaciones ya que pueden sustituirse componentes individuales de la aplicación sin afectar al todo, además, este enfoque favorece la reutilización de código ya que los componentes pueden ser reutilizados en la construcción de nuevas aplicaciones acelerando el proceso de creación de las mismas. A este enfoque de construcción de aplicaciones de software se le conoce como desarrollo basado en componentes [1] [44].

El desarrollo basado en componentes supone que los componentes que se usan en una aplicación están disponibles en el momento del ensamblado. Esta suposición, sin embargo, no es siempre válida, dado que ciertas aplicaciones requieren de poder modificar su arquitectura, es decir sus componentes e interconexiones, en tiempo de ejecución de manera dinámica. El soporte del dinamismo, sin embargo, requiere de la escritura de código dedicado a su manejo que puede fácilmente superar en complejidad a la lógica del negocio de la aplicación. Sumado a esto, otra necesidad frecuente es la de soportar cierto nivel de interoperabilidad entre aplicaciones heterogéneas de tal forma que una aplicación pueda usar los servicios provistos por otra.

Un enfoque alternativo al desarrollo basado en componentes estándar plantea una solución. En él, los componentes son ensamblados en tiempo de ejecución, es decir, conforme se vayan requiriendo sus funcionalidades o servicios. De la misma forma cuando una aplicación ya no requiere los servicios de determinados componentes, estos pueden ser removidos de la aplicación, todo en tiempo de ejecución. Dado que los componentes aparecen y desaparecen dinámicamente, su ubicación se realiza a través de un registro en donde los componentes publican los servicios que ofrecen. Cualquier aplicación puede buscar en el registro a los servicios que requiera para su funcionamiento. Los componentes se ligan en tiempo de ejecución y a partir de ese momento pueden trabajar juntos. A este enfoque de arquitectura se le conoce como arquitectura orientada a servicios (SOA por sus siglas en inglés *Service Oriented Architecture*). Componentes y servicios no son mutuamente exclusivos, de hecho, a un enfoque que combina desarrollo basado en componentes y una arquitectura orientada a servicios se le conoce como modelo de componentes orientados a servicios [1]. Este enfoque será un punto fundamental dentro de éste trabajo. Cabe señalar que este enfoque abre muchas posibilidades entre las que se incluyen la construcción de aplicaciones que se son capaces de autoconfigurarse de acuerdo a sus propias

necesidades y de aplicaciones que seleccionan los componentes que les ofrezcan, por ejemplo, mejor calidad de servicio.

1.2. Dificultades del desarrollo de componentes orientados a servicios

El desarrollo de aplicaciones basadas en componentes orientados a servicios es complejo, esto provoca que los desarrolladores tiendan a centrar sus esfuerzos los aspectos relacionados con los aspectos no funcionales, dejando detrás a la lógica del negocio (requerimientos funcionales). Esto representa riesgos en el desarrollo y puede terminar en aplicaciones técnicamente complejas que no se comportan de forma adecuada.

Entre las principales razones por las cuales el desarrollo de aplicaciones basadas en componentes orientados a servicios es complejo están:

a)Programación del dinamismo. Programar a una aplicación para que maneje de manera adecuada el dinamismo es una tarea compleja, aspectos como responder a los cambios en su ambiente, servicios que van y vienen, seleccionar los componentes adecuados para autoensamblarse, saber qué hacer cuando no cuenta con algún componente en particular, etc. representa un reto técnico. Muchas veces esto se soluciona programando una lógica de adaptación que se encarga de estos aspectos, sin embargo suele suceder que esta lógica en muchos de los casos pueda llegar a ser tan compleja que sobrepase incluso a la lógica de negocios de la aplicación.

b)Plataformas orientadas a servicios en constante cambio. La orientación a servicios es relativamente nueva y por lo tanto nuevas tecnologías que implementan este enfoque surgen y cambian todo el tiempo, de manera que nuestras aplicaciones pueden volverse obsoletas en un lapso muy corto de tiempo.

c)**Plataformas orientadas a servicios heterogéneas.** Debido al constante surgimiento de nuevas plataformas orientadas a servicios, una aplicación que utilice este enfoque probablemente tendrá que terminar interactuando con componentes de otras plataformas, de ésta forma los desarrolladores están obligados a conocer en periodos muy cortos de tiempo muchas plataformas tecnológicas para poder hacer el ensamblado de su aplicación.

1.3. Enfoque declarativo de definición de componentes

Una solución que se ha explorado para resolver el problema del manejo del dinamismo en aplicaciones basadas en componentes orientados a servicios consiste en extraer la lógica de manejo del dinamismo del código de los componentes y especificar su comportamiento de manera declarativa.

Para crear un componente siguiendo este enfoque se debe crear, junto con la implementación de los aspectos de la lógica del negocio, un archivo XML, que, en el caso particular de el modelo de componentes, es llamado descriptor de componente. Este descriptor contiene información sobre la estructura del componente así como su comportamiento con respecto a los cambios que ocurren en tiempo de ejecución. Esta información es utilizada por el entorno de ejecución para controlar el dinamismo sin que el programador tenga que preocuparse por ello [36] [38].

Pero, aunque el enfoque declarativo de componentes libera al programador de la carga del manejo del dinamismo, no simplifica completamente su tarea, el desarrollador tiene que conocer a detalle cómo construir este llamado descriptor del componente siguiendo una especificación que frecuentemente resulta bastante extensa. A lo largo de ésta especificación se presentan un conjunto de

reglas a seguir en la construcción de un componente siguiendo este enfoque, el conocerlas antes de desarrollar el componente puede llegar a ser en muchos casos un trabajo que requiere de mucho tiempo. El trabajo del desarrollador es menor, pero aún así, sigue siendo complicado.

Además no hay que olvidar que aun tenemos los problemas relacionados con los cambios tecnológicos constantes. Se requiere crear nuevas formas de desarrollar software, procesos en los cuales, tareas de programación complejas o repetitivas puedan ser automatizadas, de manera que el desarrollador pueda enfocar sus esfuerzos en la parte funcional de las aplicaciones.

1.4. Arquitecturas dirigidas por modelos

Mientras por un lado los desarrolladores están preocupados por el desarrollo de aplicaciones mas dinámicas y autoadaptables, creando maneras de desarrollar componentes de una forma mas simple (como el enfoque declarativo), en paralelo existe un grupo cuya principal preocupación es acelerar e industrializar el proceso de creación de software mas allá de la reutilización de componentes.

La mayor parte de las industrias comenzaron el desarrollo de sus productos de manera manual, ese es el caso de las industrias automotriz y electrónica, donde en sus inicios construir un automóvil o una televisión era un proceso puramente manual, que requería de mucho tiempo. Con el avance de la tecnología y la llegada de las computadoras los procesos de construcción de aparatos electrónicos y automóviles se automatizaron, de tal forma que ahora la mayor parte de la construcción de éstos productos la realizan robots manejados por computadoras.

Trasladar este concepto a la industria del software ha sido una inquietud y tema de investigación desde los inicios de la ingeniería de software. Esto es debido que gran parte del tiempo de desarrollo de un producto de software es consumido por la construcción del mismo, además de que, gran cantidad de errores en la producción de software son introducidos en esta fase de su desarrollo.

El enfoque plantea que, como en las otras industrias donde gran parte del diseño de un nuevo producto se realiza mediante el modelado del mismo, que en el desarrollo de software, estos modelos sirvan de entrada para que las computadoras construyan el producto con poca o ninguna intervención humana.

A este enfoque de desarrollo de software se le conoce como MDA (por sus siglas en inglés *Model Driven Architecure*) o desarrollos dirigidos por modelos [10]. Este plantea que lo más importante en el desarrollo de una aplicación son los modelos de la misma obtenidos durante la fase de diseño. Para MDA, si estos modelos tienen la información suficiente, entonces la aplicación puede crearse casi en su totalidad a partir de ellos. MDA plantea que la total construcción de la aplicación sea realizada de forma automatizada, sin embargo, como sucede en las demás industrias, un enfoque mas realista a corto plazo sería dejar a las computadoras las tareas repetitivas y comunes a todos los productos que se construyen, tareas que no requieran la intervención humana, dejando a los programadores tareas de programación menos complejas y que de alguna forma no pueden ser realizadas por una computadora. Aspectos no funcionales como los relacionados con la arquitectura de la aplicación y el soporte de cierta tecnología son dejados a la computadora mientras que los aspectos funcionales relacionados con la lógica del negocio son dejados a los programadores, es decir, los desarrolladores no deben preocuparse por conocer en detalle el funcionamiento de una plataforma en particular.

Una ventaja importante y la finalidad principal de MDA es hacer a las aplicaciones mas independientes de los cambios tecnológicos, debido a que los modelos son independientes de cualquier plataforma, y si en algún momento se requiere actualizar la aplicación para que soporte una nueva

tecnología, sólo se tiene que volver a generar el código a partir de los modelos. Es decir, para MDA, los modelos son lo más importante, no el código.

Para poder implementar esta idea de software que construye software, se requiere construir nuevas aplicaciones, que sigan un proceso bien definido, que sean fácilmente extensibles a las nuevas necesidades que vayan surgiendo y que no sean dependientes de una plataforma en particular. Otros aspectos tienen que ser considerados, pero sin duda el más importante es establecer un proceso relativamente simple que cualquier empresa de desarrollo de software pudiera seguir para crear sus propias herramientas MDA, de manera que este enfoque pudiera penetrar en la industria del desarrollo de software.

La idea es que en un futuro, si surge una nueva plataforma tecnológica, y se ha utilizado un enfoque MDA, entonces puede volver a generarse el nuevo código ahora para la nueva plataforma utilizando los mismos modelos que fueron utilizados para crear la aplicación en un inicio, ya que estos son independientes de la plataforma. Si se requiere conectar dos aplicaciones o varios componentes cuyas plataformas no son las mismas, se puede conectar sus modelos y dejar que la computadora se encargue de la compleja programación de esta conexión. Además, en muchas proyectos de desarrollo de software implementar nuevos requerimientos en una aplicación ya existente o modificar alguno para agregar funcionalidad no es un proceso simple, pero con MDA, la modificación se realiza a nivel de modelos, dejando a la herramienta MDA con el trabajo de hacer la implementación.

Aunque MDA puede ser utilizado bajo cualquier enfoque, se puede notar que MDA y SOA van de la mano, MDA puede ser utilizado para resolver muchos de los problemas tecnológicos que se presentan en el desarrollo de aplicaciones orientadas a servicios. Sin embargo es importante recalcar que MDA va mas allá de SOA y puede ser utilizado en otros dominios ajenos completamente a la orientación a servicios.

1.5. Herramienta MDA para construir componentes orientados a servicios

Regresando al contexto de desarrollo de componentes orientados a servicios siguiendo un enfoque declarativo, y al problema que se tiene, la complejidad en la construcción de descriptores de componente, o en su caso, la compleja programación que se tiene que desarrollar para manejar el dinamismo de la aplicación, un enfoque MDA puede acelerar de manera importante el proceso de desarrollo de componentes. En una herramienta MDA se realizaría un modelo que especifique las características no funcionales del componente, la herramienta entonces se encargará de la generación automática del descriptor del componente o en su caso de la programación de aspectos no funcionales repetitivos.

En el caso de la generación automática de descriptores de componente, la herramienta acelera el desarrollo debido a que el programador no requiere conocer a detalle la estructura del descriptor, ya que la herramienta integra todo este conocimiento y además es capaz de validar los modelos creados basándose en una serie de reglas obtenidas de una especificación que el programador no requiere conocer, dejando a este solo con el trabajo de la programación de la lógica funcional de su componente.

Este proyecto tiene tres objetivos principales, primero, la definición de un lenguaje de modelado de aplicaciones basadas en componentes orientados a servicios. Segundo, la construcción de una herramienta MDA que implemente el lenguaje definido y facilite el desarrollo de componentes siguiendo un enfoque declarativo y por último, establecer un proceso para la construcción de

herramientas MDA para cualquier otro dominio en particular que pueda ser retomado por cualquier organización en la creación de sus propias herramientas. Al momento de escritura de este documento no fue detectada ninguna herramienta de este tipo para el dominio de componentes orientados a servicios ni para ninguna arquitectura cuya base sea un modelo declarativo.

El proceso que se defina para el desarrollo de esta herramienta debe ser simple, rápido y barato, de manera que pueda ser fácilmente adoptado por cualquier organización dedicada al desarrollo de software que requiera la automatización de parte de sus procesos de construcción, contribuyendo así a la adopción de MDA en la industria.

Antes de presentar en detalle los objetivos del proyecto, la metodología utilizada, los resultados obtenidos y el resto de este trabajo, es necesario conocer en detalle el funcionamiento de MDA y SOA. El siguiente capítulo de este trabajo presenta un conjunto de conceptos básicos sobre SOA y MDA para posteriormente presentar un estado del arte de SOA y las herramientas MDA que existen actualmente en el mercado de manera que se tenga un punto de referencia para medir la importancia del trabajo que se realizó en este proyecto. Después se hace una presentación detallada de los objetivos de nuestro proyecto. Se presenta nuestro proceso de investigación y desarrollo y al final, los resultados obtenidos y las conclusiones generales.

2. Conceptos básicos sobre SOA y MDA

A continuación se presentan los conceptos más relevantes para la comprensión del resto de este documento, primero se presentan los relacionados al desarrollo de aplicaciones orientadas a servicios y posteriormente, los relacionados con las arquitecturas dirigidas por modelos.

2.1. Arquitecturas orientadas a componentes y a servicios

La orientación a servicios es un enfoque relativamente nuevo para desarrollar aplicaciones. Es una forma distinta de construir aplicaciones basándose en entidades llamadas servicios. En términos sencillos puede considerarse un servicio como un bloque de código reutilizable que ofrece cierta funcionalidad. El concepto de reutilizable se refiere a que el mismo servicio puede ser utilizado por varias aplicaciones en el ensamblado de las mismas. Este concepto de servicios puede llegar a ser similar al concepto de componente, ambos comparten el concepto de armado de aplicaciones vía bloques separados que pueden ser construidos y ensamblados por distintos actores [1] en distintos tiempos.

Sin embargo la orientación a servicios y los componentes poseen diferencias importantes, que solamente pueden entenderse si se comprenden ambos conceptos.

2.2. Orientación a componentes

En primer lugar, un componente es una "unidad binaria de composición con interfaces y dependencias contextuales explícitas" [44], es decir un componente "compone" una aplicación uniéndose a otros componentes mediante interfaces bien definidas, algunas de las cuales serán conexiones explícitamente creadas para establecer la configuración del componente, es decir, sus requerimientos, y otras que implementan la funcionalidad del componente. "Un componente puede ser implementado de forma independiente y es sujeto a composición por terceros", de tal forma que las aplicaciones pueden ser ensambladas mediante la composición de componentes, los cuales no necesariamente tuvieron que ser creados al mismo tiempo que la aplicación. La manera de lograr esta independencia es mediante paquetes de componente, que son unidades que contienen todo lo necesario para que un componente funcione correctamente [1][2]. En la figura 1 se presenta cómo se realiza el desarrollo de aplicaciones basadas en componentes. Existe un desarrollador encargado de desarrollar componentes para distintos dominios, éstos son tomados por un desarrollador de aplicaciones para ensamblar un aplicación funcional.

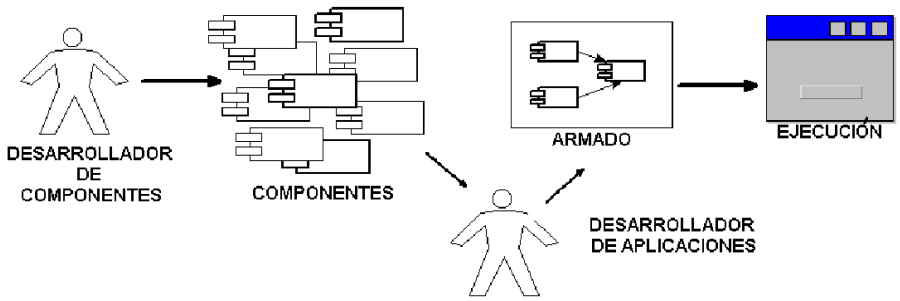

Figura 1: Pasos del desarrollo de aplicaciones basadas en componentes

2.3. Orientación a servicios

A diferencia de las aplicaciones basadas en componentes donde la aplicación es ensamblada previo a su ejecución, como se muestra en la figura 2, en las aplicaciones orientadas a servicios las aplicaciones son ensambladas de manera tardía integrando los servicios a la aplicación durante su ejecución. En la figura se aprecia un desarrollador de servicios que registra sus servicios en un llamado registro de servicios donde las aplicaciones pueden buscar por aquellos servicios que requieran e integrarlos de manera dinámica a su arquitectura.

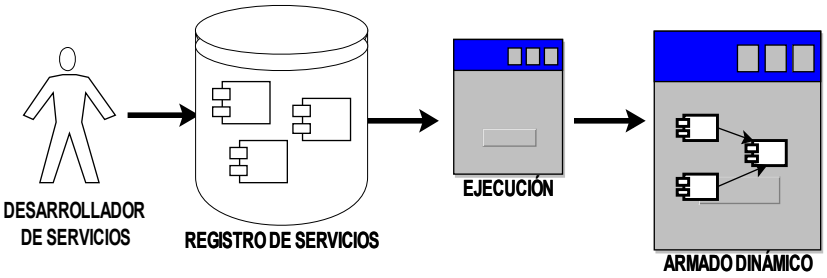

Figura 2: Pasos para el desarrollo de aplicaciones basadas en servicios

Desde el punto de vista del negocio, un servicio provee una función del negocio como validar el historial crediticio de un cliente o procesar una orden de compra, estas funciones pueden ser discretas (como un servicio que ofrece realizar el cálculo en un cambio de divisas) o complejas (como puede ser ofrecer un conjunto de funciones que provee un sistema de reservación de una aerolínea) [2]. En otras palabras la orientación a servicios es una forma de compartir funciones en una manera amplia y flexible.

Aunque componentes y servicios tienen muchas similitudes, la diferencia entre ambos se da en el momento de la integración, los componentes generalmente son ensamblados al momento de la construcción de la aplicación, mientras que en la orientación a servicios estos son ensamblados durante la ejecución de la aplicación. El enfoque de los servicios se basa en el concepto de descubrimiento y ensamblado dinámico, mientras que los componentes en la composición de aplicaciones mas estáticas, lo que hace que las aplicaciones orientadas a servicios sean mas dinámicas, aspecto que debe tomarse en cuenta durante la ejecución de las mismas ya que la programación de dicha funcionalidad puede llegar a ser muy compleja.

2.3.1. Arquitecturas orientadas a servicios

Las aplicaciones orientadas a servicios tienen dos tipos de participantes: los *proveedores* y los *consumidores* de servicios. Los proveedores son los responsables de proveer objetos de servicios que ofrezcan cierta funcionalidad. Los consumidores de servicios son clientes de los proveedores y requieren de la funcionalidad que estos proporcionan para funcionar. Existe una entidad adicional llamada *registro de servicios* la cual sirve de intermediario entre ambos (ver figura 3). El registro contiene una serie de descripciones de servicios y referencias a proveedores de estos, además de proveer de mecanismos de publicación, remoción y descubrimiento de servicios. La publicación ocurre cuando un proveedor se registra en el registro de servicios antes de poder ser utilizado, la remoción es el proceso mediante el cual un servicio es retirado del registro de servicios y el descubrimiento es el proceso mediante el cual los consumidores encuentran los servicios que les son de utilidad. El registro cambia constantemente de acuerdo a como los proveedores son publicados y removidos de este.

La manera en que un consumidor (que a su vez podría estar ofreciendo un servicio) se comunica

con el servicio no depende de la implementación del servicio de manera que este no conoce mucho del proveedor del servicio, no requiere saber en qué plataforma se encuentra o en qué lenguaje está implementado, ya que como se mencionó, el consumidor se comunica con el servicio vía una interfaz bien definida independiente de la implementación, es decir, uno de los aspectos mas importantes de la orientación a servicios es que separa el "qué" del "cómo" [4]. Esto es gracias a la creación de lo que se conoce como un contrato. Un contrato especifica el formato de la petición del consumidor y la respuesta del proveedor así como variables tales como niveles en la calidad del servicio y las precondiciones y postcondiciones en las cuales el servicio debe encontrarse antes y después de ejecutar una función particular [4].

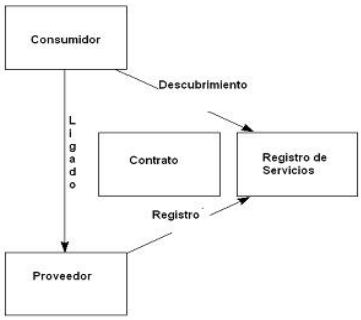

Figura 3: Distintos actores en la orientación a servicios

La orientación a servicios soporta la posibilidad de que los consumidores de servicios pueden optar por cambiar de proveedores particulares de un servicio en cualquier momento durante la ejecución de la aplicación, sin necesidad de verse atado a ninguno. El proceso para la conexión entre los consumidores y proveedores es la siguiente (ver figura 4):

a) Un proveedor publica una descripción de servicio ofrecido en el registro de servicios.

b) Un consumidor consulta el registro para descubrir servicios basado en algún criterio en particular utilizando las descripciones de servicios publicadas.

c) El registro de servicios hace una búsqueda entre todos los proveedores registrados buscando aquellos que ofrezcan el servicio requerido.

d) Si un proveedor de servicios cumple con las características buscadas, el registro de servicios regresa al consumidor una referencia a el proveedor.

e) En el caso en que múltiples proveedores sean regresados, el consumidor deberá elegir un proveedor especifico al cual sera ligado mediante un proceso de filtrado.

f) El consumidor utiliza la referencia al proveedor seleccionado para ligarse con el mismo.

g) El proveedor devuelve un un objeto que implementa el servicio al consumidor, para su uso.

h) El proveedor devuelve al consumidor la referencia al objeto de servicio creado.

i) El consumidor interactúa con el objeto de servicio agregándolo a su estructura.

j) Finalmente cuando el consumidor no requiere mas del servicio, el objeto es liberado de manera implícita o explícita [1].

k) El proveedor destruye el objeto de servicio que fue liberado por el consumidor.

l) Para que el proveedor deje de ofrecer el servicio es necesario remover la referencia a éste del registro de servicios.

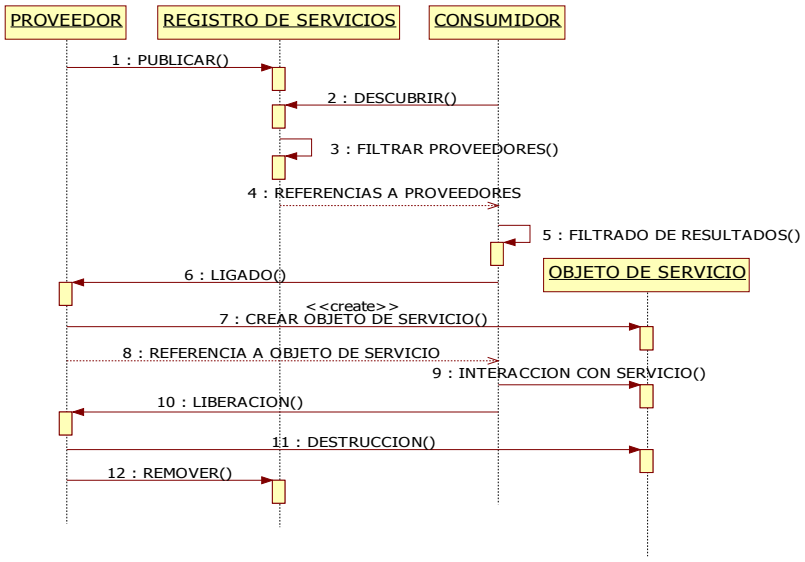

Figura 4: Proceso de registro, descubrimiento y ligado de servicios

Un ambiente de ejecución orientado a servicios debe proveer mecanismos para que los consumidores y proveedores puedan implementar el patrón de interacción orientado a servicios. Existen dos mecanismos básicos que permiten implementar dicho patrón: (a) El registro de servicios donde los proveedores pueden publicar o remover descripciones de servicios y los consumidores pueden consultar y obtener referencias a proveedores (mediante el uso de filtros) y (b) mediante notificaciones en los cambios en los servicios, donde el ambiente notifica los eventos de publicación, modificación o remoción de servicios por parte de los proveedores para que los consumidores estén enterados de la disponibilidad de éstos e incorporarlos a su estructura o, en caso de desaparecer, detener los que ya no estén disponibles.

2.3.2. Ventajas de SOA

Existen actualmente muchas razones por las cuales una organización se puede beneficiar al implementar soluciones orientadas a servicios, entre estas tenemos:

a) **Alto nivel de reutilización.** Los desarrolladores pueden tomar código desarrollado en aplicaciones existentes, exponerlas como servicios y reutilizarlas para implementar las nuevas necesidades del negocio. Los beneficios de la reutilización de servicios crece dramáticamente mientras más servicios son desarrollados e incorporados a las aplicaciones y sistemas.

b) **Interoperabilidad**. Una arquitectura orientada a servicios permite la comunicación e interacción de proveedores y consumidores sin importar en qué plataforma se encuentran ejecutándose, mediante la implementación de estándares de comunicación.

c) **Extensibilidad**. Debido al bajo nivel de acoplamiento entre objetos de servicios las aplicaciones que los utilizan son más fáciles de escalar. Esto se debe a que existen muy pocas dependencias entre la aplicación y los servicios que utiliza.

d) **Flexibilidad**. En una aplicación donde el nivel de acoplamiento es alto los objetos dependen demasiado unos de otros, esto hace difícil que la aplicación evolucione para cumplir con los requerimientos del negocio. La naturaleza asíncrona, basada en documentos (contratos) y de bajo nivel de acoplamiento de SOA permiten a las aplicaciones mucha flexibilidad.

2.4. Estado del arte de las tecnologías orientadas a servicios

Existen en la actualidad muchas tecnologías que siguen un enfoque orientado a servicios, cada una de ellas con distintas metas e implementación. Algunas de estas son: Corba Trader™, utilizada en ambientes CORBA; JavaBeans Context, utilizada para agrupar JavaBeans en tiempo de ejecución; Jini™, utilizada para la carga dinámica de código desde una red; Web Services, que se utiliza para exponer funcionalidad de aplicaciones completas como servicios y Microsft .NET™, alternativa de Microsoft a los Web Services. Debido a que el detalle del funcionamiento de todas éstas tecnologías no es relevante para este trabajo, éstas no fueron incluidas, pero, para un comparativo de estas tecnologías el lector puede referirse a [1]. Este proyecto está enfocado en particular a las necesidades surgidas en el desarrollo de aplicaciones orientadas a servicios bajo la plataforma llamada OSGi, por lo que este trabajo solo presentará en detalle el funcionamiento de esta plataforma.

2.4.1. OSGi™ : una plataforma basada en componentes orientados a servicios

La *Open Services Gateway Initiative (OSGi™)* es una corporación no lucrativa que promueve especificaciones abiertas para el uso de servicios en automóviles, casas y otros ambientes [1][35]. Las especificaciones están divididas en dos partes: el *framework* de OSGi y la definición de un conjunto de servicios estándares. Un *framework* puede definirse como un diseño de software reutilizable, de manera que los diseñadores y programadores no tengan que preocuparse tanto por aspectos no funcionales y ocuparse más de los funcionales, un *framework* puede contener programas de soporte y código de librerías. Aunque el *framework* fue originalmente concebido para su uso en ambientes restringidos como aparatos electrónicos (figura 5), ahora ha sido ampliamente aceptado en otros ambientes siendo el soporte y base del IDE (Ambiente de desarrollo integrado) de Eclipse [35].

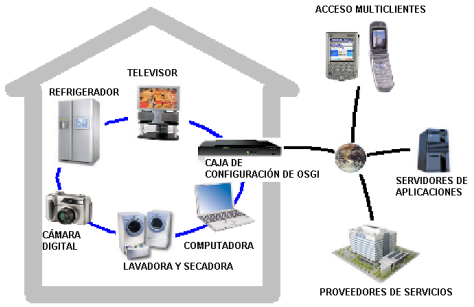

Figura 5: Uso de OSGi para conexión de dispositivos

El *framework* provee mecanismos para implantar proveedores y consumidores de servicios. Los proveedores y consumidores son entregados y desplegados vía un paquete lógico y físico llamado *bundle*, véase la figura 6. Los proveedores publican los servicios que ofrecen en un registro de servicios. Los consumidores buscan y descubren servicios que cubren sus necesidades en el registro de servicios de acuerdo al proceso descrito en la figura 4. Físicamente un bundle corresponde a una unidad de entrega e implementación dentro de un paquete de archivos similar a un archivo comprimido llamado archivo *jar* que contiene el código y los recursos (imágenes, librerías, etc), además de un

archivo que contiene información acerca del *bundle*, llamado archivo *manifest.*

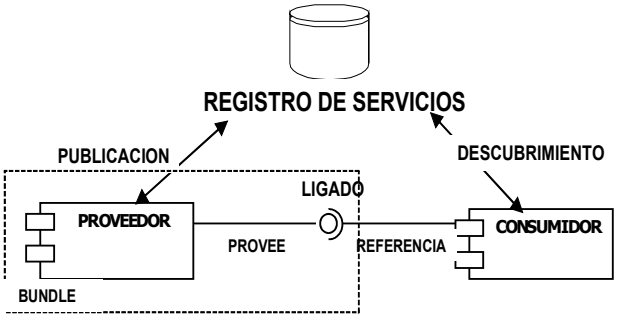

Figura 6: Elementos que componen el modelo de OSGi

El *framework* provee de mecanismos para instalar, activar, desactivar, actualizar y remover los *bundles*. Cuando un *bundle* es activado puede publicar o descubrir servicios y ligarse a ellos a través de un registro de servicios que ofrece el mismo *framework*. El descubrimiento de servicios se lleva a cabo mediante la operación *getServiceReferences* que regresará un conjunto de referencias a objetos que proveen dicho servicio, esta consulta recibe como parámetro una cadena que contiene un filtro de consulta en LDAP (*Lightweight Directory Access Protocol*) que permite el filtrado de los resultados de acuerdo a ciertos atributos de los servicios.

Para publicar un servicio el proveedor debe ejecutar una operación llamada *registerService* que recibe el nombre del servicio junto con el objeto del servicio y un diccionario de atributos. OSGi soporta dos distintas políticas de creación de objetos, la creación de objetos compartidos entre varios consumidores y la creación de un objeto por consumidor. Para implementar la segunda un objeto llamado *ServiceFactory* es usado, este es responsable de la creación de instancias de objetos de servicio.

OSGi ofrece un servicio de notificación para consumidores de servicios usando el patrón de diseño Observer que se implementa usando *listeners* de Java. Los eventos soportados son publicación, revocación y modificación de servicios. Para remover un servicio del registro se utiliza una referencia al objeto (previamente definida durante su registro) para ejecutar el método *unregister*.

2.4.2. Service Binder: un modelo de componentes orientados a servicios

En el modelo de componentes orientado a servicios, el manejo del dinamismo se considera un aspecto no funcional y puede ser manejado por un contenedor que forme parte del entorno de ejecución.

De esta forma, un enfoque propuesto en [38] propone especificar de manera declarativa en un documento XML la definición de los aspectos no funcionales de los componentes. Esta definición es utilizada por un ambiente de ejecución implementado como un servicio en OSGi que se encarga de manejar, de manera transparente para el desarrollador, aspectos complejos de la programación de componentes orientados a servicios relacionados con el manejo del dinamismo de una aplicación de este tipo, así, el programador ya no tiene que lidiar con ello. A este servicio encargado de manejar el dinamismo de los componentes definidos de forma declarativa se le llamó *Service Binder* [38]. Este enfoque fue bien recibido en el entorno empresarial y fue retomado como parte de la especificación de OSGi en su versión 4 en donde se le llamó "Servicios Declarativos" [36], los cuales son la base de este trabajo y serán discutidos en la sección 3.1.1. El *Service Binder* ha sido utilizado dentro de distintos proyectos en empresas tales como Schneider Electric, General Motors, Trialog y Ascert.

El *Service Binder* introduce el concepto de componente de servicio (ver figura 7). Este expone un conjunto de interfaces de servicio ofrecidas y un conjunto de interfaces de servicio requeridas. Durante la ejecución una instancia del componente de servicio se conecta a otras instancias para crear una aplicación. Existen también como parte del componente algunas propiedades que son utilizadas para identificar la instancia y configurar al servicio cuando estos se publican en el registro de servicios.

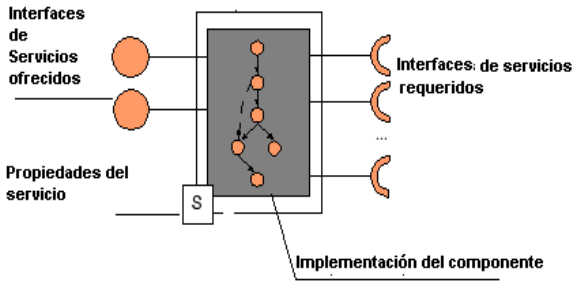

Figura 7: Componente de servicio

Los componentes son descritos dentro de un archivo XML llamado descriptor de servicio, que está contenido dentro del *bundle*. Un ejemplo sencillo de un descriptor se muestra a continuación, mas adelante en este trabajo se explicará como se construye este documento:

```
<?xml version="1.0" encoding="UTF-8"?>
<bundle>
  <component class="org.simpleclient.impl.ServiceImpl">    ---> Clase de implentación
    <provides service="org.simpleclient.interfaces.SimpleClientServiceA"/>    ---> Servicios que ofrece
    <provides service="org.simpleclient.interfaces.SimpleClientServiceB"/>
    <property name="provider" value="Beanome.org" type="string"/>          ---> Propiedades de instancia
    <requires                                                              ---> Servicios requeridos
        service="org.simpleservice.interfaces.SimpleService"               ---> Parámetros de control del dinamismo
        cardinality="1..n"
        policy="static"
        bind-method="setServiceReference"
    />
  </component>
</bundle>
```

Durante la ejecución toda instancia de componente es manejada independientemente por el entorno de ejecución. El entorno de ejecución (implementado como un servicio de OSGi) se encarga de las actividades relacionadas con el registro de servicios y administración de dependencias de servicios basándose en la información contenida en el descriptor del servicio.

En el modelo de componentes orientado a servicios, las instancias de componentes pueden tener dos estados: valido e inválido. El estado de la instancia es relativo al hecho de que las dependencias del componente estén satisfechas o no, entendiéndose por dependencia aquellos servicios que requiere para comenzar a funcionar, y, dependiendo de los cambios que ocurren durante la ejecución con estas dependencias, las instancias alternan entre estados de validez e invalidez. Cabe señalar que las aplicaciones construidas a partir de este modelo evolucionan continuamente y son capaces de adaptarse de manera autónoma con respecto a cambios en entorno de ejecución.

2.5. Arquitecturas dirigidas por modelos (MDA - *Model Driven Architecture*)

2.5.1. Importancia del modelado

Los modelos proveen abstracciones de un sistema físico que permiten a los ingenieros analizar el sistema ignorando detalles complejos mientras se enfocan en las partes mas relevantes, como puede ser la lógica del negocio. Todas las formas de ingeniería se basan en el uso de modelos para facilitar la comprensión de sistemas complejos, los modelos son utilizados en muchas formas: para predecir la calidad de un sistema, razonar acerca de propiedades específicas cuando ciertos aspectos del sistema cambian, y para comunicar características claves del sistema a todos los involucrados en el desarrollo. Los modelos pueden ser los precursores de la implementación física del sistema o ser generados de sistemas ya existentes para comprender su funcionamiento. [11]

En el mundo del software, el modelado existe desde hace tiempo, se remonta a los primeros días de la programación. El estado actual de esta práctica emplea el lenguaje de modelado unificado o UML (por sus siglas en inglés *Unified Modeling Laguage*) como la arma principal en la notación del modelado [18]. UML es un lenguaje de símbolos específicamente diseñado para la representación de la estructura y funcionamiento de aplicaciones de software. UML permite a los equipos de desarrollo capturar una variedad de características importantes de un sistema, en modelos. Durante el proceso de desarrollo de software, se realizan transformaciones entre modelos, por ejemplo, el modelo de análisis es transformado en un modelo de diseño y así hasta llegar al código. Sin embargo, la transformación entre estos modelos es primordialmente manual, lo que puede llegar a ser complejo. Una forma útil de describir la distintas maneras en que es utilizado el modelado en la actualidad, es observar las distintas formas en que el código es sincronizado con el modelo, tal como se muestra en la figura 8, éstas aproximaciones al modelado van desde no utilizar modelos en ninguna fase del desarrollo hasta proyectos donde solo se realizan modelos conceptuales sin llegar a nunca a codificar.

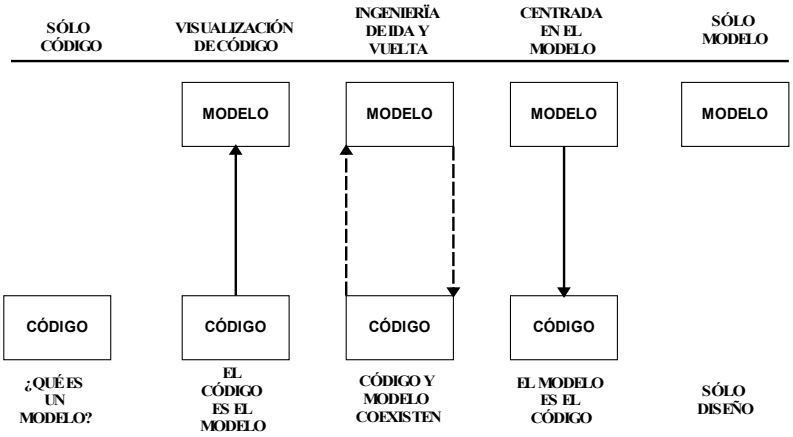

Figura 8: Distintas relaciones entre el modelo y el código

Actualmente buena parte de los desarrolladores toma la aproximación de "solo código" y utilizan poco los modelos. Esta aproximación hace difícil la evolución de estos sistemas dado que en muchos de los casos los desarrolladores originales del sistema no se encuentran durante las fases de mantenimiento del mismo.

Una mejora es proveer de visualizaciones de código en alguna notación adecuada, esto es, mientras el desarrollador va generando el código, éste es visualizado para entender de mejor manera su estructura.

Las ventajas del modelado pueden ser apreciadas en mayor grado cuando se habla de ingeniería de ida y vuelta (RTE por sus siglas en inglés *RoundTrip Engineering*) que ofrece un intercambio bidireccional entre el modelo abstracto que describe la arquitectura del sistema y el código. Esta aproximación requiere de mucha disciplina de los participantes ya que puede ocurrir un defasamiento entre los modelos y su implementación si no hay comunicación y procesos bien establecidos [11].

En la aproximación centrada en el modelo, los modelos del sistema tienen un nivel de detalle lo suficientemente completo como para generar la implementación completa del sistema basándose solo en los modelos. El proceso de generación del código puede llegar a aplicar una serie de reglas de transformación, los cuales generalmente le permiten al desarrollador la elección entre los distintos patrones que podrán ser aplicados a los modelos para transformarlos en modelos más complejos o código.

La última aproximación está basada sólo en modelos, en ésta, los desarrolladores usan los modelos sólo como ayuda para entender y comprender el negocio o el dominio del problema, o simplemente para analizar la arquitectura de una solución posible.

2.5.2. Desarrollos dirigidos por modelos (MDD-*Model Driven Development*).

MDD es un enfoque de desarrollo de software basado en modelos y en la generación de la aplicación a partir de éstos. Solo proporciona una estrategia general a seguir en el desarrollo de software pero no define ni técnicas, fase del proceso, ni ninguna guía metodológica [45].

La meta de éste enfoque es la automatización del proceso de construcción de una aplicación. Esto incluye:

- La completa generación del código de los programas a partir de los modelos.
- La verificación automática de los modelos en una computadora (probando la solución antes de construirla).

Con la generación completa del código, es raro, o innecesario la inspección del código, sólo se tienen los modelos tal como ocurre en la actualidad con los lenguajes de 3a. generación donde no es necesario inspeccionar el código en ensamblador. Las herramientas y técnicas para hacer esto posible han llegado a un estado de madurez donde se ha vuelto práctico incluso para aplicaciones a gran escala [45].

Como parte de este esfuerzo para hacer más incrementar la aceptación del enfoque MDD, la OMG creó una serie de estándares de soporte a MDD a los cuales agrupo en las especificaciones de MDA. Así MDA es un estándar que promueve a MDD y agrupa a varios lenguajes que pueden ser utilizados para seguir un enfoque dirigido por modelos en una organización, MDA intenta estandarizar MDD, que durante muchos años ha estado a la deriva. MDA no define técnicas, etapas ni artefactos, pero si proporciona una estructura tecnológica y conceptual para poder implementar de manera correcta MDD.

Para poder comprender todos estos estándares definidos en MDA y comprender el resto de éste capítulo se presentan a continuación algunas partes fundamentales de la estructura de MDA.

2.5.3. Conceptos básicos de MDA

Algunos de los conceptos más importantes que forman parte de la especificación MDA son [10]:

Modelo. Es una descripción o especificación mediante un lenguaje visual de un sistema.

Metamodelo. Es la descripción y especificación de los elementos y reglas que se utilizan para crear modelos semánticamente correctos para un dominio en particular. También puede definirse como el modelo de un lenguaje de modelado.

Dirigido por modelos (*Model Driven*). Se dice que es dirigido (o guiado) por modelos porque provee mecanismos que usan modelos para dirigir el curso del diseño, la construcción, la implementación, la operación, el mantenimiento y la modificación de una aplicación. Es decir, el proceso depende de los modelos.

Arquitectura. La arquitectura de un sistema es la especificación de las partes y conectores del sistema, así como las reglas de interacción éstas.

Vista. Es una representación del sistema desde la perspectiva de un punto de vista determinado.

Plataforma. Es un conjunto de subsistemas y tecnologías que proveen un conjunto de funcionalidades que cualquier aplicación soportada por la plataforma puede utilizar sin preocupación sobre los detalles de implementación de dicha plataforma.

Punto de vista. Un punto de vista en un sistema es una técnica de abstracción que utiliza un conjunto selecto de conceptos arquitecturales y reglas de estructuración, de manera que se enfoque la atención sólo en un problema particular del sistema. MDA especifica tres puntos de vista sobre un sistema: el punto de vista independiente de la computación, el punto de vista independiente de la plataforma y el punto de vista específico de la plataforma.

- Punto de vista independiente de la computación. Este se enfoca en el ambiente del sistema y los requerimientos del mismo, es decir, la lógica del negocio. Los detalles de la estructura y el procesamiento del sistema están escondidos o no han sido determinados.
- Punto de vista independiente de la plataforma. Se enfoca en la operación del sistema mientras oculta los detalles específicos para cierta plataforma, es decir, muestra la parte de la implementación que es idéntica de una plataforma a otra.
- Punto de vista específico de una plataforma. Combina el punto de vista independiente de la plataforma con el detalle del uso de una plataforma especifica.

Modelo independiente de la computación (CIM por sus siglas en inglés *Computer Independent Model*). Es una modelo del sistema desde el punto de vista independiente de la computación. Un CIM no muestra detalles de la estructura del sistema y a veces es llamado modelo del dominio o del negocio. El CIM forma parte importante en la construcción de puentes entre aquellos expertos en el dominio y sus requerimientos, por un lado, y aquellos expertos en el diseño y construcción de artefactos para satisfacer estos requerimientos por el otro.

Modelo Independiente de la plataforma (PIM por sus siglas en inglés *Platform Independent Model*). Es una vista del sistema desde el punto de vista independiente de la plataforma. Una técnica común para lograr la independencia de la plataforma es construir un modelo para una máquina virtual de tecnología neutral. Esta puede ser vista como un conjunto de partes que son definidas independientes de cualquier plataforma.

Modelo especifico a una plataforma (PSM por sus siglas en inglés *Platform Specific Model*). Es una vista del sistema desde el punto de vista de plataforma específica. Un PSM combina las especificaciones en el PIM con los detalles de como el sistema utiliza un tipo de plataforma en particular.

Transformación de modelos. Es el proceso de convertir un modelo a otro modelo del mismo sistema. Generalmente el PIM es combinado con alguna información adicional para producir un PSM.

2.5.4. Funcionamiento de MDA

En la actualidad, ya sabemos cómo traducir, por compilación o interpretación, un lenguaje de alto nivel (Java o SQL) a operaciones que un microprocesador es capaz de ejecutar, de igual manera, en MDA existen "compiladores" capaces de traducir modelos de datos o de la aplicación basados en UML a lenguajes de alto nivel y por lo tanto a las distintas plataformas de los sistemas actuales, pero más importante, a las plataformas del futuro [10]. La idea es que, como en la actualidad sucede en la industria automotriz, donde mucho del proceso de desarrollo de nuevos vehículos se hace en computadoras y simuladores, para después construir las partes y ensamblar los vehículos de forma automatizada, en la industria de software suceda algo similar: que el proceso de desarrollo de software se base en modelos en una computadora los cuales, inicialmente, serán PIMs y mediante transformaciones hechas por la computadora, poder generar los PSMs para una o varias plataformas, y al final transformar éstos a código que implemente la solución descrita en los modelos. Si nuevas tecnologías surgen, solo hay que transformar los modelos independientes de la plataforma a los modelos específicos de la nueva plataforma y regenerar la aplicación. Si la aplicación requiere integrarse con otras aplicaciones, se modifican los modelos, y después se regeneran las aplicaciones. Así, entre las metas de MDA se tienen: la portabilidad, la interoperabilidad y la reutilización. La siguiente figura resume el proceso de desarrollo utilizando MDA. En MDA un desarrollador sólo crea PIMs que son interpretados por una computadora para generar PSMs y posteriormente el código para distintas plataformas. De ésta forma el proceso de desarrollo es acelerado de manera considerable.

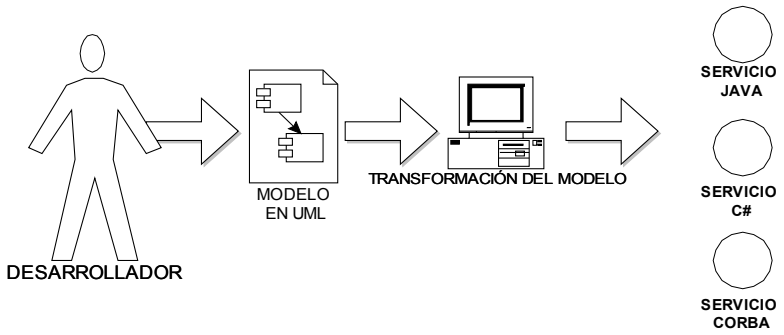

Figura 9: Proceso de desarrollo MDA

Esto es la premisa principal de una arquitectura manejada por modelos, el permitir la definición de modelos de datos y aplicaciones que puedan ser transformados por una computadora permitiendo la flexibilidad a largo plazo de:

- **Implementación**. Nueva infraestructura de implementación puede ser integrada o generada de los diseños existentes, para después regenerar el código de la aplicación.
- **Integración**. Se puede también automatizar la producción de puentes de integración de datos y la conexión entre aplicaciones.

- **Mantenimiento**. Se da acceso directo a la especificación del sistema, lo que facilita su mantenimiento, se da mantenimiento a los modelos, no al código.
- **Pruebas y simulación**. Los modelos pueden ser validados contra los requerimientos, probados para varias infraestructuras y usados para, de manera directa, simular el comportamiento del sistema que se está diseñando.

Para utilizar MDA uno de los primeros pasos del proceso es modelar los requerimientos del sistema en un CIM, el cual describirá los aspectos relacionados con el uso de la aplicación, de manera que ayuda a presentar de manera precisa qué es lo que se espera que el sistema haga, esto será útil no sólo como ayuda para comprender el problema, sino también como una fuente donde se genera un vocabulario compartido que será utilizado en otros modelos.

Mas tarde un PIM es desarrollado, éste describe el funcionamiento del sistema pero no los detalles de su implementación en una plataforma especifica y puede ser creado para implementar una o más arquitecturas. El arquitecto entonces escoge una plataforma (o varias) que le permitan implementar el sistema con las características arquitecturales definidas, de aquí el arquitecto genera un PSM [10].

Una forma de comprender todo el funcionamiento de MDA y sus elementos principales es observando su metamodelo. La figura 10 expresa el metamodelo de la descripción de MDA, el cual resume todos los elementos que se han mencionado [10]. Se puede apreciar que la base de todo en MDA es la definición de un metamodelo, es decir, un lenguaje de modelado, el cual es utilizado para la creación de PIM's, PSM's y CIM's. El metamodelo de MDA también muestra que para hacer el mapeo entre los distintos modelos es necesario establecer reglas o técnicas de mapeo entre los distintos modelos.

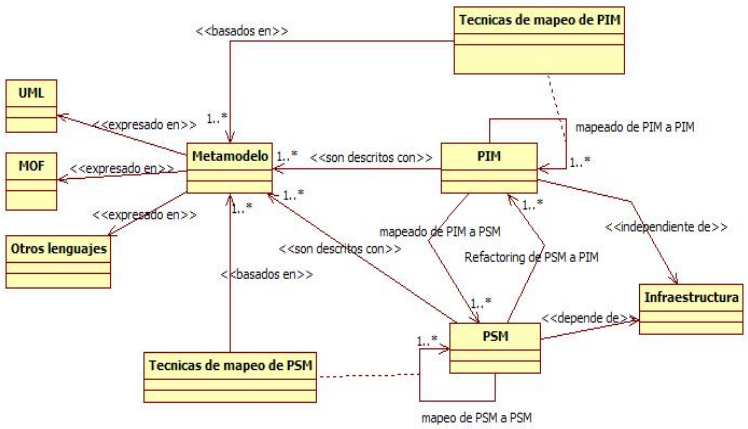

Figura 10: Metamodelo de MDA.

Mapeo y transformación de modelos

El mapeo es un conjunto de reglas y técnicas usadas para modificar un modelo de manera que se pueda generar uno nuevo. El mapeo es utilizado para transformar de [11]:

- **PIM a PIM.** Esta transformación es utilizada cuando los modelos son mejorados, filtrados o especializados durante el ciclo de vida de desarrollo sin necesitar ninguna información dependiente de la plataforma. Una de las formas de mapeo más obvias es entre los modelos de

análisis y el diseño.

- **PIM a PSM.** Esta transformación es utilizada cuando el PIM está lo suficientemente refinado para ser proyectado a una infraestructura de ejecución. La proyección está basada en las características de la plataforma utilizada.
- **PSM a PSM.** Esta transformación puede requerirse en la implementación y realización de componentes. Por ejemplo, el empaquetado de un componente se realiza seleccionando servicios y configuración. Una vez empaquetado, la entrega del componente puede ser realizada especificando los datos de inicialización, servidores de instalación, generación y configuración del contenedor, etc. Esta transformación está ligada al refinamiento de un modelo PSM a un PSM mejorado y más completo.
- **PSM a PIM.** Esta transformación puede ser requerida para abstraer modelos de implementaciones existentes en una tecnología específica a una independiente de la plataforma. Este procedimiento es sin duda uno de los más complicados y difícilmente puede ser automatizado.
- **PSM a Código.** Esta transformación es la última de la cadena y permite generar el código específico para una plataforma en particular utilizando un PSM. Una vez mejorados los PSM o actualizados debido al surgimiento de nuevos requerimientos una transformación de este tipo es necesaria para regenerar el sistema.

Para implementar definir las reglas de transformación de mapeo se requiere conocer los metamodelos de los modelos de entrada y salida. Las reglas de la ejecución de la transformación pueden ser generadas utilizando herramientas UML. Existen muchas maneras de transformar PIMs expresados en UML en su correspondientes PSMs:

1. Una persona puede estudiar el PIM y manualmente construir un PSM y quizás construir o refinar el mapeo entre los dos.
2. Una persona puede estudiar el PIM y utilizar patrones de refinamiento conocidos para reducir la carga en la construcción del PSM y la relación entre ambos.
3. Un algoritmo puede ser aplicado al PIM y crear un esqueleto del PSM, el cual sera mejorado de manera manual, quizás utilizando alguno de los patrones de refinamiento de 2.
4. Un algoritmo puede crear un PSM completo de un PIM detallado. Este deberá de manera implícita o explícita grabar los patrones de refinamiento para ser utilizados por otras herramientas automatizadas.

Una consideración adicional es que es más fácil generar código ejecutable de características estructurales de un modelo que de características de comportamiento. La automatización de transformaciones es mas trazable cuando la transformación está parametrizada de alguna manera, como por ejemplo, cuando una persona selecciona opciones de un conjunto predefinido que determinará como será realizada la transformación.

Generalmente se debe de realizar un proceso de marcado del PIM, en el cual se seleccionarán ciertas características no funcionales que se desea tenga el PSM, características que no pueden ser determinadas con la información que ofrece el PIM. Las marcas en un modelo también pueden especificar la calidad de la implementación, estas podrían requerir parámetros, por ejemplo, una marca que indique “soporte de múltiples conexiones” puede requerir un parámetro que indique el límite máximo de conexiones que aceptará, o alguno que indique políticas de *timeout*. Para que las marcas

sean utilizadas de manera apropiada es recomendable que sean estructuradas, limitadas y modeladas, por ejemplo, un conjunto de marcas mutuamente exclusivas necesitaran ser agrupadas, de manera que el arquitecto sepa que no más de una de estas marcas puede ser aplicada a la transformación de un mismo elemento.

El mapeo también debe incluir plantillas, que son modelos parametrizados que especifican tipos particulares de transformaciones, son como patrones de diseño, pero incluyen especificaciones más detalladas para guiar la transformación. Un conjunto de marcas en un modelo pueden estar asociadas a una plantilla, de manera que éstas marcas indiquen que las instancias de un modelo en particular deberán ser transformadas de acuerdo a ésta platilla. Otras marcas pueden ser utilizadas para llenar parámetros requeridos por una plantilla [10].

Métodos específicos de transformación de modelos

Transformación por marcado. En un modelo son colocadas una serie de marcas que serán utilizadas para guiar la transformación (figura 11) [10]. Una vez que se tiene el modelo marcado, se aplican reglas de mapeo definidas para una plataforma específica para transformar el PIM en un PSM.

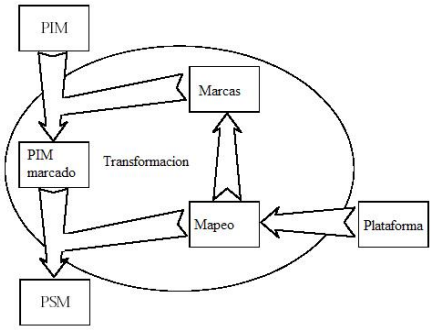

Figura 11: Mapeo por marcado de modelos

Transformación por metamodelo. Un modelo es construido usando un lenguaje independiente de la plataforma especificado por algún metamodelo. Se hace la transformación de este modelo a un lenguaje específico de la plataforma especificado en algún otro metamodelo. Es decir se transforma de un lenguaje de modelado a otro (figura 12) [10]. Se tiene un lenguaje de modelado fuente que será transformado en un lenguaje de modelado objetivo.

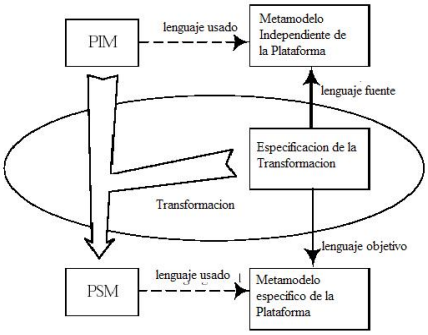

Figura 12: Transformación por medio de metamodelos

Transformación por patrones. Patrones pueden ser utilizados en la especificación del mapeado (figura 13). El mapeado incluirá entonces patrones y marcas correspondientes a elementos de dichos patrones.

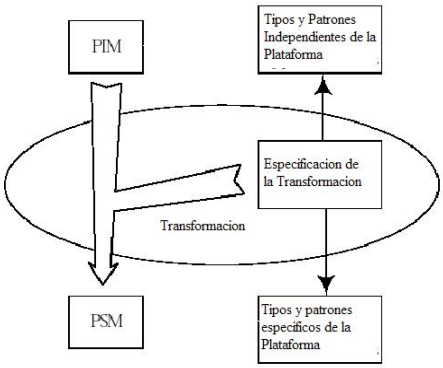

Figura 13: Transformacion por patrones

La figura 14 muestra cómo mediante transformaciones utilizando distintas técnicas puede llevarse a cabo el proceso de convertir un modelo en otro.

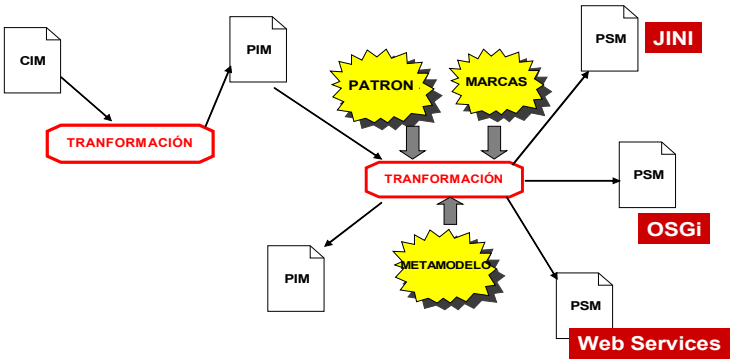

Figura 14: Transformaciones entre distintos modelos

Se debe notar que sea cual sea el método de transformación seleccionado para pasar de un modelo a otro se debe determinar un conjunto de reglas de transformación, en las cuales se indica que elementos del modelo nuevo serán generados tomando como base elementos del modelo original. Así el mapeo consiste en determinar estas reglas de transformación, mapear los elementos de un modelo a los de otro, y de esta forma, facilitar el proceso de tranformación.

2.5.5. Base tecnológica de MDA

MDA esta basado en tecnologías de la OMG (Object Managment Group) que es una organización no lucrativa encargada de definir estándares en el dominio de la orientación a objetos. Estas se describen brevemente a continuación a manera de introducción para comprender los siguientes capítulos [10].

UML

El lenguaje de modelado unificado (UML) es un lenguaje de modelado estándar para la

visualización, especificación y documentación de sistemas de software [10]. Los modelos de MDA pueden ser especificados usando UML. UML resuelve el problema de la arquitectura, objetos e interacciones entre objetos. Los artefactos capturados en UML (en términos de casos de uso, clases, diagramas de actividad, etc.) pueden ser exportados a otras herramientas del proceso de desarrollo usando XMI (XML *Metadata Interchange*).

XMI

XMI (XML *Metadata Interchange*) es un mecanismo para estándar de intercambio de modelos de manera textual siguiendo un formato XML entre distintas herramientas, repositorios y *middleware* [10]. XMI es parte fundamental del mundo del modelado y juega un rol importante en el uso de XML como parte importante de MDA.

Perfiles UML

UML es un lenguaje que nos provee de flexibilidad y amplia expresividad en el modelado de aplicaciones, sin embargo, existen casos en que el uso de UML presenta limitaciones para ciertos dominios donde deseamos crear modelos mas detallados ya que UML no permite expresar conceptos específicos para cierto dominio o plataforma, esto es debido a que UML fue originalmente pensado para ser independiente de la plataforma. Cuando esto sucede se requiere definir un nuevo lenguaje específico para nuestro dominio [14].

La OMG (*Object Managment Group*), organización encargada de definir estándares como UML y MDA nos ofrece dos formas de definir nuevos lenguajes de modelado. O bien se define un nuevo lenguaje (alternativo a UML) o, se extiende UML, especializando ciertos conceptos y restringiendo otros, pero respetando la semántica original de UML (elementos como clases, atributos y métodos se mantienen). Esta forma de extender UML a un dominio en particular se le denomina perfil UML [17].

De forma más precisa, el paquete *Profiles* de UML 2.0 define una serie de mecanismos para extender y adaptar las metaclases de un metamodelo cualquiera (y no sólo el de UML) a las necesidades concretas de una plataforma (como puede ser OSGi o *Web Services*) o de un dominio de aplicación (orientación a servicios, modelado de procesos de negocio, etc.).

UML 2.0 señala varias razones por las que un diseñador puede querer extender y adaptar un metamodelo existente, entre algunas de estas razones están:

1. Disponer de una terminología y vocabulario propio de un dominio de aplicación o de una plataforma de implementación concreta (por ejemplo, poder manejar dentro del modelo terminología propia de OSGi como "*Servicio*", "*Componente*", "*Bundle*", etc.).

2. Definir una nueva notación para símbolos ya existentes, más acorde con el dominio de la aplicación objetivo (poder usar, por ejemplo, una figura con una computadora en lugar del símbolo de un cubo que por defecto ofrece UML para representarlas).

3. Añadir restricciones a las existentes en el metamodelo, restringiendo su forma de utilización (por ejemplo, impidiendo que se conecten dos clases o forzando la existencia de ciertas asociaciones entre las clases de un modelo).

4. Añadir información que puede ser útil a la hora de transformar el modelo a otros modelos, o a código. Importante para nosotros, ya que esto precisamente tenemos que hacer.

Un perfil se define en un paquete UML, estereotipado «profile», que extiende a un metamodelo o a

otro perfil. Tres son los mecanismos que se utilizan para definir perfiles: *estereotipos* (stereotypes), *restricciones* (constraints), y *valores etiquetados* (tagged values). Un estereotipo sirve para definir nuevos elementos de nuestro lenguaje, que generalmente son creados extendiendo las metaclases "Class" (clase) y "Association" (asociación) del metamodelo de UML. Un valor etiquetado es un atributo de un estereotipo. Una restricción nos sirve para validar que el modelo creado a partir de nuestro perfil UML sea semánticamente correcto y siga un conjunto de reglas establecidas. La OMG especifica que las restricciones en un perfil UML deben estar especificadas para cada estereotipo y estar escritas en OCL [40].

OCL (*Object Constraint Language*)

En el modelado orientado a objetos, un modelo como el de clases no es suficiente para lograr una especificación precisa. Puede ser necesario describir características adicionales sobre los objetos del modelo. Muchas veces estas características se describen en lenguaje natural. La práctica ha revelado que muy frecuentemente esto produce ambigüedades. Para escribir especificaciones correctas se han desarrollado los lenguajes formales.

OCL es un lenguaje formal para expresar restricciones libres de efectos colaterales. Los usuarios de UML y de otros lenguajes visuales pueden usar OCL para especificar restricciones y otras expresiones incluidas en sus modelos. OCL tiene características de un lenguaje de expresión, de un lenguaje de modelado y de un lenguaje formal. Es un lenguaje formal, fácil de leer y escribir. Ha sido desarrollado como un lenguaje de modelado para negocios dentro de la división de seguros de IBM.

OCL es un lenguaje de expresión puro. Por lo tanto, garantiza que una expresión OCL no tendrá efectos colaterales; no puede cambiar nada en el modelo. Esto significa que el estado del sistema no cambiará nunca como consecuencia de la evaluación de una expresión OCL. Todos los valores de todos los objetos, incluyendo todos los enlaces, no cambiarán cuando una expresión OCL es evaluada, simplemente devuelve un valor. OCL no es un lenguaje de programación, por lo tanto, no es posible escribir lógica de programa o flujo de control en OCL. No es posible invocar procesos o activar operaciones que no sean consultas en OCL [40].

Ejemplo simple de perfil UML.

Un ejemplo sencillo que puede servir para ilustrar estos conceptos se muestra a continuación, en este ejemplo se definen dos elementos con los cuales pueden construirse un modelo carretero: Ruta y Ciudad.

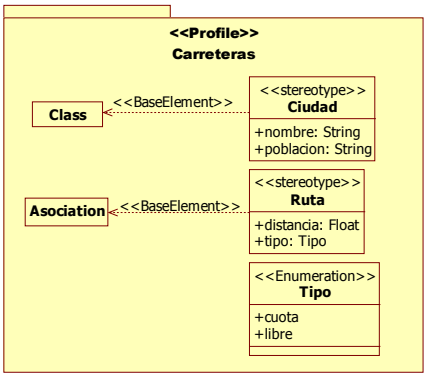

Figura 15: Ejemplo simple de perfil UML

(1) En primer lugar, un estereotipo viene definido por un nombre, y por una serie de elementos del metamodelo sobre los que puede asociarse. Gráficamente, los estereotipos se definen dentro de cajas, etiquetadas «stereotype». En nuestro ejemplo, el perfil UML "Carreteras" define dos estereotipos, Ruta y Ciudad. Tal y como se indica en el perfil, el estereotipo Ruta extenderá a una asociación UML y Ciudad extenderá a una clase UML. Obsérvese cómo el perfil especifica los elementos del metamodelo de UML sobre los que se pueden asociar los estereotipos estereotipados «metaclass».

(2) A los estereotipos es posible asociarles restricciones, que imponen condiciones sobre los elementos del metamodelo que han sido estereotipados. De esta forma pueden describirse, entre otras, las condiciones que ha de verificar un modelo "bien formado" de un sistema en un dominio de aplicación. Por ejemplo, supongamos que el metamodelo de nuestro dominio de aplicación impone la restricción de que dos o mas ciudades no pueden estar conectadas por más de una ruta. Dicha restricción se traduce en la siguiente restricción del Perfil UML, en el lenguaje OCL:

```
context Ciudad
 inv :
        self.connection->select(isStereotyped('Ruta')->size() <= 1
```

Las restricciones pueden expresarse tanto en lenguaje natural como en OCL.

(3) Finalmente, un valor etiquetado es un metaatributo adicional que se asocia a una metaclase del metamodelo extendido por un perfil. Todo valor etiquetado ha de contar con un nombre y un tipo, y se asocia un determinado estereotipo. De esta forma Ciudad tiene dos valores etiquetados que son nombre y población y, Ruta tiene los valores etiquetados distancia y tipo. También se define una enumeración que define los valores que puede tomar el valor etiquetado tipo, que puede ser una ruta de cuota o libre. Los valores etiquetados se representan de forma gráfica como atributos de la clase que define el estereotipo.

Una vez definido el perfil UML se pueden crear modelos utilizando el nuevo lenguaje definido por este, la figura 52 muestra un ejemplo de un modelo generado utilizando los elementos definidos en el perfil.

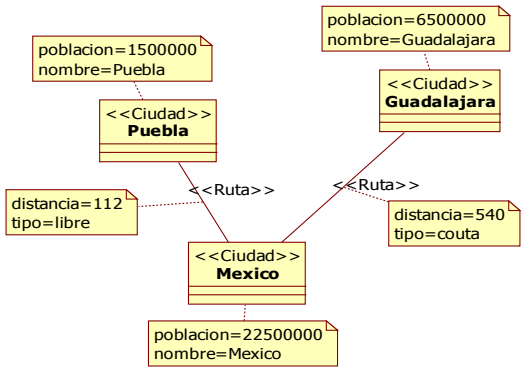

Figura 16: Modelo basado en el perfil UML de ejemplo

2.5.6. Construcción de PIM's y PSM's en UML

El poder de UML (a diferencia de otros lenguajes) radica en que fue definido basándose en los

conceptos más importantes del modelado. Sumado a esto tenemos la ventaja de que los modelos en UML pueden ser representados tanto de forma gráfica como textual utilizando XMI, lo que facilita la transformación entre modelos. Los modelos representados en UML pueden ser muy ricos semánticamente, ya que UML provee elementos para la definición de restricciones y comportamiento tales como [12]:

- Indicar limitaciones sobre un conjunto de atributos.
- Indicar pre y poscondiciones para especificar métodos.
- Indicar si el valor de un parámetro puede ser nulo.
- Indicar si una operación tiene efectos colaterales.
- Indicar patrones de especificaciones y diseño.

Al mismo tiempo como parte de la especificación de UML se define un lenguaje formal de verificación llamado OCL (*Object Constraint Language*) [40] que facilita la especificación de este tipo de restricciones. Especificar las restricciones en un lenguaje formal en lugar de utilizar lenguaje normal permite reducir la ambigüedad de la especificación y por lo tanto facilita la implementación de 3 importantes aspectos:

- Provee al programador de instrucciones más precisas, eliminando la probabilidad de que éste tenga que adivinar el pensamiento del diseñador.
- Disminuye la cantidad de trabajo requerido para hacer que diferentes implementaciones de la misma especificación trabajen juntas, o integrar implementaciones de dos especificaciones cuyos modelos están relacionados.
- Provee la base para definir pruebas de conformidad para diferentes implementaciones.
- Estandariza la especificación de restricciones de manera que herramientas pueden comunicarse entre si siguiendo este lenguaje.

Debido a que UML es un lenguaje diseñado específicamente para ser independiente de la plataforma resulta una solución natural para realizar la construcción de PIM's.

UML está definido para ser independiente de la plataforma, por lo que no resulta obvio como generar un PSM en UML. Sin embargo pensemos en un PIM que deseamos transformar a su correspondiente PSM de CORBA (*Common Object Request Broker Architecture*), para poder realizar este trabajo se requieren tomar algunas decisiones, ¿las clases en UML representan interfaces, uniones, tipos de datos o estructuras de CORBA?, si es así, ¿cómo indico cual de todos es?, si no es así, ¿qué representan las clases y donde serán utilizadas?.

Tales decisiones pueden ser especificadas en un perfil UML usando facilidades definidas en este lenguaje como estereotipos (*stereotypes),* que especifican nuevos elementos del lenguaje, valores etiquetados (*tagged values)* que especifican propiedades de estos elementos y restricciones del modelo (*constraints*) en OCL que especifican las reglas a seguir en la construcción de modelos usando este nuevo lenguaje visual. De esta manera se puede definir un nuevo lenguaje para un dominio en particular definiendo nuevos tipos basándose en los tipos base de UML (clase, atributo, asociación, etc.). Los estereotipos que son utilizados para crear nuevos tipos de elementos, etiquetan un elemento del modelo para indicar que ese elemento sigue una semántica particular. Los valores etiquetados son utilizados para definir los atributos de estos nuevos elementos y las restricciones para validar que los modelos generados sigan reglas definidas para un dominio en particular. La figura 17 muestra un

modelo creado a partir de un perfil UML para CORBA, se especifica mediante el estereotipo <<CORBAInterface>> el tipo de elemento que se define, el cual tiene dos atributos definidos como valores etiquetados y una restricción en OCL que indica los valores que el atributo "número" puede tomar, en este caso indica que éste debe ser un valor entre 1000 y 9000. Mas adelante se detallan los elementos de un perfil UML y el proceso a seguir para la de creación de uno.

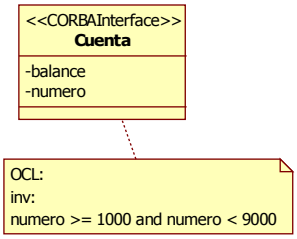

Figura 17: Especificación de una interfaz CORBA usando un perfil UML

2.6. Estado del arte de las herramientas MDA

Aunque las herramientas de transformación son el corazón del desarrollo en MDA, no son las únicas herramientas que se requieren. Algunas de las herramientas necesarias en un ambiente de desarrollo MDA son [15]:

- **Editor de código (IDE).** Las funciones que proveen los ambientes de desarrollo integrado (IDE), como la depuración, compilación y edición de código, no deben pasarse por alto.
- **Repositorio de modelos.** Una base de datos de modelos.
- **Editor de modelos (herramienta CASE).** Donde los modelos pueden ser construidos y modificados.
- **Validador de modelos.** Los modelos usados para la generación de otros modelos deben de estar extremadamente bien definidos. Los validadores revisan los modelos contra un conjunto de reglas (predefinidas o definidas por el usuario) para asegurar que el modelo está listo para ser usado en una transformación.
- **Editor de definición de transformaciones.** Un editor para crear y modificar una definición un de una transformación.
- **Repositorio de definiciones de transformación.** Un lugar de almacenamiento de las definiciones de transformaciones que son utilizadas para pasar de modelo a otro.
- **Ejecución de modelos.** Un motor que provee una plataforma virtual donde se pueden ejecutar los modelos.

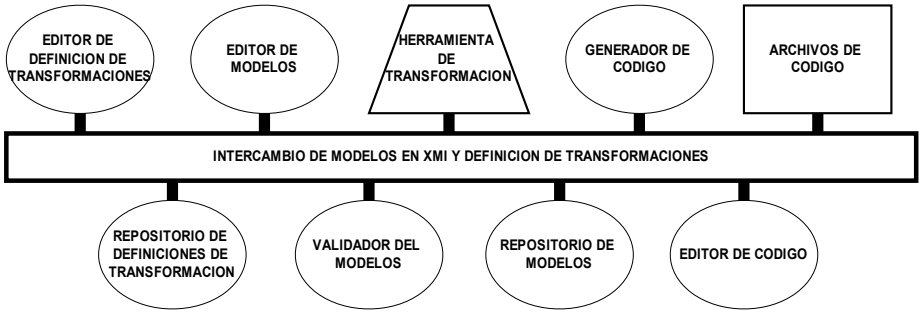

Figura 18: Elementos de un ambiente MDA de desarrollo

Una parte fundamental de las herramientas que soportan MDA debe ser la capacidad de dividir los modelos en CIM, PIM y PSMs, así como el marcado de estos y la definición de las transformaciones, para después generar transformaciones entre modelos y generación de código a partir de ellos (y no solo a la generación de código a partir del diagrama de clases como manejan algunas herramientas CASE) [21].

Existen a la vez dos tipos de herramientas MDA, aquellas que generan el código y aquellas que "ejecutan" el modelo. El código generado del primer grupo de herramientas tiene que ser en la mayoría de los casos modificado, ya que no es código terminado que pueda ser compilado y ejecutado (es más bien un 'esqueleto'), eso sin tomar en cuenta que en la mayoría de los casos los programadores tienden a cambiar la estructura del código generado, de tal forma que el modelo queda desactualizado. Algunas herramientas intentan arreglar este problema mediante lo que se conoce como ingeniería inversa, con la cual es posible volver a actualizar el modelo a partir del código modificado, sin embargo aun así debe de existir disciplina en los cambios para evitar desincronización entre modelos.

El segundo tipo de herramientas utiliza UML y algunas extensiones del mismo para modelar a detalle una aplicación de tal forma que el modelo pueda ser ejecutado dentro de la herramienta. Algunas de estas herramientas generan código como Java, C++ o algún lenguaje propio y generan una serie de paquetes que pueden ser ejecutados dentro del servidor de la misma aplicación. El principal problema de estas herramientas es que el generar un modelo UML que pueda ser ejecutado no es cosa sencilla, uno debe de "hablar" UML de manera fluida para obtener beneficios reales de una herramienta de este tipo [16]. A continuación se presentan un conjunto de herramientas encontradas durante el proceso de investigación para la escritura de esta tesis.

2.6.1. Comparación de distintas herramientas MDA disponibles en el mercado

La siguiente tabla muestra un comparativo de herramientas MDA encontradas en el mercado, los puntos que se compararon en cada herramienta tienen que ver con aspectos importantes que debe ofrecer una herramienta de desarrollo MDA para considerarse como MDA, además de otros aspectos importantes como su extensibilidad y facilidad de uso. Las características consideradas para la evaluación se presentan a continuación, éstas fueron definidas en base a las características que define la especificación de MDA [15] además de aspectos importantes relacionados con un enfoque SOA:

a) **PSM a código de cualquier plataforma.** Se revisó que la herramienta fuera capaz de generar código para cualquier plataforma, sin estar atada a una plataforma en particular, esto es porque un aspecto importante de MDA es que, cuando surgen nuevas tecnologías, la herramienta debe adaptarse a estas de forma natural.

b) **PIM a PSM.** Es importante que la herramienta cuente con una forma de transformar modelos independientes de la plataforma a modelos en distintas plataformas, de manera que teniendo un solo PIM se pueda regenerar una aplicación una y otra vez.

c) **Editor de metamodelos.** Un aspecto básico de MDA es que los modelos que se generan deben de alguna forma ser validados con respecto a una serie de reglas que se establezcan para cada dominio en particular. La única manera de realizar este proceso de validación es mediante la definición de un metamodelo. Esto puede hacerse vía un editor visual o textual.

d) **Editor de reglas de transformación.** La herramienta deberá proveer una manera simple de definir qué elementos de un modelo corresponderán a qué elementos de otro modelo, de manera que se pueda realizar la transformación entre uno y otro. Esto puede hacerse vía un editor visual o textual.

e) **Facilidad adaptación a un dominio en particular.** La herramienta debe de permitir de manera simple adaptar metamodelos, reglas de transformación y código generado para cualquier dominio o plataforma existente o no existente aún.

f) **Comercial.** Es importante que la herramienta no tenga un precio excesivo, de manera que pueda ser accesible a la mayoría de los proyectos de desarrollo de software.

g) **Dificultad de uso y aprendizaje.** Un problema con la mayor parte de las herramientas en el mercado es que son difíciles de utilizar y comenzar a sacarles provecho puede llegar a ser una tarea complicada. Esto provoca que las herramientas y los procesos MDA sean rechazados por la mayor parte de los proyectos de desarrollo.

Aplicación	PSM a Código de cualquier plataforma	PIM a PSM	Editor de metamodelos	Editor de reglas de transformación	Fácil adaptación a un dominio en particular	Comercial	Dificultad de uso y aprendizaje
UVM-QVT	NO	SI	SI	SI	NO	SI	Alta
OpenMDX	NO	SI	SI	SI	NO	SI	Alta
AndroMDA	SI	SI	SI	SI	NO	SI	Alta
EMF	NO	NO	SI	SI	NO	NO	Media
Acceleo	SI	NO	NO	SI	SI	NO	Baja

Tabla 1: Comparación de las distintas herramientas MDA en el mercado

Este análisis junto con la comprensión del estado del arte es determinante en el entendimiento del los objetivos y justificación del proyecto, ya que lo que se realizó durante este proyecto fue la construcción de una herramienta cubriera el mayor número posible de características de MDA, pero sobre todo una de las características que casi ninguna herramienta tiene: la fácil adaptación a un dominio en particular.

2.7. Comentarios finales

Se llevó a cabo una investigación sobre todos los temas directamente relacionados con nuestro proyecto por dos importantes razones:

1. Conocer en detalle la terminología y entender los conceptos relacionados con SOA, MDA y

componentes orientados a servicios.

2. Conocer el estado actual de SOA y MDA, así como de las herramientas que las implementan de manera que se pudiera hacer una propuesta novedosa en este proyecto.

Se decidió este capítulo al inicio de este trabajo, para que el lector pudiera conocer toda la terminología que se utilizará posteriormente y, al mismo tiempo, tener una referencia sobre donde se encuentra nuestro proyecto con respecto a otros proyectos en la misma rama de investigación de manera que se pueda juzgar de manera mas precisa las contribuciones del mismo.

Ahora se tienen las bases para comenzar a presentar el planteamiento del problema que se abordó en este proyecto, la justificación del mismo y la estructura que se definió para este proyecto.

3. Planteamiento del problema y estructura del proyecto

3.1. Antecedentes

El proyecto descrito en este documento surgió como continuación de los trabajos derivados del *Service Binder* [38]. Se relacionó con la construcción de aplicaciones ensambladas a partir de componentes orientados a servicios utilizando una estructura declarativa. Como se mencionó anteriormente este enfoque tuvo tanto éxito que fue adoptado como parte de la especificación de OSGi en su *release* 4 como servicios declarativos [36]. Para poder expresar los objetivos de este proyecto en los términos correctos como primera parte de este capítulo se hace una descripción detallada del funcionamiento de los servicios declarativos de OSGi y sus problemáticas actuales, de manera que, posteriormente pueda hacerse una presentación detallada de los objetivos generales y específicos para este proyecto.

3.1.1. Los servicios declarativos de OSGi

El *framework* de OSGi contiene un modelo para la publicación, descubrimiento y conexión de servicios. Permite la creación de aplicaciones mediante *bundles* que se comunican a través de estos servicios. Adicionalmente el *framework* permite la implantación de *bundles* durante la ejecución, es decir, actividades como el registro y retiro de servicios pueden realizarse durante la ejecución de la aplicación, lo cual provoca que el entorno de OSGi pueda ser altamente dinámico. Existen muchos aspectos a considerar cuando se habla de aplicaciones de este tipo, donde la dinámica de servicios es muy compleja, tales como:

- **Desperdicio de memoria en el registro de servicios.** Esto se debe a que puede suceder que existan servicios que no sean utilizados.
- **Programación de la dinámica de los servicios.** En un ambiente tan dinámico, los servicios pueden partir o llegar en cualquier momento. Por ello, es necesario que los consumidores de dichos servicios sean capaces de adaptarse con respecto a estos cambios. La programación de la lógica que permite soportar la adaptación dinámica puede resultar complicada.
- **Registro de los servicios.** Se deben programar todos los aspectos relacionados con el registro de los servicios en un repositorio común.
- **Descubrimiento y ligado de los servicios a los que se hace referencia.** Cuando se hace referencia a un servicio se debe programar el código para realizar el descubrimiento de los estos servicios y el ligado con ellos.
- **Programación de la lógica del negocio.** Además se tiene que pensar en la programación de la lógica del negocio. Esto puede llegar a pasar a segundo plano en algunos casos donde la complejidad de la arquitectura de la aplicación es la preocupación principal de los desarrolladores, olvidando los requerimientos funcionales de la aplicación.

Con excepción del último punto, todos los demás son aspectos no funcionales de la aplicación, ésto es, están directamente relacionados con la arquitectura de la misma, y no con los aspectos de la lógica del negocio de la aplicación. En OSGi estándar, el desarrollador debe encargarse de implementar toda la lógica de soporte de estos aspectos no funcionales además de la lógica de negocio. En algunos casos,

la programación del primer tipo de lógica puede tomar más tiempo y ser más complicado que la de la segunda.

La figura 19 muestra un fragmento de código de un *bundle* muy simple de OSGi en el cual se presentan fragmentos correspondientes a cada uno de los puntos a desarrollar antes mencionados. Se puede apreciar que la parte correspondiente a la lógica de negocio resulta pequeña en comparación con lo demás.

Figura 19: Fragmento de código de un bundle OSGi y sus problemáticas

Por tanto la programación de *bundles* de la forma tradicional es un reto técnico importante. Una solución a este problema consiste en extraer el manejo de los aspectos no funcionales del código de los *bundles* y utilizar un servicio del entorno de ejecución que se encargue manejar dichos aspectos. La extracción de la lógica no funcional se logra mediante una descripción declarativa en un archivo XML. Esta descripción contiene información acerca de la forma en que se debe manejar el dinamismo soportado por el entorno de ejecución. Este entorno, llamado *Service Component Runtime,* es instalado como un *bundle* y tiene la función de ofrecer un componente que maneje todos los aspectos relacionados con la publicación, descubrimiento y ligado de servicios [36].

El modelo declarativo simplifica la creación de servicios OSGi realizando la tarea de registro de servicios y manejando la administración de sus dependencias. Esto minimiza la cantidad de código que el programador debe realizar, ya que únicamente debe escribir la lógica de negocio y el archivo XML (ver figura 20), además de proveer un mecanismo mediante el cual los servicios sólo son cargados cuando se requieren.

Figura 20: Descriptor de componente en XML y código de la lógica de negocio

Entidades que componen el modelo de los servicios declarativos

Dentro de la especificación de los servicios declarativos se manejan múltiples conceptos y restricciones, de tal forma que, cuando un desarrollador quiere seguir este enfoque, tiene que comprender y conocer todos estos conceptos y restricciones para poder utilizarlos de forma correcta. A continuación se presenta un resumen de algunos de estos conceptos [36].

Componente del servicio. Contiene un descriptor que será interpretado en tiempo de ejecución para crear y liberar objetos. Se utiliza el termino componente para referirse a un componente orientado a servicios. Es una clase común en Java dentro de un *bundle*. Un componente puede tener un cierto numero de dependencias hacia otros servicios que tienen que ser satisfechas antes de que el componente pueda ser activado y que a su vez pueda proveer sus propios servicios. Un componente requiere de los siguientes elementos dentro del *bundle* (1) Un documento XML que contiene el descriptor del componente, (2) el archivo *manifest* que da la localización del documento XML que contiene las descripciones del componente y (3) la clase que implementa el servicio descrito en el descriptor del componente y que contiene únicamente lógica de negocio.

Descriptor del componente. Contenido dentro de un documento XML en el bundle, especifica cómo los servicios declarativos administrarán el registro, descubrimiento y ligado de servicios, además de describir el comportamiento del componente.

Propiedades del componente. Un conjunto de propiedades que pueden ser especificadas en el descriptor del servicio que indican cómo los servicios declarativos manejarán el dinamismo del componente.

Configuración del componente. Representa la descripción parametrizada de cada propiedad, es la entidad que rastrea las dependencias del componente y administra la instancia del mismo. Una configuración de componente activa tiene asociado un contexto del componente. Cada instancia de componente recibe un contexto único.

Referencia. Es la dependencia de un componente de un conjunto de servicios. Estas son adquiridas mediante el registro de servicios. Una configuración de componente se considera satisfecha cuando cada referencia especificada está satisfecha. Una referencia es satisfecha si especifica cardinalidad

opcional (no requiere necesariamente de algún servicio para funcionar) o cuando los servicios referenciados existen con al menos un elemento.

Sin embargo debido a la dinámica en el registro de servicios de OSGi, puede suceder que los servicios referenciados sean modificados o eliminados una vez que ya están ligados a otros componentes, provocando que, las funciones que nos ofrecía se vuelvan obsoletas. Existen dos **políticas** que los componentes pueden especificar para tratar con estas situaciones.

La política *estática* es la mas simple, en ésta, una instancia del componente nunca ve ningún cambio. La configuración del componente es desactivada antes de que cualquier servicio ligado por una referencia con política estática desaparezca. Una vez que un servicio referenciado está disponible para remplazar al servicio que desapareció, la configuración del componente es reactivada y ligada al servicio de repuesto. Este tipo de referencia puede llegar a ser muy costoso si se depende de servicios que están constantemente registrándose y desregistrándose o si la activación o desactivación de la configuración del componente tarda demasiado.

La otra política de ligado es la *dinámica*, con esta política se puede cambiar el conjunto de servicios ligados sin desactivar o activar la configuración del componente.

Service Component Runtime (SCR). Es el componente del entorno de ejecución encargado de administrar los componentes y su ciclo de vida. Este lee descriptores de componentes de los *bundles* ya activados. El SCR basa sus decisiones en la información de la descripción del servicio, con información básica del componente como su nombre y tipo, servicios implementados por el componente y referencias a otros componentes. El SCR debe de activar la configuración de un componente cuando el componente es habilitado y la configuración del componente ha sido satisfecha. Durante la activación de una configuración de componente, el SCR debe ligar algunos o todos los servicios referenciados hacia la configuración del servicio. Durante el tiempo de vida de una configuración de componente el SCR puede notificar al componente de cambios en sus referencias.

El SCR desactivará una configuración de componente previamente activada cuando el componente es deshabilitado, la configuración del componente se vuelve insatisfecha o la configuración del componente ya no es requerida.

Instancia del componente. Es la instancia de la clase del componente. Dicha instancia es creada cuando la configuración del componente es activada y es eliminada cuando la configuración del componente es desactivada.

Una instancia de componente debe ser capaz de usar los servicios que son referenciados por la configuración del componente, es decir los servicios ligados. Existen dos estrategias para que una instancia del componente pueda adquirir estos servicios:

- Eventos. El SCR ejecuta un método de la instancia del componente cuando un servicio es ligado y otro método cuando sucede lo contrario. Estos métodos (llamados *bind* y *unbind*) deberán estar especificados en la referencia. Esta estrategia es útil cuando la instancia requiere ser notificada de cambios en las ligas a servicios de manera dinámica.
- Descubrimiento. Una instancia del componente puede hacer uso de alguno de los métodos específicamente definidos en un objeto que almacena el contexto de la configuración del componente para localizar un servicio ligado.

Las instancias de componente nunca son reutilizadas. Cada vez que una configuración del componente es activada, el SCR debe crear una nueva instancia a ser usada junto con la configuración

del componente activado. Una vez que la configuración del componente es desactivada o falla su activación, el SCR debe descartar todas las referencias a las instancias del componente asociadas a la activación.

Componente tardío (*Delayed*). Es un componente cuya configuración sólo es activada cuando el componente es requerido. Una vez que la configuración del componente es satisfecha, el SCR debe registrarlo en el servicio de registro pero su activación es retrasada hasta que el servicio sea requerido.

Componente inmediato (*Immediate*). Es un componente cuya configuración es activada inmediatamente en cuanto sus dependencias están satisfechas. Si un componente inmediato no tiene dependencias es activado de inmediato. Un componente es inmediato si no es un componente fabricado o no especifica un servicio.

Componente fabricado (*Factory*). En un componente cuya configuración es creada y activada por un llamado componente fábrica que crea configuraciones del componente según la demanda del servicio. Este patrón requiere de la implementación de lo que se conoce como un patrón fabrica. Nuevas configuraciones de componente puedes ser instanciadas ejecutando el método definido con este fin en el servicio de fábrica de componentes y un nuevo servicio será registrado cada vez que una nueva configuración del componente sea creada utilizando el método.

Cardinalidad. La cardinalidad representa dos conceptos importantes:

- Multiplicidad. Indica si la implementación del componente asumirá un solo servicio o manejara múltiples ocurrencias.
- Opcionalidad. Que indicará si el componente puede funcionar sin ninguna liga a algún servicio.

La cardinalidad de una referencia puede ser especificada como una de las siguientes cuatro:

- 0..1 - Opcional o uno solo.
- 1..1 - Obligatoriamente uno.
- 0..n - Opcional o múltiple.
- 1..n - Mandatoria y múltiple.

Estos valores indican los parámetros de satisfacibilidad para una configuración de componente, por ejemplo, en una cardinalidad 1..1 debe existir a lo más un servicio ligado para cada referencia y al menos uno. Si la referencia se vuelve insatisfecha entonces la configuración del componente debe ser desactivada.

Descriptor del componente

El esquema XML para la creación de este descriptor se muestra a continuación. Éste esquema es importante ya que contiene la mayor parte de los elementos del modelo de los servicios declarativos, y fue utilizado como base para comenzar el desarrollo del proyecto. Se han agregado comentarios intentando hacer mas clara cada parte del descriptor.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.osgi.org/xmlns/scr/v1.0.0"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.0.0">
      <!-- COMPONENTE DE SERVICIO -->
      <element name="component" type="scr:Tcomponent"/>
      <complexType name="Tcomponent">
              <sequence>
                <!-- CLASE DE IMPLEMENTACION DEL SERVICIO UNA POR COMPONENTE-->
                <element name="implementation" type="scr:Timplementation" minOccurs="1" maxOccurs="1"/>
                   <!-- PROPIEDADES DEL SERVICIO PUEDEN SER CERO O VARIAS POR COMPONENTE-->
```

```
            <choice minOccurs="0" maxOccurs="unbounded">
               <element name="property" type="scr:Tproperty"/>
               <element name="properties" type="scr:Tproperties"/>
            </choice>
            <!-- NOMBRE DEL SERVICIO OFRECIDO ES CERO O UNO POR COMPONENTE-->
            <element name="service" type="scr:Tservice" minOccurs="0" maxOccurs="1"/>
            <!-- NOMBRE DE LOS SERVICIOS REFERENCIADOS PUEDEN SER CERO O VARIOS POR COMPONENTE-->
           <element name="reference" type="scr:Treference" minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
        <!-- ¿UN COMPONENTE ESTARA HABILITADO AL INICIO? -->
        <attribute name="enabled" type="boolean" default="true" use="optional"/>
        <!-- NOMBRE DEL COMPONENTE DE SERVICIO -->
        <attribute name="name" type="token" use="required"/>
        <!-- ¿UN COMPONENTE ES DE TIPO FACTORY? -->
        <attribute name="factory" type="string" use="optional"/>
        <!-- ¿UN COMPONENTE ES DE TIPO IMMEDIATE? -->
        <attribute name="immediate" type="boolean" use="optional"/>
</complexType>
<!-- NOMBRE DE LA CLASE DE IMPLEMENTACION DEL COMPONENTE -->
<complexType name="Timplementation">
        <attribute name="class" type="token" use="required"/>
</complexType>
<!-- ELEMENTOS DE UNA PROPIEDAD -->
<complexType name="Tproperty">
    <simpleContent>
        <extension base="string">
            <attribute name="name" type="string" use="required"/>
            <attribute name="value" type="string" use="optional"/>
            <attribute name="type" type="scr:TjavaTypes" default="String" use="optional"/>
        </extension>
    </simpleContent>
</complexType>
<complexType name="Tproperties">
        <attribute name="entry" type="string" use="required"/>
</complexType>
<!-- SERVICIO QUE SE PROVEE -->
<complexType name="Tservice">
        <sequence>
                <element name="provide" type="scr:Tprovide" minOccurs="1" maxOccurs="unbounded"/>
        </sequence>
</complexType>
<!-- INTERFAZ QUE PROVEE EL COMPONENTE PARA OFRECER SU SERVICIO -->
<complexType name="Tprovide">
        <attribute name="interface" type="token" use="required"/>
</complexType>
<!-- REFERENCIAS DE UN COMPONENTE DE SERVICIO -->
<complexType name="Treference">
        <attribute name="name" type="NMTOKEN" use="required"/>
         <!-- INTERFAZ QUE OFRECE LA REFERENCIA -->
        <attribute name="interface" type="token" use="required"/>
         <!-- CARDINALIDAD ENTRE COMPONENTE Y REFERENCIA -->
        <attribute name="cardinality" type="scr:Tcardinality" default="1..1" use="optional"/>
         <!-- POLITICA DE LIGADO -->
        <attribute name="policy" type="scr:Tpolicy" default="static" use="optional"/>
         <!-- METODO DE EJECUCIÓN EN EL LIGADO -->
         <attribute name="bind" type="token" use="optional"/>
         <!-- METODO DE EJECUCIÓN EN LA LIBERACION -->
        <attribute name="unbind" type="token" use="optional"/>
</complexType>
 <!-- TIPOS DE DATOS SOPORTADOS EN LAS PROPIEDADES DE UN COMPONENTE -->
<simpleType name="TjavaTypes">
        <restriction base="string">
                <enumeration value="String"/>
                <enumeration value="Long"/>
                <enumeration value="Double"/>
                <enumeration value="Float"/>
                <enumeration value="Integer"/>
                <enumeration value="Byte"/>
                <enumeration value="Char"/>
                <enumeration value="Boolean"/>
                <enumeration value="Short"/>
```

```
        </restriction>
  </simpleType>
   <!-- TIPO DE CARDINALIDAD -->
  <simpleType name="Tcardinality">
        <restriction base="string">
                <enumeration value="0..1"/>
                <enumeration value="0..n"/>
                <enumeration value="1..1"/>
                <enumeration value="1..n"/>
        </restriction>
  </simpleType>
   <!-- TIPO DE POLITICAS -->
  <simpleType name="Tpolicy">
        <restriction base="string">
                <enumeration value="static"/>
                <enumeration value="dynamic"/>
        </restriction>
  </simpleType>
</schema>
```

3.1.2. Problemas en el desarrollo de bundles siguiendo el enfoque declarativo

El modelo declarativo simplifica mucho el trabajo de los desarrolladores de *bundles* para OSGi. Sin este apoyo los desarrolladores tendrían que lidiar con todos los aspectos relacionados con el manejo del dinamismo adicional a los aspectos funcionales de su aplicación.

Sin embargo, utilizando el enfoque declarativo, el desarrollador debe aún tener mucho cuidado en la creación de sus descriptores de componente, debido a que los descriptores deben de mapear directamente al código que implementa el componente y, además el descriptor debe de respetar un conjunto de restricciones y reglas, descritas en la especificación de los servicios declarativos [35]. En éste sentido, uno de los problemas del enfoque declarativo es que los errores generalmente no son detectados por el compilador, de hecho, un *bundle* con un descriptor que contiene errores puede ser instalado dentro del *framework* de OSGi y no es sino hasta tiempo de ejecución cuando el desarrollador se da cuenta de los problemas en la construcción de su descriptor, lo que provoca que el proceso de detección y corrección de los errores pueda llegar a ser tedioso.

La figura 21 muestra un pequeño fragmento de dicha especificación donde se habla de componentes tardíos. La especificación indica que si un componente no está especificado como fabricado y no tiene el atributo *immediate* igual a verdad entonces por defecto se trata de un componente tardío. Como ésta, existen muchas reglas y restricciones descritas informalmente dentro de la especificación y un desarrollador que desee utilizar este enfoque debe conocerlas todas. La complejidad de los descriptores puede aumentar si el modelo de componentes es modificado, llevando al desarrollador a un constante aprendizaje sobre la construcción de descriptores.

Delayed Component

A *delayed component* specifies a service, is not specified to be a factory component and does not have the immediate attribute of the component element set to true. If a delayed component configuration is satisfied, SCR must register the component configuration as a service in the service registry but the activation of the component configuration is delayed until the registered service is requested. The registered service of a delayed component look like

Figura 21: Fragmento de la especificación de OSGi R4

Este proyecto plantea una solución a este problema mediante una herramienta MDA que permita al desarrollador modelar sus componentes de manera visual. La herramienta integrará todas las reglas y restricciones en un ambiente de desarrollo donde el desarrollador será capaz de generar el código de su descriptor de componente y los esqueletos del archivo *manifest* y las clases de implementación del servicios de forma automática a partir de su modelo, dejando la validación de las reglas y restricciones a la herramienta. En las siguiente sección se plantea con mayor detalle la organización del proyecto y los objetivos del mismo.

3.2. Planteamiento y estructura del proyecto

Este proyecto formó parte de un proyecto más grande llamado *Service Binder NG*, cuya motivación principal son las necesidades de los industriales participantes en el proyecto, que son, France Telecom Recherche et Développement (FTR&D) y Schneider Electric.

El proyecto estuvo dividido en dos ejes principales: Eje infraestructura y eje desarrollo.

El eje de infraestructura está encargado de introducir mejoras al *Service Binder* [38] haciendo una implementación de la especificación de los servicios declarativos de OSGi, además de implementar soporte a dos nuevos tipos de conexión entre *bundles*, el servicio *WireAdmin*, basado en el enfoque productor consumidor, y el servicio *Event*, basado en la publicación y suscripción de eventos asíncronos.

El eje de desarrollo estuvo encargado de desarrollar herramientas que faciliten la construcción de aplicaciones que utilicen la infraestructura del primer eje, éstas herramientas siguen un enfoque moderno que privilegia el modelado y la transformación de modelos (MDA). Es en este eje donde se ubicó el proyecto "Un enfoque MDA para el desarrollo de componentes orientados a servicios" presentado en éste trabajo.

El proyecto tenía entonces como objetivo principal construir una herramienta MDA de apoyo para los desarrolladores de *bundles* bajo la plataforma OSGi que basan su desarrollo el modelo de los servicios declarativos. Como segundo objetivo se deseaba definir un proceso de desarrollo que pudiera ser fácilmente adoptado para la creación de otras herramientas MDA. Esto es debido a que se utilizará este proceso en la creación de nuevas herramientas para apoyar a los desarrolladores en otros temas dentro del eje de infraestructura del proyecto *Service Binder NG.*

Se planteó un tiempo de desarrollo de 3 trimestres, tiempo en el cual tenía que tenerse un resultado palpable, tenía que desarrollarse un lenguaje de modelado específico para nuestro dominio y posteriormente implementarlo en alguna herramienta de software, por lo cual, se decidió hacer el proceso de desarrollo lo mas sencillo posible, utilizando *frameworks* como soporte para la construcción de la herramienta y siguiendo un enfoque que pudiera ser fácilmente replicable mediante el uso de estándares ampliamente conocidos como UML.

3.3. Objetivos generales y objetivos particulares del proyecto

3.3.1. Objetivos generales

Los objetivos generales de este proyecto son:

a) **Propuesta de un vocabulario visual de modelado.** Uno de los objetivos de este proyecto era desarrollar un vocabulario que permitiera modelar una aplicación construida a partir de componentes

orientados a servicios de manera visual. Este vocabulario sería obtenido mediante la creación de un perfil UML específico al modelo de componentes orientados a servicios. Con este lenguaje se podrán hacer las descripciones de aplicaciones que sigan este modelo, y además, ser utilizado por alguna herramienta para por ejemplo, automatizar la implementación de los *bundles*.

b) **Definición de un proceso replicable de desarrollo de herramientas MDA.** Para poder cumplir con los objetivos planteados dentro del proyecto se debe de definir un proceso simple para la creación de una herramienta MDA. Éste debe favorecer el uso de estándares ampliamente aceptados y de *frameworks* de desarrollo que aceleren el proceso. Además se deseaba que éste pudiera ser fácilmente replicable por cualquier organización de desarrollo de software para crear sus propias herramientas MDA de acuerdo a sus propias necesidades y su propio dominio.

c) **Creación de herramientas de desarrollo.** Además de los aspectos de modelado, en el proyecto se construyeron herramientas que facilitan el desarrollo de aplicaciones basadas en el lenguaje de modelado de componentes y el proceso de desarrollo propuestos. El proyecto buscó favorecer el desarrollo de *plugins* para la plataforma de Eclipse.

3.3.2. Objetivos particulares

Perfil UML para los servicios declarativos.

Se debió definir y especificar un perfil UML para los servicios declarativos siguiendo los estándares propuestos por la OMG, un perfil UML que pueda ser utilizado para modelar aplicaciones basadas en componentes orientadas a servicios utilizando el modelo propuesto por los servicios declarativos de OSGi.

***Plugin* de Eclipse para modelado de aplicaciones basadas en componentes**

Utilizando el perfil UML definido se construyó un editor visual en el cual se pueden modelar *bundles* de OSGi siguiendo el enfoque declarativo. El desarrollador de bundles puede utilizar la herramienta y, mediante una interfaz gráfica generar los modelos que describan su *bundle*, conectando componentes con los servicios que serán ofrecidos y referenciados por éstos.

Un aspecto importante de este editor gráfico de modelos es que es capaz de validar los modelos creados con él, verificando que se sigan las reglas que se detallan en la especificación del perfil UML para los servicios declarativos. La solución propuesta es fácilmente extensible, de manera que nuevos editores visuales pueden crearse para otras plataformas orientadas a servicios y a dominios fuera de SOA. En este caso se establecieron en un inicio los siguientes requerimientos:

Descripción	Detalle
Agregar elementos tipo clase al modelo	El editor deberá de permitir agregar las clases necesarias al dibujo distinguiéndolas de acuerdo a los estereotipos definidos en el perfil UML.
Conectar elementos tipo clase	El editor deberá de permitir conectar clases por medio de relaciones de acuerdo a las reglas establecidas en el perfil UML.
Establecer las propiedades de las clases	El editor deberá de permitir establecer las propiedades de cada clase de acuerdo a los valores etiquetados establecidos en el perfil UML.
Validación de restricciones	El editor deberá de validar que el modelo cumpla con las restricciones en OCL establecidas dentro del perfil UML.
Editar elementos del modelo	El editor deberá de permitir cambiar las propiedades de los elementos en el dibujo, borrarlos o moverlos dentro del área de dibujo.
Seguimiento de estándares de modelado	El editor deberá seguir estándares definidos por la OMG para el modelado tales como MDA, OCL, XMI y perfiles UML.

Tabla 2: Requerimientos del editor visual

Generación de código a partir del modelo

La herramienta provee una forma de generar código a partir de los modelos creados en el editor gráfico, además de permitir la personalización y adaptación del código generado hacia distintas plataformas diferentes de OSGi. En esta herramienta se definen las reglas de transformación entre el modelo y el código, mapeando elementos del modelo a elementos del código.

El generador de código es ser capaz de generar el descriptor de componente en XML que sigue las especificaciones del perfil UML para los servicios declarativos además de generar los esqueletos de los *bundles* en Java y los archivos de configuración del mismo. Adicionalmente, es relativamente fácil configurar la herramienta para modificar el código que será generado o adaptar la herramienta a otros dominios distintos de SOA.

El *plugin* deberá de generar el código de:

Descripción	Detalle
Descriptor del componente.	El editor deberá de permitir la generación del código XML con el descriptor del componente de acuerdo a los parámetros y elementos establecidos en el modelo para cada uno de los componentes definidos en el mismo. Este descriptor deberá seguir los estándares establecidos en el esquema XML que se encuentra en la especificación de OSGi dentro del capítulo de los servicios declarativos [36].
Esqueleto del código Java de la interfaz del servicio del bundle y la clase de implementación del servicio.	El editor deberá de permitir la generación del código Java de los esqueletos de la interfaz y la clase del servicio proveído por el componente que serán desplegados en el bundle definidos en el modelo. Este código deberá seguir los estándares establecidos en el estándar de codificación de Java de Sun.
Archivo META-INF de configuración del bundle.	El editor deberá permitir la generación del archivo META-INF de configuración del bundle contenido en un archivo jar.

Tabla 3: Requerimientos de generación de código

3.4. Metodología y proceso de desarrollo

Una parte importante de este proyecto es la definición de un proceso de desarrollo de herramientas MDA que sea simple de seguir y fácil de adaptar a cualquier dominio. Como punto de partida fue establecido un proceso basado en el proceso unificado (RUP – *Rational Unified Process*) [39],

adecuándolo a las necesidades del proyecto, esto era debido a que no era un proyecto de desarrollo de software tradicional, sino un proyecto de investigación.

Como parte del proceso unificado se realizaron tareas cuyo objetivo fue el control de los riesgos del proyecto y los artefactos generados a lo largo del mismo. Se dividió el proyecto en cuatro fases principales: Concepción, Elaboración, Construcción y Transición, cada una de estas fases subdividida en iteraciones de 2 semanas, véase figura 22.

Durante la fase de concepción se plantearon los alcances y objetivos del proyecto, se planteó la visión del proyecto y se especificaron los requerimientos de la herramienta que sería desarrollada.

Durante la fase de elaboración se creó el perfil UML para los servicios declarativos y se realizó el diseño de la aplicación siguiendo el proceso unificado, que para éste proyecto en particular consistió en una serie de modelos que fueron utilizados por herramientas de desarrollo para generar la aplicación. El desarrollo mismo de la herramienta MDA se convirtió en un desarrollo MDA, y el enfoque cambió por completo del propuesto en el proceso unificado.

Debido a que una parte importante de nuestro proyecto fue la definición de un proceso de desarrollo que pueda ser utilizado por cualquier organización de desarrollo de software para crear herramientas MDA, era importante descubrir herramientas que facilitaran el desarrollo de nuestro producto. Así, la fase de construcción se dividió en cuatro actividades, primero se realizó una investigación sobre *frameworks* de desarrollo que facilitaran el desarrollo del editor visual. La segunda actividad consistió en crear el editor visual utilizando el *framework* encontrado. La tercera fue la investigación de un *framework* que nos facilitará implementar la generación de código a partir de nuestros modelos, y la cuarta actividad se refiere a la implementación de la generación de código basándonos en este *framework*. La última parte de la construcción de la herramienta es la validación de la misma mediante pruebas sobre su funcionamiento.

La última fase del proceso unificado, transición, consistió en definir como será entregada y distribuida la aplicación a los usuarios finales, además del desarrollo de un manual de usuario y redacción de un artículo para comunicación de los resultados..

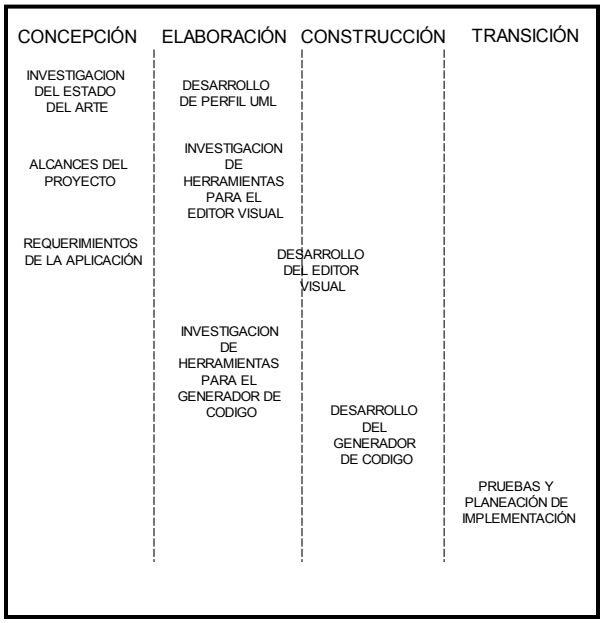

Figura 22: Proceso de desarrollo definido para la construcción de la herramienta MDA

4. Creación del lenguaje de modelado para los servicios declarativos

Se siguió un estándar de modelado que nos permite realizar las transformaciones entre las distintas representaciones (PIM y PSM). Existen distintas formas de realizar la transformación entre modelos, una forma de hacerlo es mediante el uso de un metamodelo, es decir, definir un lenguaje de modelado propio. Una forma de construir un metamodelo es mediante un perfil UML.

Entonces el proyecto fue dividido en dos grandes fases, en la primera se construyó un perfil UML para los servicios declarativos, y, en la segunda, se implementó éste perfil en una herramienta de software con tres partes principales: un editor gráfico de modelos, un validador de modelos y un generador de código.

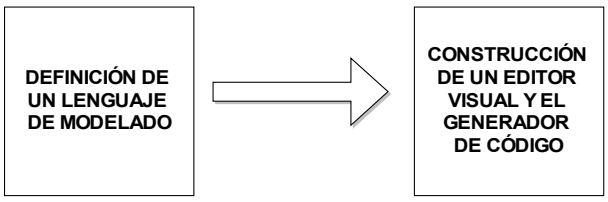

Figura 23: Proceso definido para la creación de un herramienta MDA

En éste capítulo se presenta el detalle de la fase I, la definición de un lenguaje de modelado. En los siguientes capítulos se presentará el detalle de la fase II, la construcción del editor visual y la generación de código.

4.1. Proceso para la creación de un perfil UML

La creación de un perfil UML se puede dividir en varias fases, y la notación utilizada para ello es la definida en la versión 2.0 de UML. Los pasos a seguir para la definición de un perfil son los siguientes [17] [14]:

1. Antes de comenzar es preciso disponer de la correspondiente definición del metamodelo del dominio de la aplicación o la plataforma a modelar con el perfil. En caso de que este no exista entonces deberá proceder a definir este metamodelo utilizando los mecanismos que el propio UML nos ofrece, tal y como se hace en el desarrollo de cualquier aplicación. Se deberá de definir las entidades propias del dominio, las relaciones entre ellas y las restricciones que limitan el uso de estas entidades y sus relaciones.

2. Una vez que se cuente con el metamodelo del dominio del problema, se procede a definir el perfil. Dentro de un paquete «profile» se incluirá un estereotipo por cada uno de los elementos del metamodelo que deseemos incluir en el perfil. Estos estereotipos deberán tener el mismo nombre que los elementos en el metamodelo, de esta manera se podrá establecer una relación directa entre ambos.

3. Es importante tener claro cuales son los elementos del metamodelo de UML que estamos

extendiendo sobre los cuales se puede aplicar un estereotipo. Se deberán de definir como valores etiquetados de los elementos del perfil los atributos que aparezcan en el metamodelo, así mismo incluir la definición de sus tipos y sus posibles valores iniciales.

4. Definir las restricciones que forman parte del perfil en OCL, algunas de ellas en relación directa con las relaciones entre los elementos del dominio y otras en relación con las restricciones establecidas en el mismo dominio. Es decir, las multiplicidades de las asociaciones que aparecen el el metamodelo del dominio, o las propias reglas del negocio de la aplicación deben traducirse en restricciones dentro del perfil.

Con base en estos cuatro puntos la primera fase se subdividió en varias actividades. La figura 24 describe el proceso que se definió para la creación del perfil UML.

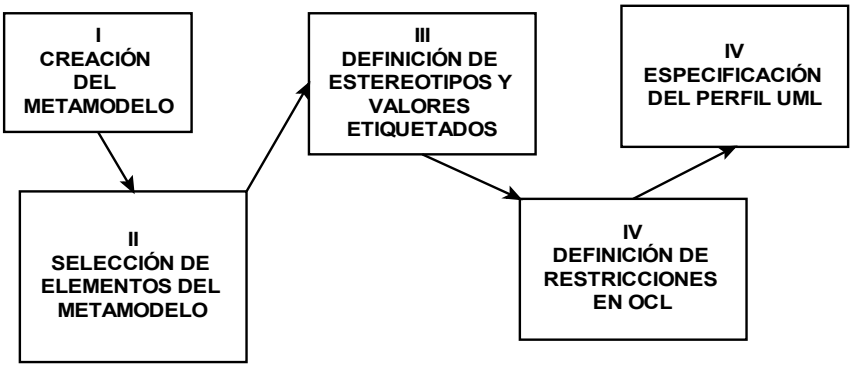

Figura 24: Proceso definido para la creación de un perfil UML

En el contexto de este proyecto, no se contaba con un metamodelo existente para el dominio, que en este caso corresponde a los servicios declarativos de OSGi, así que para poder definir el perfil UML se comenzó con la creación del metamodelo del dominio específico.

Como base para la creación del metamodelo se tomó la especificación de los servicios declarativos, poniendo especial atención en las entidades que componen el modelo y el descriptor del componente descritos en la sección 3.1.1.

En este esquema se pueden apreciar muchas de las entidades definidas para los servicios declarativos, y, que aunque no contiene toda la información necesaria para crear el metamodelo, se utilizó como base principal para comenzar con la creación del perfil UML para los servicios declarativos.

De la figura 24, primero se debe crear un metamodelo del dominio para los servicios declarativos. Este trabajo consistió en el estudio exhaustivo de la especificación de OSGi para los servicios declarativos de manera que se pudieran plasmar todos los conceptos de interés en éste metamodelo. El resultado de este trabajo se presenta a continuación.

4.2. Definición del metamodelo del dominio

Muchos de los elementos definidos en la versión final del metamodelo fueron extraídos del esquema XML del descriptor del servicio (sección 3.1.1). Sin embargo existen muchas reglas en la construcción de estos descriptores que no se encuentran en el esquema XML, y que tuvieron que ser

extraídas directamente del texto de la especificación. Es importante recalcar este punto, ya que es una contribución importante de este trabajo.

En la figura 25 se presenta el metamodelo final, en éste, se presentan todos los elementos del dominio que son de interés para nuestro trabajo. Este modelo abstracto es usado para caracterizar los servicios declarativos y será la base principal para la definición del perfil UML.

Este metamodelo resume muchas de las reglas que hay que conocer para poder desarrollar una aplicación utilizando los servicios declarativos:

a) La entidad central es *Component*, este posee una serie de atributos que definirán su comportamiento como lo son *immediate, delayed y factory*.

b) Este componente puede proveer un servicio (*Service*) y puede depender de muchos otros (*Reference*). El código del componente será implementado en una clase especificada en *Implementation* y su configuración será escrita de forma declarativa en un descriptor de componente (*ComponentDescription*).

c) Esta descripción del componente será utilizada por la configuración del componente (*ComponentConfiguration*) para configurar y controlar la instancia del componente (*ComponentInstance*) que será creada por el SCR (*ServiceComponentRuntime*).

d) Adicionalmente a esto se establecen ciertas restricciones en OCL relacionadas con el comportamiento de los componentes de acuerdo a su tipo (*Immediate, Factory o Delayed*), tomando en cuenta las reglas definidas en la especificación de los servicios declarativos.

e) Un servicio puede proveer varias interfaces (*Provide*) para su ligado con otros servicios

f) Una referencia deberá especificar aspectos como su cardinalidad, su política de ligado y los métodos de ligado y liberación de servicios.

g) Un componente puede tener varias propiedades útiles para realizar su configuración.

4.3. Perfil UML para los servicios declarativos

Con base en el metamodelo definido, fue construido el perfil UML para los servicios declarativos. El proceso de creación fue iterativo e incremental, validando las distintas versiones del perfil mediante la creación de modelos basados en él y observando posibles problemas en estos. Cabe señalar que el perfil UML para los servicios declarativos no pretende ser un perfil que pueda utilizarse para definir en toda su complejidad el funcionamiento de la plataforma OSGi, la intención de este proyecto sólo está enfocada en la definición de un perfil con el cual puedan modelarse descriptores de componente para ser utilizados dentro de los *bundles* de un servicio de OSGi.

Este perfil fue definido utilizando los mecanismos de extensibilidad de UML y los estándares especificados para la descripción de un perfil UML dados por la OMG. Para realizar la transformación del metamodelo al perfil, se tomaron aquellas clases de relevancia para nuestro trabajo y se transformaron en estereotipos, sus atributos en valores etiquetados y muchas de las relaciones entre clases se convirtieron en restricciones en OCL. Algunas clases se unieron en un solo estereotipo y otras fueron desechadas por completo debido a que no proveían información importante para nuestro modelo.

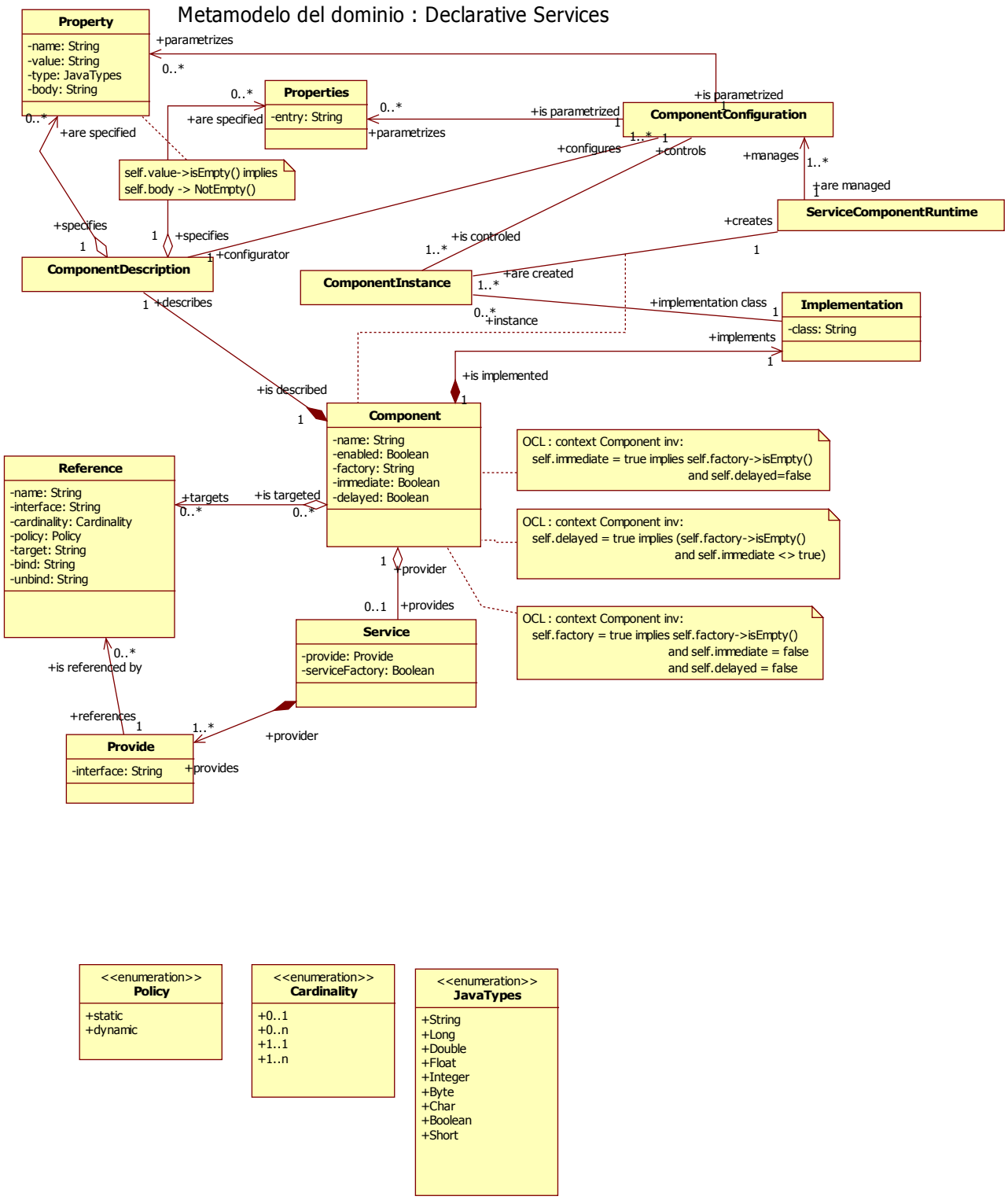

Figura 25: metamodelo de los servicio declarativos

4.3.1. Resumen de los elementos del perfil

Los estereotipos de este perfil se resumen a continuación:

Nombre	Clase Base
<<Component>>	*Class*
<<Service>>	*Class*
<<Properties>>	*Class*
<<Property>>	*Class*
<<Provide>>	*Assocation*
<<Reference>>	*Association*
<<Declares>>	*Association*

Tabla 4: Elementos del perfil

4.3.2. <<Component>>

Descripción

Un componente contiene una descripción que es interpretada en tiempo de ejecución para crear y desechar objetos dependiendo de la disponibilidad de otros servicios, la necesidad de dicho objeto o los datos de configuración del mismo. Dichos objetos pueden de manera opcional además proveer un servicio.

Representación UML para <<Component>>

Elemento del metamodelo	Nombre del estereotipo	Clase Base de UML	Superclase (del metamodelo)
Component, Implementation, Service	<<Component>>	Class	NA

Definición de valores etiquetados para <<Component>>

Nombre del estereotipo	Valor Etiquetado	Tipo del valor etiquetado	Valor por omisión	Definición del valor etiquetado
<<Component>>	*name*	*String*	requerido	Especificado por el usuario
	enabled	*Boolean*	*true*	
	factory	*String*	no requerido	Especificado por el usuario
	immediate	*Boolean*	no requerido	
	delayed	*Boolean*	no requerido	
	implementationClass	*String*	requerido	Especificado por el usuario
	serviceFactory	*Boolean*	no requerido	

Restricciones de <<Component>>

```
OCL :
package DeclarativeServices
    context Component
    inv:
```

Existen cero o varias asociaciones «Provide» para cada <<Component>>:

```
self.connection -> select(isStereotyped('Provide'))->size() >= 0
```

Existe cero o muchas asociaciones «Reference» para cada <<Component>>:

```
self.connection -> select(isStereotyped('Reference'))->size() >= 0
```

Existe cero o varias asociaciones «Declares» para cada <<Component>>:

```
self.connection -> select(isStereotyped('Declares'))->size() >= 0
```

Si *immediate* es verdadero entonces *factory* debe estar vacío y *delayed* debe ser falso

```
self.immediate = true implies self.factory -> isEmpty() and
self.delayed = false
```

Si *delayed* es verdadero entonces *factory* debe estar vacío e *immediate* debe ser falso

```
self.delayed = true implies self.factory->isEmpty() and
self.immediate = false
```

Si el componente es de tipo *factory* entonces *immediate* y *delayed* deben ser falsos

```
self.factory -> NotEmpty() implies self.immediate = false and
self.delayed = false
```

```
endpackage
```

4.3.3. <<Service>>

Descripción

Especifica las interfaces que ofrece un componente para ofrecer o referenciar un servicio.

Representación UML para <<Service>>

Elemento del metamodelo	Nombre del estereotipo	Clase Base de UML	Superclase (del metamodelo)
Provide,Service	<<Service>>	Class	NA

Definición de valores etiquetados para <<Service>>

Nombre del estereotipo	Valor Etiquetado	Tipo del valor etiquetado	Valor por omisión	Definición del valor etiquetado
<<Service>>	*interface*	*String*	requerido	Especificado por el usuario

Restricciones de <<Service>>

```
OCL :
package DeclarativeServices
     context Service
     inv:
```

Existe una asociación «Provide» para cada «Service»:

```
self.connection -> select(isStereotyped('Provide'))->size() = 1
```

Existe una o más asociaciones «Reference» para cada «Service»:

```
      self.connection -> select(isStereotyped('Reference'))->size() >= 0
endpackage
```

4.3.4. <<Properties>>

Descripción.

Un componente puede definir una serie de propiedades, que serán utilizadas para su configuración. En este caso las propiedades están en un archivo dentro del *bundle*

Representación UML para <<Properties>>

Elemento del metamodelo	Nombre del estereotipo	Clase Base de UML	Superclase (del metamodelo)
Properties	<<Properties>>	Class	NA

Definición de valores etiquetados para <<Properties>>

Nombre del estereotipo	Valor Etiquetado	Tipo del valor etiquetado	Valor por omisión	Definición del valor etiquetado
<<Properties>>	*entry*	*String*	requerido	Especificado por el usuario

Restricciones de <<Properties>>

```
package DeclarativeServices
      context Properties
      inv:
```

Existe sólo una asociación «Requires» para cada Properties:

```
      self.connection -> select(isStereotyped('Requires'))->size() = 1
endpackage
```

4.3.5. <<Property>>

Definición

Un componente puede definir una serie de propiedades, que serán utilizadas para su configuración. En este caso las propiedades se definen en la misma descripción del componente.

Representación UML para <<Property>>

Elemento del metamodelo	Nombre del estereotipo	Clase Base de UML	Superclase (del metamodelo)
Property	<<Property>>	Class	NA

4.3.5.1.Definición de valores etiquetados para Property.

Nombre del estereotipo	Valor Etiquetado	Tipo del valor etiquetado	Valor por omisión	Definición del valor etiquetado
<<Property>>	*name*	*String*	requerido	Especificado por el usuario

value	*String*	no requerido	Especificado por el usuario
type	*javaTypes*	*'String'*	*'String', 'Long', 'Double', 'Float', 'Integer', 'Byte', 'Char', 'Boolean', 'Short'*
body	*String*	no requerido	Especificado por el usuario

Restricciones de <<Property>>

```
package DeclarativeServices
    context Property
    inv:
```

Existe sólo una asociación «Requires» para cada *Property*:

```
    self.connection -> select(isStereotyped('Requires'))->size() = 1
```

Si *value* esta vacío implica que *body* debe de tener asignado un valor.

```
    self.value->isEmpty() implies not self.content -> isEmpty()
endpackage
```

4.3.6. <<Reference>>

Definición

La mayor parte de los bundles requerirá de otros servicios del registro de servicios. Estas dependencias son definidas vía referencias a los servicios requeridos.

Representación UML para <<Reference>>

Elemento del metamodelo	Nombre del estereotipo	Clase Base de UML	Superclase (del metamodelo)
Reference	<<Reference>>	Association	NA

4.3.6.1.Definición de valores etiquetados para Reference.

Nombre del estereotipo	Valor Etiquetado	Tipo del valor etiquetado	Valor por omisión	Definición del valor etiquetado
<<Reference>>	*name*	*String*	requerido	Especificado por el usuario
	targetFilter	*String*	no requerido	Especificado por el usuario
	bind	*String*	no requerido	Especificado por el usuario
	unbind	*String*	no requerido	Especificado por el usuario
	cardinality	*Cardinality*	'1..1'	'1..1', '1..n', '0..1', '0..n'
	policy	*Policy*	*'static'*	*'static', 'dinamic'*

Restricciones de <<Reference>>

```
OCL :
package DeclarativeServices
     context Reference
     inv:
```

La relación debe estereotiparse «Reference»

```
self.isStereotyped('Reference')
```

La asociación es de tipo binario

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

Una relación de tipo «Reference» tendrá una entidad «Component» de un lado y una entidad «Service» del otro:

```
self.connection -> exists(participant.isStereotyped('Component'))
and self.connection-> exists(participant.isStereotyped('Service'))
```

La cardinalidad del lado «Component» es 1

```
self.connection -> exists(participant.isStereotyped('Component')) and
multiplicity.min = 1 and multiplicity.max = 1
```

La cardinalidad del lado «Service» es 1

```
self.connection -> exists(participant.isStereotyped('Service')) and
multiplicity.min = 1 and multiplicity.max = 1

endpackage
```

4.3.7. <<Provide>>

Definición

Un componente puede ofrecer un servicio, para establecer que interfaces de servicios proveerá un componente se utilizará la relación <<Provide>>.

Representación UML para <<Provide>>

Elemento del metamodelo	Nombre del estereotipo	Clase Base de UML	Superclase (del metamodelo)
Provide	<<Provide>>	Association	NA

Definición de valores etiquetados para <<Provide>>

Nombre del estereotipo	Valor Etiquetado	Tipo del valor etiquetado	Valor por omisión	Definición del valor etiquetado
Provide	NA	NA	NA	NA

Restricciones de <<Provide>>

```
OCL : package DeclarativeServices
   context Provide
   inv:
```

La relación debe estereotiparse «Provide»

```
self.isStereotyped('Provide')
```

La asociación es de tipo binario

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

Una relación de tipo «Provide» tendrá una entidad «Service» de un lado y una entidad «Component» del otro:

```
self.connection -> exists(participant.isStereotyped('Provide'))
and self.connection-> exists(participant.isStereotyped('Component'))
```

La cardinalidad del lado «Service» es 1

```
self.connection -> exists(participant.isStereotyped('Provide')) and
multiplicity.min = 1 and multiplicity.max = 1
```

La cardinalidad del lado «Provide» es 1

```
self.connection -> exists(participant.isStereotyped('Component')) and
multiplicity.min = 1 and multiplicity.max = 1
```

```
endpackage
```

4.3.8. <<Declares>>

Definición

Algunas veces la instancia del componente requiere ser parametrizada por medio de propiedades definidas por el usuario, estas propiedades serán agregadas a un componente vía la relación <<Declares>>.

Representación UML para <<Declares>>

Elemento del metamodelo	Nombre del estereotipo	Clase Base de UML	Superclase (del metamodelo)
Property, Properties, specifies, parametrizes	<<Declares>>	Association	NA

4.3.8.1.Definición de valores etiquetados para Declares

Nombre del estereotipo	Valor Etiquetado	Tipo del valor etiquetado	Valor por omisión	Definición del valor etiquetado
Declares	NA	NA	NA	NA

Restricciones de <<Declares>>

```
OCL : package DeclarativeServices
   context Declares
   inv:
```

La relación debe estereotiparse «Declares»

```
self.isStereotyped('Declares')
```

La asociación es de tipo binario

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

Una relación de tipo «Declares» tendrá una entidad «Component» de un lado y una entidad «Properties» o «Property» del otro:

```
self.connection ->
   exists(participant.isStereotyped('Component')) and
    (  self.connection-> exists(participant.isStereotyped('Property')) or
       self.connection-> exists(participant.isStereotyped('Properties'))   )
```

La cardinalidad del lado «Component» es 1

```
self.connection -> exists(participant.isStereotyped('Component')) and
multiplicity.min = 1 and multiplicity.max = 1
```

La cardinalidad del lado «Properties» es 1 o la cardinalidad del lado «Property» es 1

```
self.connection -> exists(participant.isStereotyped('Properties')) and
multiplicity.min = 1 and multiplicity.max = 1 or
self.connection -> exists(participant.isStereotyped('Property')) and
multiplicity.min = 1 and multiplicity.max = 1


endpackage
```

La figura 26 muestra los elementos del perfil UML de los servicios declarativos de OSGi.

Se puede apreciar además de las clases definidas tres clases adicionales estereotipadas como *Enumeration.* Estas son utilizadas para definir un conjunto de valores que pueden ser tomados por valores etiquetados que utilicen como tipo de dato un tipo especial, por ejemplo se puede notar que en la clase *Reference* existe un valor etiquetado llamado *policy*, que recordando es la forma en que el Componente de servicio manejará el ligado con otros servicios, que puede ser de dos formas: estático y dinámico, estos valores son listados en una enumeración para permitir que el valor etiquetado sólo pueda tomar cualquiera de los dos. Así mismo existen enumeraciones para listar los valores que pueden tomar la cardinalidad y el tipo de dato de una propiedad.

4.3.9. Validación del perfil UML de los servicios declarativos

La validación del perfil fue realizada a través del modelado de aplicaciones OSGi utilizando los elementos del perfil. La figura 27 muestra un modelo de una de las aplicaciones modeladas para validar el perfil.

El modelo está basado en un ejemplo que es parte de un tutorial de OSGi y el *ServiceBinder* [38]. Este modelo representa varios componentes ofreciendo sus servicios y haciendo referencia a otros para proveer de un servicio de revisión ortográfica a un editor de textos. Existe un componente llamado *Dictionary* que ofrece dos servicios, un servicio de diccionario en inglés y uno en francés. Estos servicios son referenciados por un componente llamado *SpellCheck* el cual ofrece un servicio de revisión ortográfica llamado *SpellCheckService* que se encargará de utilizar los diccionarios para realizar la revisión ortográfica de un texto. Por ultimo el componente *WordEditor* hace referencia al servicio *SpellCheckService* para ofrecer dentro de sus características la corrección ortográfica.

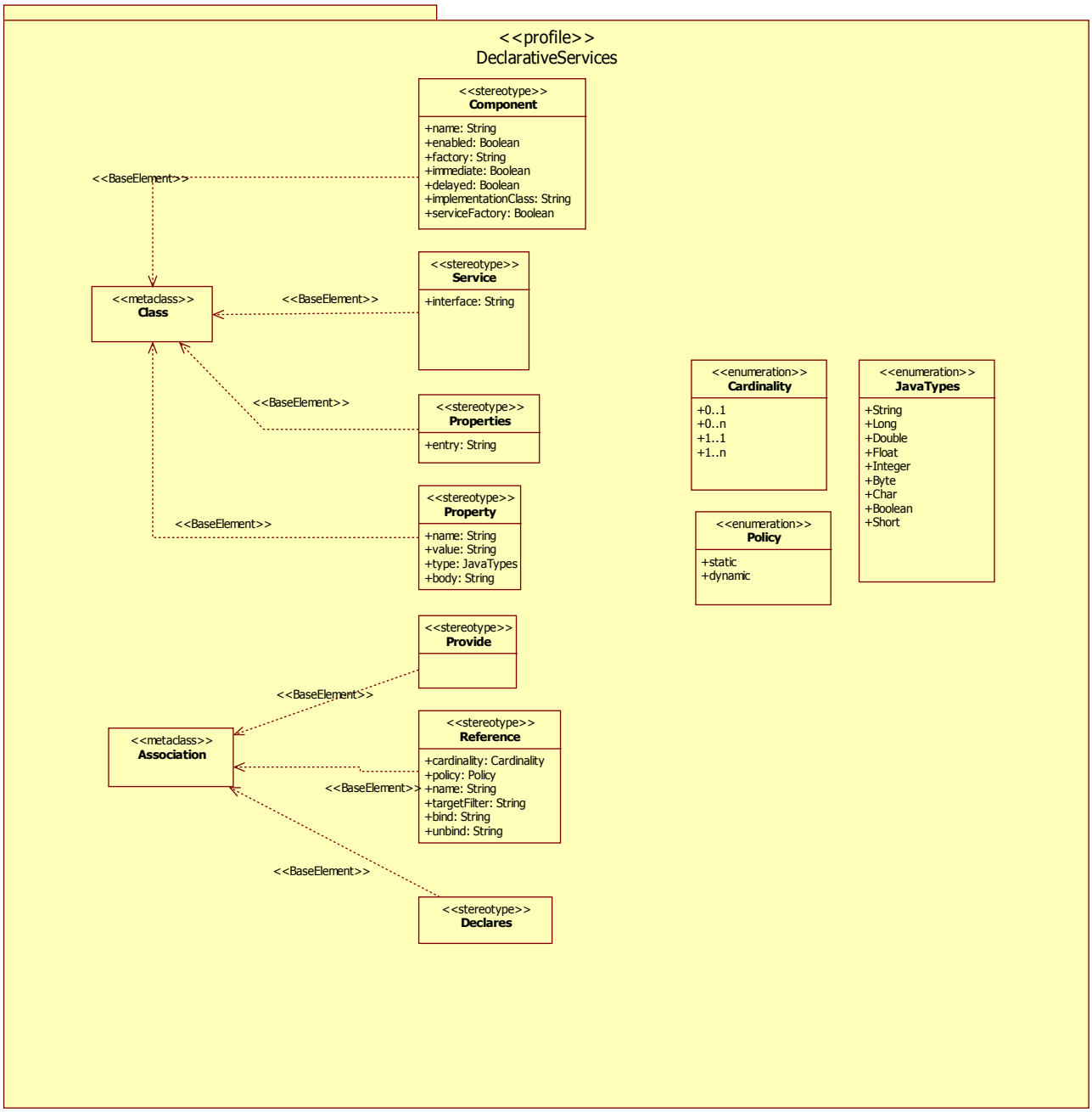

Figura 26: Paquete del perfil UML para los servicios declarativos de OSGi

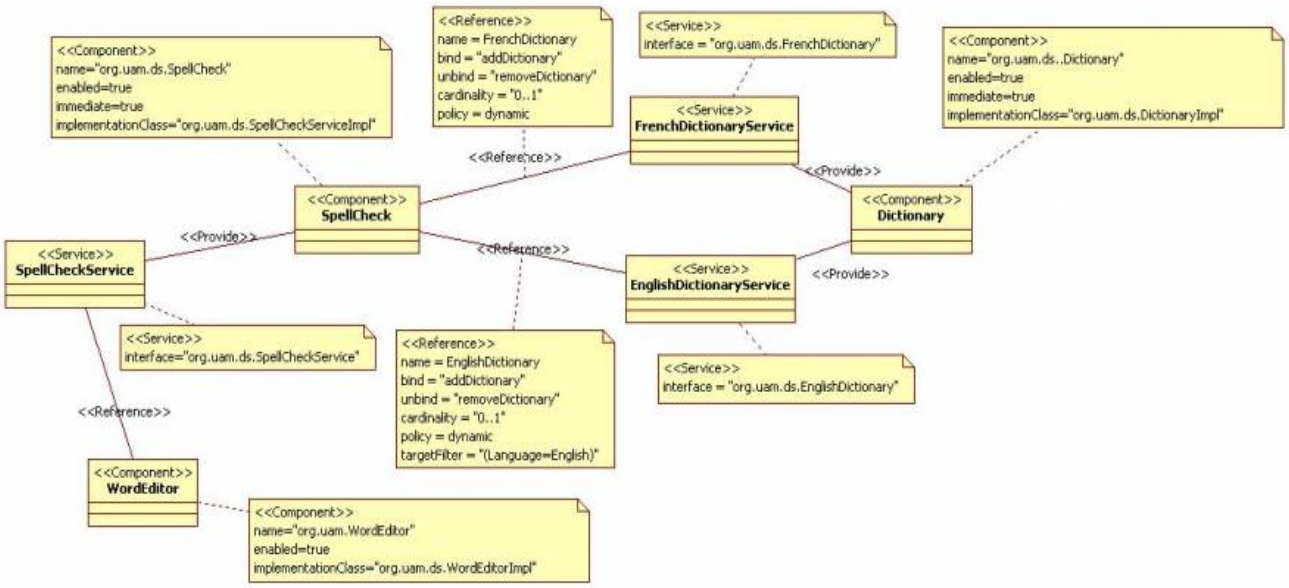

Figura 27: Modelo de componentes creado con el perfil UML de los servicios declarativos

La siguiente fase del proyecto fue la construcción de la herramienta MDA que implementa el perfil UML, es decir, que permite crear modelos de manera visual utilizando el perfil UML y validar estos modelos utilizando las restricciones en OCL que expresan las reglas extraídas del texto de la especificación. A continuación se presenta el trabajo de desarrollo del editor visual y el generador de código.

5. Creación de la herramienta MDA

Este capítulo presenta el proceso de construcción del editor visual que permite la creación de modelos basados en el perfil UML para los servicios declarativos, y validarlos utilizando las restricciones en OCL definidas en el mismo perfil para después generar el descriptor XML y el esqueleto del código del componente.

5.1. Aspectos a considerar en la creación de la herramienta y proceso de desarrollo

Los requerimientos de la herramienta que se desarrolló para este proyecto incluyeron:

a) La herramienta debía ser construida siguiendo los estándares de la OMG para MDA. Esto es importante para hacer la herramienta final fácilmente extensible y capaz de comunicarse con otras aplicaciones que también sigan los estándares de la OMG (MDA, OCL, UML y XMI).

b) La herramienta debía ser construida siguiendo un proceso bien definido, sencillo y fácil de replicar que pudiera ser retomado para crear una herramienta nueva para un dominio similar o completamente distinto al de SOA. De esta forma se establece un proceso para facilitar la adopción de MDA en una mayor cantidad de proyectos de desarrollo de software.

c) La construcción de la herramienta está dividida en dos fases, en la primera fase se llevó a cabo la construcción del editor visual que valida los modelos de acuerdo a las restricciones OCL. En la segunda fase se definieron las reglas de transformación entre modelo y código y se construyó el generador de código que las implementa (figura 28).

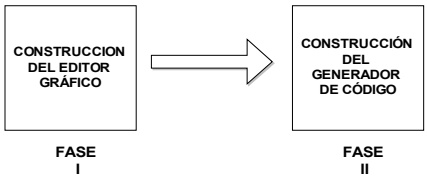

Figura 28: Proceso definido para la implementación del perfil en la herramienta MDA

d) El desarrollo de la herramienta tenía que ser acelerado debido al corto tiempo, así que para ambas fases se realizó una investigación sobre *frameworks* de desarrollo. Las herramientas de soporte debían también seguir los estándares definidos por la OMG para MDA.

e) Un aspecto importante que se consideró en la selección de la herramienta de soporte para la construcción del editor gráfico fue que nos proveyera alguna manera de capturar nuestras reglas escritas en OCL para la validación del modelo.

f) La herramienta final se ejecuta sobre la plataforma de Eclipse. Esto es para aprovechar al máximo las ventajas que nos da el uso de *plugins* dentro del ambiente de Eclipse y para facilitar la adopción de la herramienta por parte de los desarrolladores al ser ejecutada en una ambiente donde pueda modificarse de manera inmediata el código generado, compilarse y probarse, todo dentro del mismo ambiente.

5.2. Investigación sobre las herramientas de soporte para la creación del editor gráfico y validación del modelo

Esta sección presenta tres *frameworks* que fueron considerados para ser usados como soporte para basar el desarrollo del editor visual de nuestra herramienta, GEF, Merlin y GMF, al final se presenta una comparación de todas ellas.

Se buscó una herramienta capaz de generar un editor visual a partir de un metamodelo, es decir, una herramienta MDA para la creación de editores gráficos basados en metamodelo (el lenguaje visual). Dos de las tres herramientas mostradas generan editores gráficos utilizando un metamodelo y todas utilizan como base para realizar su trabajo un *framework* de desarrollo llamado EMF (*Eclipse Modeling Framework*), que es un *framework* para el manejo de modelos y generación de código de Eclipse, por lo que, a continuación se presenta una breve introducción a lo que es EMF.

5.2.1. EMF (*Eclipse Modeling Framework*)

Es un *framework* que provee de facilidades para la definición de modelos ya sea como código, como un diagrama UML o como una descripción en XML, además de permitir realizar transformaciones entre estas distintas representaciones.

El modelo utilizado para representar modelos en EMF es llamado Ecore. Ecore es por si mismo un modelo de EMF y por tanto es su propio metamodelo. Este metamodelo es un subconjunto de UML. La ventaja mas importante de EMF es la generación de código. El modelo Ecore puede ser creado utilizando directamente una serie de editores de EMF, a través de herramientas gráficas o convirtiendo el modelo a Interfaces de Java, modelos UML expresados en formato XMI o esquemas XML a través de importadores de modelos que provee la misma herramienta [18] [27]. Una vez definido el modelo Ecore, EMF puede convertir cualquiera de las representaciones a la implementación en Java, esquemas de XML o proyectos de Eclipse.

5.2.2. GEF (*Graphical Editing Framework*)

El *framework* de GEF permite a los desarrolladores generar a partir de los modelos de una aplicación, un editor gráfico [33] [29]. El *plugin* provee un conjunto de herramientas para el manejo de gráficos. El desarrollador puede tomar ventaja de muchas de las operaciones mas comunes ofrecidas por GEF y extenderlas a un dominio específico.

GEF es completamente neutral a un dominio particular y provee las bases para construir casi cualquier aplicación de este tipo. GEF asume que se tiene un modelo que se desea desplegar y editar de manera gráfica. Para realizar esto GEF provee de visores (*viewers*) que pueden ser utilizados en cualquier parte del ambiente de Eclipse. Los visores de GEF están basados en la arquitectura MVC (Modelo-Vista-Controlador). Los controladores crean un puente entre la vista y el modelo. Cada controlador, es responsable del mapeo del modelo a su vista y de realizar los cambios necesarios al modelo. Los controladores también observan constantemente el modelo y actualizan la vista para reflejar los cambios en el estado del modelo. La vista es la representación del modelo o una de sus partes en la pantalla. Puede ser algo tan simple como un rectángulo, línea o elipse, o tan complejo como un elemento en un diagrama electrónico. Si el modelo consiste de varios fragmentos, este controlador informa a GEF acerca de esto y nuevos controladores son creados para cada fragmento. Si el proceso se repite con las subpartes entonces nuevos controladores son creados de manera que todos

los elementos del modelo tengan su controlador. Ya que ni el modelo ni la vista se ven uno al otro el controlador se encarga de esperar por cambios en la vista o el modelo. Cuando un controlador recibe una notificación reacciona apropiadamente ajustando la representación visual o estructural del modelo.

El problema con cualquier *framework* genérico y GEF no es la excepción, es que su diseño tan extenso lo hace difícil de aprender. El ejemplo más simple generado con GEF requiere arriba de 75 clases. Tratar de comprender todos los aspectos de GEF es un ejercicio de paciencia y resulta complicado aún para los desarrolladores mas experimentados [29].

A pesar de todas las ventajas de crear un editor con GEF, ésta es una tarea compleja de programación en Java. Aspectos como el manejo de la persistencia del modelo, manejo de referencias entre modelos, controladores y vistas, dependencias con otras aplicaciones y notificación de cambios en el modelo, etc. deben ser manejados por el desarrollador. Además, GEF no ofrece facilidades para la validación de los modelos creados con OCL.

5.2.3. Merlín

Merlín es un *plugin* de Eclipse basado en EMF que provee herramientas de generación de código y transformación de modelos [34][29]. Merlín viene con una serie de asistentes y editores para personalizar la generación de código y las transformaciones de modelos. Una de estas herramientas es un generador de GEF que permite a los usuarios generar editores GEF basándose en cualquier modelo EMF.

Merlín es un avance en el desarrollo de una herramienta mucho más simple de utilizar, capaz de generar editores visuales de modelos creados a partir de su metamodelo, sin embargo, aun no es capaz de validar los modelos usando las reglas en OCL. Los editores visuales generados con Merlín son muy básicos y si se desea personalizar su comportamiento de nuevo se tiene que modificar el código generado, por lo que se requiere conocer de aspectos de programación con GEF.

5.2.4.GMF (*Graphical Modeling Framework*)

La idea de llenar el espacio vacío entre GEF y EMF para generar editores visuales basándose en metamodelos fue tomada por un proyecto de Eclipse llamado GMF. Muchas de las características de Merlín terminaron como parte del proyecto GMF debido a que Merlín fue uno de los participantes iniciales. A pesar de que Merlín representa un avance en este sentido sobre GEF, GMF es una herramienta más poderosa, que no sólo resuelve el problema de la creación del editor gráfico sino que ofrece funciones para establecer validación de los modelos mediante el uso de las restricciones en OCL.

El *framework* de GMF puede ser dividido en dos partes principales: las herramientas y el motor de ejecución. Las herramientas consisten de editores para crear y modificar modelos que describen los aspectos semánticos, de notación y visualización de los elementos del editor gráfico, así como de un generador para producir la implementación de los editores gráficos. Los *plugin* generados dependen del motor de ejecución de GMF para producir un editor gráfico [24].

El proyecto de GMF intenta ocultar todos los modelos que utiliza mediante el uso de asistentes, sin embargo, cuando se intenta personalizar el funcionamiento de los editores generados se tiene que aprender el uso y razón de cada uno de estos modelos, así como las funciones principales de estos en el desarrollo del editor gráfico. Mas adelante en este trabajo se detalla este proceso, pero como breve introducción se puede decir que GMF maneja todo el desarrollo del editor visual basándose en modelos

visuales que representan distintos elementos del editor. Así, existirá un modelo que represente el metamodelo del dominio para el que el editor será creado, existe un modelo de definición gráfica que contiene información relacionada con los elementos gráficos que aparecerán en el motor de ejecución basado en GEF, pero no tienen ninguna conexión directa con los modelos del dominio definidos que van a representar. Un modelo opcional que se define también es el de modelo de definición de herramientas que es utilizado para diseñar la paleta de dibujo y otros aspectos como menús, barras de herramientas, etc.

Puede esperarse que los elementos gráficos y la definición de herramientas puedan ser idénticos para varios dominios, por lo que una meta de GMF es permitir que estos puedan ser reutilizados para otros dominios. Esto se logra gracias a un modelo separado donde se mapean todos los elementos del dominio a sus contrapartes gráficas. Una vez que los mapeos están definidos, GMF provee un modelo generador que permite que sean definidos todos los aspectos relacionados con la generación del código del editor visual.

Con GMF, poco o ningún código es generado por el desarrollador, todo el código de la aplicación es generado en su totalidad por GMF, permitiendo el desarrollo de editores visuales bastante complejos y completos sin necesidad de conocer muchos aspectos de EMF y GEF, se puede decir que GMF es una herramienta MDA para la generación de editores visuales.

5.2.5. Comparación y selección de la herramienta para el desarrollo del editor gráfico

A continuación se presenta una tabla de comparación entre los distintas *frameworks*. Se realizaron una serie de pequeños prototipos en los distintos *frameworks* de manera que pudiera compararse el desarrollo de editores con cada uno. Los aspectos que se consideraron a la hora de la comparación fueron:

a) **Facilidad de uso y aprendizaje.** Debido al tiempo corto del proyecto, el que el *framework* pudiera explotarse de una manera inmediata sin tener que llevar una curva de aprendizaje muy alta era muy importante.

b) **Soporte de estándares MDA de la OMG.** Es importante para poder llegar a la extensibilidad deseada que el *framework* implemente estándares como OCL, XMI, UML, etc.

c) **Tiempo de creación del editor.** Durante la creación de los prototipos un aspecto importante fue con que rapidez fueron creados, comparando los tiempos que se tardó en realizar el mismo prototipo funcional en los distintos *frameworks.*

d) **Integración con otras herramientas.** Se realizó una investigación sobre las facilidades que ofrecen los *frameworks* para integrarse fácilmente con herramientas que sigan los estándares definidos en el desarrollo sobre Eclipse como GEF y EMF.

e) **Soporte OCL.** Uno de los aspectos más relevantes fue el soporte de definición de reglas de validación de los modelos en OCL dentro del *framework*.

f) **Utilización de metamodelos para la generación del editor.** Se creó un perfil UML para los servicios declarativos con la idea de usarlo como base para el desarrollo del editor. El que el *framework* facilitara el uso de éste metamodelo para la creación del editor fue un aspecto importante en la selección.

Herramienta	Característica evaluada
	Utilización de metamodelos para la generación del editor
GEF	No define claramente como se generar un editor basándose en un metamodelo o herramientas para su manejo.
Merlín	Utiliza EMF para la definición del metamodelo y a partir de este genera el editor.
GMF	Utiliza EMF para la definición del metamodelo y a partir de este genera el editor. También cuenta con un editor visual para la definición del metamodelo.
	Soporte OCL para la validación de los modelos generados con el editor.
GEF	No tiene ninguno.
Merlín	No tiene ninguno.
GMF	Restricciones en OCL sobre los elementos del metamodelo pueden ser definidas para validar los modelos.
	Soporte de estándares MDA de la OMG.
GEF	El soporte de estos estándares tiene que ser desarrollado por el programador.
Merlín	Soporta XMI y metamodelos
GMF	Soporta XMI, OCL y metamodelos.
	Facilidad de aprendizaje y uso. Tiempo de creación del editor.
GEF	Representa un reto importante para un desarrollador utilizar y comprender el *framework* de GEF para su correcta utilización. El desarrollo se hace por completo en código Java. Crear un editor en GEF puede llevar bastante tiempo, dependiendo de las capacidades de los desarrolladores para entender y explotar el *framework*.
Merlín	Basado en modelos, al inicio puede ser complicado sino se tiene conocimiento sobre modelos, pero puede llegar a ser muy sencillo la creación de un editor simple, el problema es que solo pueden generarse editores de cierta complejidad, provocando que si se requiere agregar funcionalidad se tenga que conocer GEF. Crear un editor simple utilizando Merlín es una tarea de horas.
GMF	Basado en modelos, al inicio puede ser complicado si no se tiene conocimiento sobre modelos, pero después de la creación de un sencillo ejemplo pueden comenzar a crearse editores bastante complejos. Una vez comprendidos todos los modelos involucrados en la creación del editor en GMF, crear un editor complejo con GMF es una tarea de días.
	Integración con otras herramientas
GEF	Puede integrarse fácilmente con otros plugins de Eclipse.
Merlín	Basado en los plugins de EMF y GEF, puede integrarse fácilmente con otros plugins de Eclipse.
GMF	Basado en los plugins de EMF y GEF, puede integrarse fácilmente con otros plugins de Eclipse.

Tabla 5: Comparación de las herramientas para la creación del editor gráfico.

Fue seleccionado como *framework* para el desarrollo del editor a GMF. El motivo más importante para la selección fue que facilita la captura de las restricciones del perfil en OCL, de manera que el perfil pudo ser utilizado en su totalidad para la creación del editor. La siguientes secciones describen en detalle el proceso de desarrollo del editor gráfico utilizando GMF y los resultados finales que se obtuvieron.

5.3. Proceso de desarrollo del editor gráfico con GMF.

5.3.1. Desarrollo con GMF.

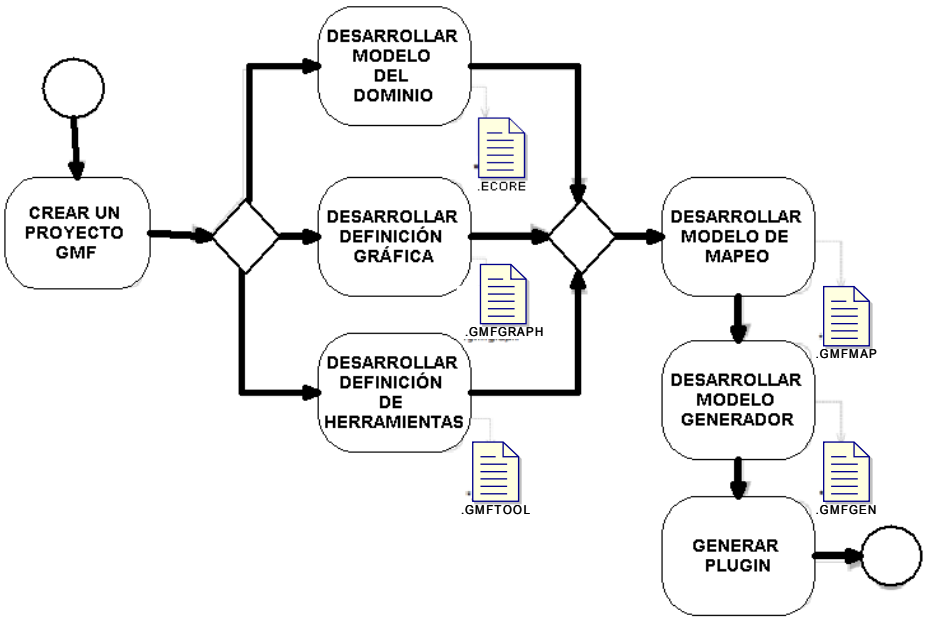

Figura 29: Modelos utilizados y proceso de desarrollo con GMF

GMF maneja una serie de modelos EMF que describen varios aspectos relacionados con el editor visual que será creado, se puede decir que son distintas vistas del mismo editor. La figura 29 muestra los modelos y componentes que se utilizan en el desarrollo con GMF, así como el orden en que éstos son generados [19] [24] [26]. Como parte importante de GMF se encuentra el modelo de definición gráfico, este modelo contiene la información relacionada con los elementos gráficos que presentará el editor, sin embargo, estos elementos no tienen conexión directa con los elementos definidos en el modelo del dominio, en nuestro caso, el perfil UML, de manera que los elementos gráficos presentados podrían idealmente utilizarse para editores de distintos dominios, por ejemplo, los conceptos de proveedores y consumidores que se manejan en varios dominios (*Web Services*, .NET, etc). Adicionalmente existe un modelo de definición de las herramientas, el cual representa los elementos como la paleta de dibujo, menús y barras de herramientas que poseerá el editor.

Estos tres modelos son mapeados y relacionados por medio de un modelo de mapeo, el cual relaciona los elementos del dominio con los elementos gráficos y los elementos del modelo de herramientas. Aquí mismo, en este modelo de mapeo, se definen las restricciones OCL utilizadas por el editor para validar los modelos generados. A partir de este modelo se genera un quinto modelo llamado generador, en el cual se personalizan algunos aspectos relacionados con la fase de generación. Todos los modelos creados para este proyecto y una explicación detallada de su estructura puede ser consultada en el apéndice I.

5.3.2. Editor gráfico para los servicios declarativos

A partir de los modelos y sin haber escrito ni una sola línea de código, se generó el código del editor visual para los servicios declarativos, el cual se ejecuta como un *plugin* de Eclipse. La figura 30 muestra la pantalla de creación de un nuevo diagrama DS para los servicios declarativos.

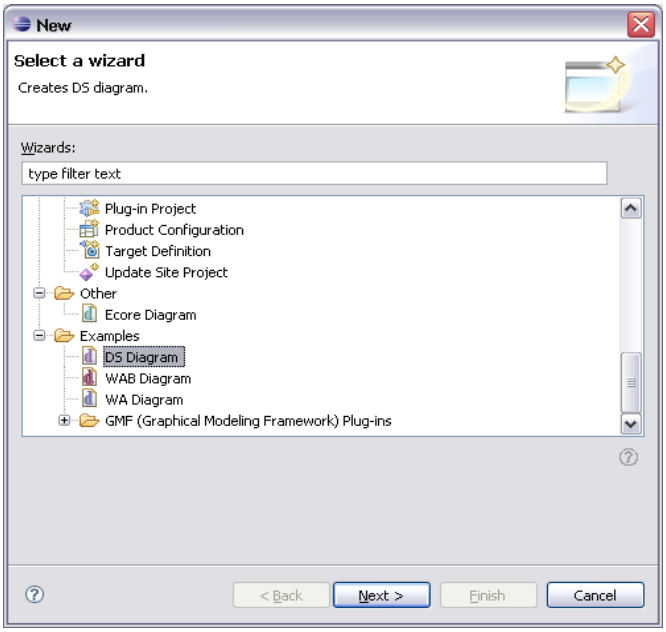

Figura 30: Asistente para la creación de un nuevo diagrama DS

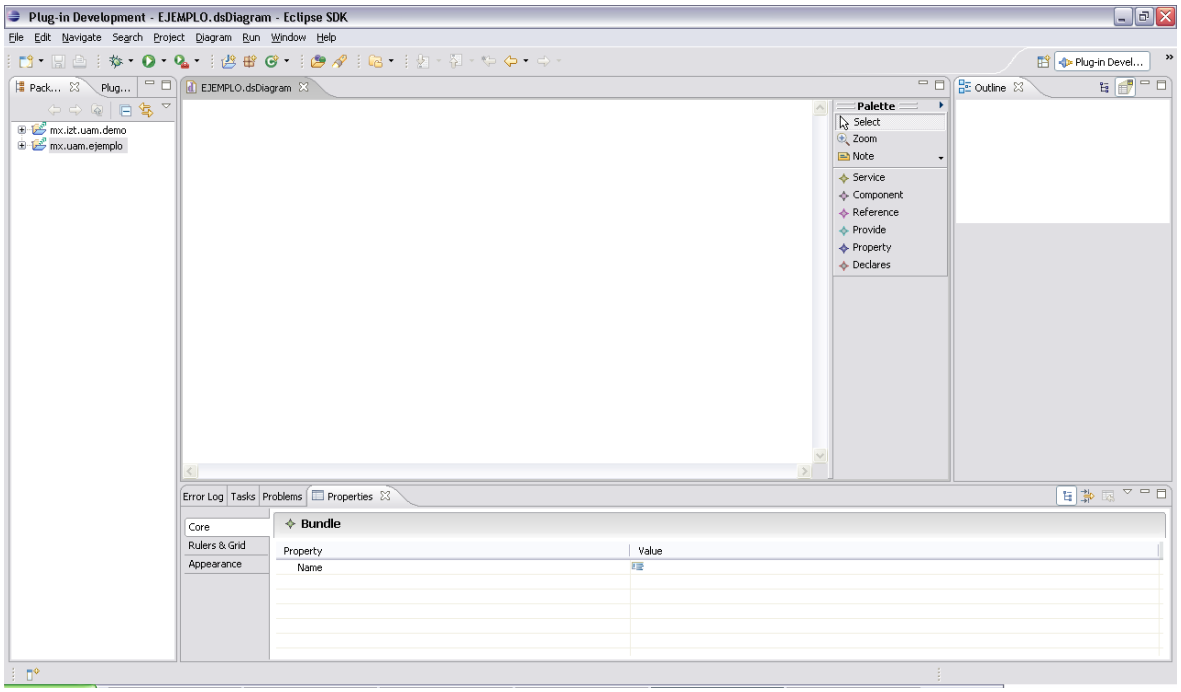

Figura 31: Editor gráfico para los servicios declarativos

En la figura 31 se muestra la pantalla principal del editor, la cual tiene una paleta de dibujo con los

elementos definidos en el modelo de definición de las herramientas (*Component, Service, Property, Reference, Provide, Declares*).

La figura 32 presenta un modelo creado utilizando la herramienta.

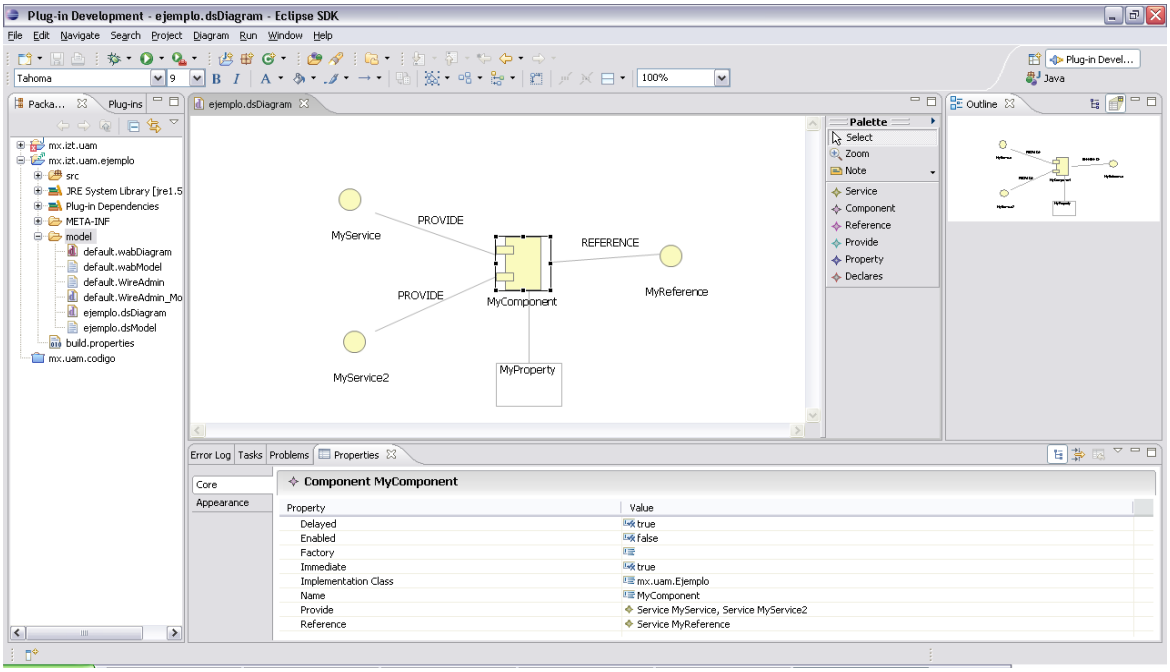

Figura 32: Ejemplo de modelo creado utilizando la herramienta

Cuando se selecciona algún elemento del modelo, en la parte inferior se muestran las propiedades del mismo, que no son otra cosa que los valores etiquetados definidos en el perfil UML.

En el ejemplo que se muestra en la figura 32 se aprecia un error que, de no conocer la especificación para los servicios declarativos podría cometerse, el componente tiene sus propiedades *Delayed* e *Immediate* igual a true. El problema es que un componente debe puede ser tardío o inmediato, no ambos. Un error más en este pequeño ejemplo es el indicar que el componente provee dos servicios (*MyService* y *MyService2*), lo cual también es incorrecto según las especificaciones de los servicios declarativos. El editor, al tener integradas todas las restricciones del dominio en OCL es capaz de validar el modelo y determinar errores de este tipo.

La figura 33 muestra los errores en el modelo después de validarlo con la herramienta y las marcas que son establecidas en los elementos que presentan los errores. En la misma figura se pueden apreciar algunos errores mas, estos tienen que ver con que algunas propiedades no tienen datos, necesarios para la generación correcta del descriptor XML del componente.

De esta forma, se tiene un editor gráfico para modelar *bundles* de manera visual que sigue los estándares de la OMG, y no sólo eso, sino que no se requiere conocer en detalle las especificaciones de los servicios declarativos, pues el editor se encarga de validar el modelo utilizando las reglas en OCL definidas en el perfil UML. Un punto importante es que se estableció un proceso simple para la creación del editor, proceso que puede ser adoptado por una organización de desarrollo de software para la creación editores para sus dominios específicos. Este proceso se muestra en la figura 34.

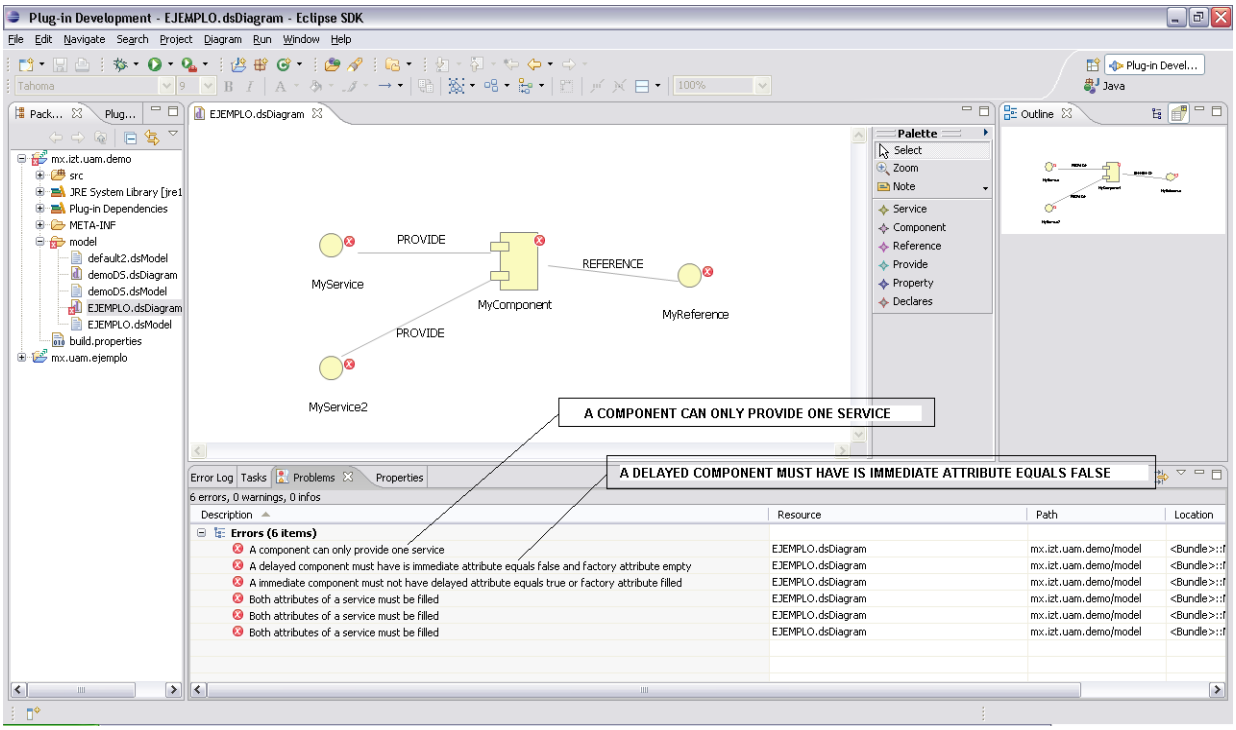

Figura 33: Errores detectados por la herramienta en base a las restricciones OCL

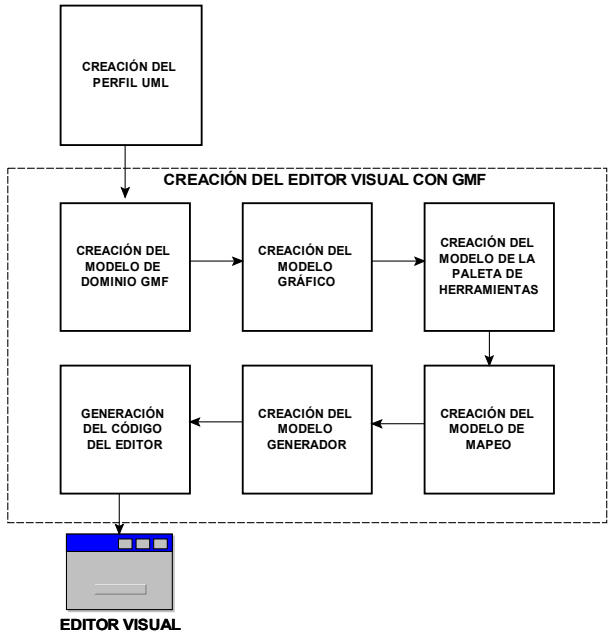

Figura 34: Proceso de creación del editor visual

5.1. Investigación de *frameworks* de soporte para la implementación de la generación de código

El siguiente paso en el proceso de construcción de la herramienta, es la generación de código a partir del modelo creado en el editor visual. Para esto dos *frameworks* fueron considerados: JET y Acceleo. Se buscaba una herramienta que pudiera integrarse de manera inmediata con el editor visual creado, para esto, la herramienta debía seguir los estándares establecidos de la OMG y de Eclipse, así como integrarse fácilmente con una aplicación cuya base fuera GEF. La herramienta debía permitir la personalización del código que generado para adaptarlo a cualquier necesidad y estilo de programación o plataforma. A continuación se presentan una breve introducción a los *frameworks* que fueron considerados en en éste trabajo.

5.1.1. JET

EMF viene con dos herramientas de generación de código, JET (*Java Emitter Templates*) y JMerge (*Java Merge*) [32]. Con JET el desarrollador utiliza una sintaxis similar a JSP para escribir plantillas que expresan el código que se desea generar. JET es un motor genérico de plantillas utilizadas para generar archivos SQL, XML, Java o cualquier otra salida de texto. Esta plantilla es convertida a una clase que lo implementa en Java. Esta clase tiene un método llamado *generate* que es ejecutado para obtener una cadena de texto.

Es relativamente simple integrar JET con un modelo EMF, sin embargo cuando se requiere generar código a partir de un editor como el generado por GMF se requiere de conocimientos sobre el *framework* de GEF, y la construcción de un generador de código puede transformarse en una tarea de desarrollo compleja.

5.1.2. Acceleo

Basado en los estándares definidos por EMF, como JET, Acceleo brinda un enfoque MDA simple en la búsqueda de la industrialización del desarrollo de software. Acceleo permite la generación de archivos a partir de modelos UML, MOF o EMF. Acceleo tiene las siguientes características [30] [31]:

a) Generación de código basada en plantillas que expresan reglas de transformación entre un modelo y el código utilizando una sintaxis similar a JSP.

b) Una completa integración con el ambiente de Eclipse y el *framework* de EMF.

c) Navegación por los elementos de cualquier modelo que siga los estándares de EMF (XMI).

d) Administración de la sincronización entre el código y el modelo.

e) Generación incremental de código.

f) Simplicidad en el mantenimiento y actualización de todas las plantillas.

g) Coloreado sintáctico de las plantillas así como detección de errores basada en el metamodelo.

h) Previsualización del código.

Acceleo provee una manera de realizar generación de código de manera incremental definiendo zonas del código específicas con *tags* de usuario. Estas zonas serán conservadas cuando el código vuelva a generarse.

Acceleo ofrece además un sistema de generación de código integrado mediante cadenas de generación son realmente simples de configurar y ejecutar. Dado una elemento en un modelo, es relativamente sencillo generar varios archivos a partir de éste, así, pueden generarse archivos XML, Java, Descriptores de Servicios y en general cualquier archivo de texto. La idea es integrar en las plantillas conceptos arquitecturales básicos así como patrones de diseño y garantizar de esta forma que las mejores prácticas de desarrollo sean tomadas como base de manera automática y consistente.

De nuevo Acceleo utiliza una sintaxis similar a JSP, ofreciendo un set de instrucciones que nos permiten realizar ciclos, tomar decisiones y navegar por los elementos del modelo de una manera muy natural, lo que provee una forma de integrar conceptos relativos a una plataforma especifica dentro de las plantillas y extraer esos conceptos de los modelos de manera que estos puedan ser lo mas independientes de la plataforma posible, Acceleo es completamente independiente de la plataforma de implementación, incluso documentación puede ser generada utilizando Acceleo.

Acceleo está basado en los principales estándares MDA, lo que garantiza su compatibilidad e interoperabilidad con otras aplicaciones, como GMF. Es compatible con XMI, lo que asegura compatibilidad con muchos de los modeladores de UML en el mercado, además fue diseñado para trabajar con cualquier metamodelo y permite extender la funcionalidad ofrecida mediante la importación de librerías de Java, que pueden utilizarse para agregar funcionalidad a las plantillas y la generación de código. Si se desea saber un poco más acerca del funcionamiento de Acceleo puede consultarse el apéndice II.

5.1.3. Comparación de las herramientas de soporte para la generación de código

Para realizar la comparación de ambas soluciones se realizó la construcción de prototipos simples utilizando ambas herramientas, para la comparación se tomaron en cuenta los siguientes aspectos:

a) **Integración con GMF.** Antes que nada la herramienta debía de poder integrarse de manera sencilla con el editor gráfico desarrollado.

b) Facilidad en la implementación de reglas de transformación. La herramienta debe permitir implementar las reglas que definen qué bloques de código se generarán para cada elemento en el modelo.

c) Independencia de la plataforma del código generado. La herramienta debe ser capaz de generar cualquier tipo de archivo de texto que se necesite.

d) **Fácil navegación por los elementos del modelo creado con el editor**. La herramienta debe proveer facilidades para acceder a los elementos del modelo creado con el editor.

e) Facilidad de uso. Por último, la herramienta debe ser sencilla de utilizar, de ser posible que el desarrollo de código se minimice, de manera que una organización de desarrollo de software pueda adoptar el enfoque sin problemas.

Herramienta	Característica evaluada
	Integración con GMF
JET	JET puede integrarse con GMF debido a que GMF está basado en GEF y EMF, sin embargo la integración no es tan simple y se requiere de un alto nivel de programación para lograrlo.
Acceleo	La integración es inmediata y sencilla, Acceleo puede acceder a los elementos de los modelos creados en cualquier editor que siga el estándar XMI.
	Facilidad en la implementación de las reglas de transformación
JET	Debido a la riqueza del lenguaje utilizado por JET, cualquier regla de transformación puede ser implementada de manera rápida.
Acceleo	Acceleo tiene un lenguaje mas restringido que JET, sin embargo, reglas de transformación bastante complejas pueden implementarse con este.
	Independencia de la plataforma del código generado
JET	Con JET se puede generar cualquier archivo de texto, en cualquier formato.
Acceleo	Con Acceleo se puede generar cualquier archivo de texto, en cualquier formato.
	Fácil navegación por los elementos del modelo creado con el editor
JET	La navegación por los elementos del modelo creado con el editor es una tarea de programación.
Acceleo	La navegación por lo elementos del modelo es completamente natural y simple, a través de editores especialmente diseñados para ello.
	Facilidad de uso.
JET	Se requiere leer mucha información y tener muy buenas bases de programación en Java para utilizar la herramienta.
Acceleo	Solo se requieren elementos básicos de programación para crear las plantillas de Acceleo e integrarlo con otras herramientas. Esto se debe a que sigue un enfoque de modelos como GMF.

Tabla 6: Comparación de las herramientas de soporte para la generación de código

De esta forma, se aprecia que Acceleo es una herramienta más evolucionada que JET. Acceleo permite la generación de código a partir de un modelo creado en el editor gráfico generado por GMF.

Para poder utilizar la herramienta apropiadamente se definió un proceso de desarrollo. La figura 35 muestra el proceso de desarrollo definido para construir el generador de código con Acceleo. El proceso presentado comienza con la definición de las reglas de transformación para la aplicación, es decir, qué elementos de código se generarán a partir de los elementos del modelo. Este es un elemento crucial en el desarrollo, ya que el modelo debe contener toda la información necesaria para generar el código, y el código generado debe seguir las especificaciones expresadas de manera visual en el modelo.

El siguiente paso en el proceso de desarrollo es la implementación de las reglas de transformación en plantillas de Acceleo, para después probar que el código que se genere sea correcto.

Por último todo debe ser integrado de la mejor manera posible con el editor desarrollado anteriormente, de manera que se cuente con una herramienta completa MDA. A continuación se presentan en detalle las reglas de transformación definidas para este proyecto.

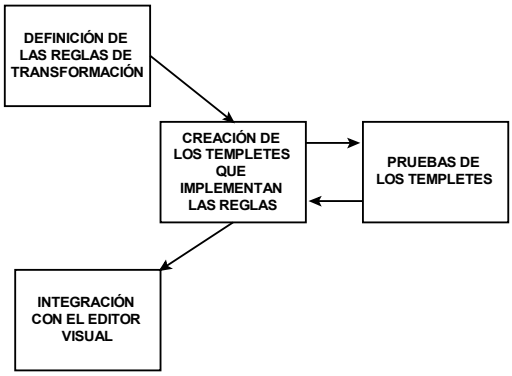

Figura 35: Proceso de desarrollo establecido para la generación de código

5.1.4. Definición de las reglas de transformación para los servicios declarativos

Recordando, una regla de transformación es el mapeo de los elementos del modelo al código que será generado, especificando qué código será generado con respecto cada uno de los elementos del modelo. En el caso de los servicios declarativos la herramienta generará básicamente los siguientes archivos:

a) **Descriptor.xml.** Archivo que contiene la especificación declarativa del componente de servicio de acuerdo a la especificación de los servicios declarativos.

b) **Manifest.mf.** Debido a que el *bundle* estará empaquetado en un archivo *jar*, se requiere un archivo *manifest* dentro del paquete que contiene toda la información de los archivos dentro del *jar*.

c) **Interfaz del componente de servicio.** Esqueleto del código en Java de la interfaz de java que el componente utiliza para exponer su funcionalidad.

d) **Clase de implementación.** Esqueleto del código en Java de implementación del servicio ofrecido por el componente.

Se debe establecer una regla de transformación para cada elemento del modelo, es decir, para cada propiedad establecida en el perfil UML. En la tabla 6 se presentan todas las reglas de transformación definidas. Se puede apreciar que una regla de transformación no hace validación sobre los elementos del modelo, sólo define qué fragmento de código será generado para cada elemento del modelo creado con el editor visual. Por ejemplo, si un nodo de tipo componente se le asigna en su propiedad *immediate* el valor de "true" en el modelo, entonces el generador de código deberá crear el código

```
immediate = "true"
```

dentro del archivo descriptor.xml. En la tabla se aprecian dos columnas, la primera representa el elemento del modelo a partir del cual se generará el código que se describe en la segunda columna. Se utiliza el operador punto para acceder a los elementos del modelo, por ejemplo, *Component.immediate* se refiere a la propiedad *immediate* del elemento de tipo *Component* en el modelo.

También es importante destacar que algunas partes del código generado se crean independientemente del modelo, es decir, son siempre iguales de componente en componente y no dependen de los elementos definidos en el modelo. Estos elementos pueden verse en las plantillas de Acceleo que se muestran en el apéndice III.

Elemento del modelo	Código generado
Component	**descriptor.xml** <?xml version="1.0" encoding="UTF-8"?> <component </component> **Manifest.mf** Manifest-Version: 1.0 Bundle-ManifestVersion: 2 Bundle-Version: 1.0.0 Service-Component: metaINF/descriptor.xml Bundle-Localization: plugin Import-Package: org.osgi.framework;version="1.3.0"
Component.name	**descriptor.xml** name="Component.name" **Manifest.mf** Bundle-Name: Component.name Bundle-SymbolicName: Component.name
Component.delayed = true	-
Component.enabled = true	**descriptor.xml** enabled="true"
Component.immediate = true	**descriptor.xml** immediate="true"
Component.factory <> vacio	**descriptor.xml** factory="Component.factory"

Component.implementationClass	**descriptor.xml**
	<implementation class="Component.implementationClass"/>
	
	Implementación de la clase en Java
	package Component.implementationClass;
	
	import org.osgi.framework.BundleContext;
	
	public class Component.implementationClass {
	
	public void activate(BundleContext ctxt) {
	}
	}

Component.Reference	**descriptor.xml** <reference > </reference>
Component.Reference.bindMethod	**descriptor.xml** bind= "Component.Reference.bindMethod"
Component.Reference.unbindMethod	**descriptor.xml** unbind= "Component.Reference.unbindMethod"
Component.Reference.cardinality	**descriptor.xml** cardinality=”Component.Reference.cardinality”
Component.Reference.policy	**descriptor.xml** policy="Component.Reference.policy"
Component.Reference.targetFilter	**descriptor.xml** target= "Component.Reference.targetFilter"
Component.Reference.targetReference.interface	**descriptor.xml** interface="Component.Reference.targetReference.interface" **Manifest.mf** Import- Package:Component.Reference.targetReference.interface;
Component.Reference.targetReference.name	**descriptor.xml** name="Component.Reference.targetReference.name"

Elemento del modelo	Código generado
Component.Provide	**descriptor.xml** <service> <provide /> </service>

Component.Provide.targetProvide.interface	**descriptor.xml**
	interface="Component.Provide.targetProvide.interface"
	
	Interfaz de Java
	import org.osgi.framework.BundleContext;
	
	public interface Component.Provide.targetProvide.interface {
	
	abstract void activate(BundleContext ctxt);
	// Methods that are provided by this interface
	}
	
	Implementacion de la clase en Java
	implements Component.Provide.targetProvide.interface {

Component.Declares.targetDeclares	**descriptor.xml** <property />
Component.Declares.targetDeclares.name	**descriptor.xml** name="Component.Declares.targetDeclares.name"
Component.Declares.targetDeclares.value	**descriptor.xml** value="Component.Declares.targetDeclares.value"
Component.Declares.targetDeclares.type	**descriptor.xml** type="Component.Declares.targetDeclares.type"
Component.Declares.targetDeclares.body	**descriptor.xml** Component.Declares.targetDeclares.body
Component.Declares.targetDeclares.entry	**descriptor.xml** <properties entry="Component.Declares.targetDeclares.entry"/>

Tabla 7: Reglas de transformación del modelo al código para los servicios declarativos

Una vez que se estableció qué es lo que será generado por el editor visual a partir de cada elemento del modelo, el siguiente paso fue implementar todas estas reglas de transformación en plantillas de Acceleo. Se crearon 4 plantillas, una para cada archivo que sería generado: el descriptor.xml, la interfaz del servicio en Java, la implementación del servicio en Java y el archivo de configuración manifest.mf.

El detalle completo de la estructura de estas plantillas y su funcionamiento puede consultarse en el apéndice III.

El siguiente capítulo tiene como finalidad presentar los resultados del proyecto: el proceso de desarrollo definido para la creación de herramientas MDA y la herramienta MDA construida para los servicios declarativos.

6. Resultados

6.1. Herramienta MDA para la creación de bundles siguiendo el enfoque de los servicios declarativos

Se estableció un proceso para definir la manera en que iba a ser distribuida y entregada a los usuarios finales. En cuanto a la distribución, la aplicación esta dividida en dos partes: el editor visual y el generador de código. El editor visual depende del *framework* de GMF para funcionar, así que la distribución lo incluye. El generador de código depende del *plugin* de Acceleo así que éste también fue incluido en la distribución. Además, deben incluirse las plantillas para la generación de código y el modelo de cadena de transformación citados en el apéndice III. La herramienta entonces se distribuye en dos partes, por una parte en un archivo jar los *frameworks* de soporte (GMF y Acceleo) y por otro lado el *plugin* del editor visual con las plantillas y librerías.

La siguiente sección tiene la finalidad de presentar al lector los resultados finales de este proyecto de investigación y desarrollo, para esto, se presenta un ejemplo sencillo desarrollado sobre la herramienta MDA que fue construida, de manera que se pueda apreciar su funcionamiento.

6.1.1. Utilización de la herramienta MDA

Para comenzar se debe crear un proyecto y dentro de éste una carpeta con el nombre *model*, donde se colocarán nuestros modelos. La figura 39 muestra el proyecto creado, en este caso mx.izt.uam.ejemplo.

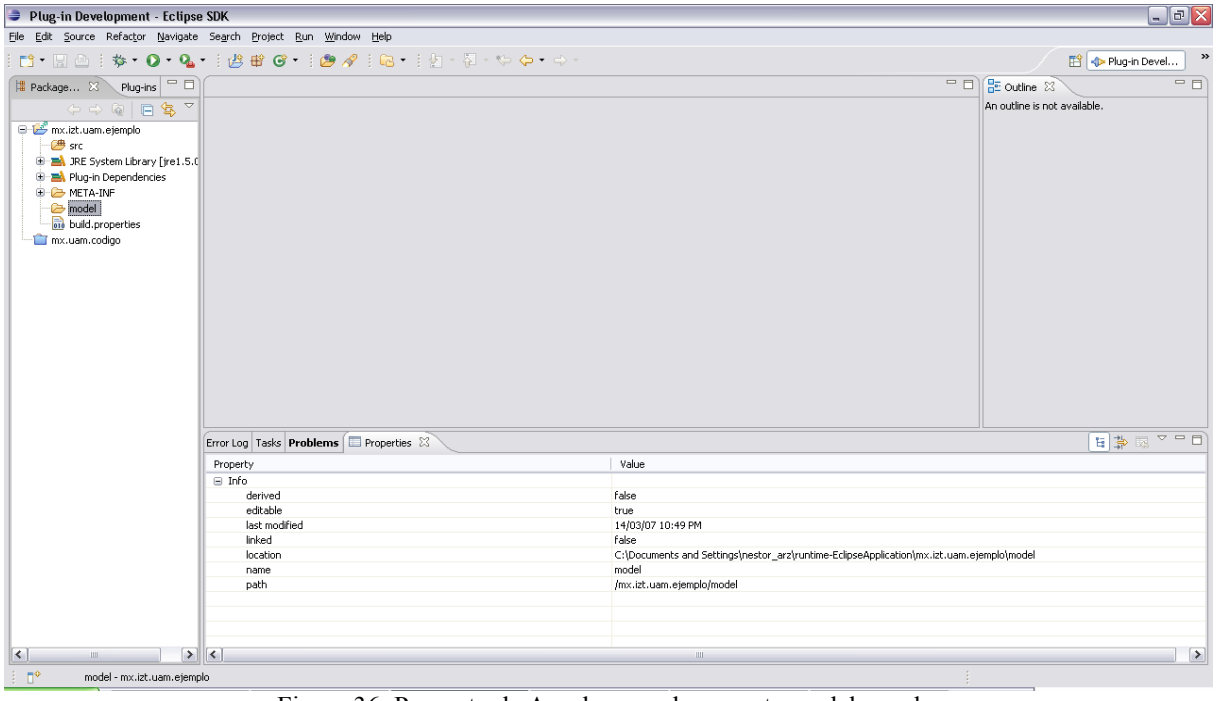

Figura 36: Proyecto de Acceleo creado, carpeta model creada.

Se crea un modelo utilizando el editor visual, la herramienta muestra un asistente para la creación del éste diagrama (DS Diagram).

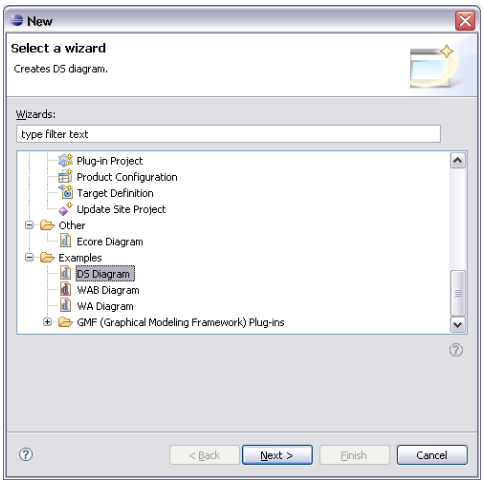

Figura 37: Asistente para la creación de un nuevo modelo.

Se dibuja el diagrama completo indicando todas las propiedades necesarias y validándolo posteriormente para asegurar que cumpla con las especificaciones. La validación la realiza la misma herramienta, presentando en caso de existir, los errores del modelo.

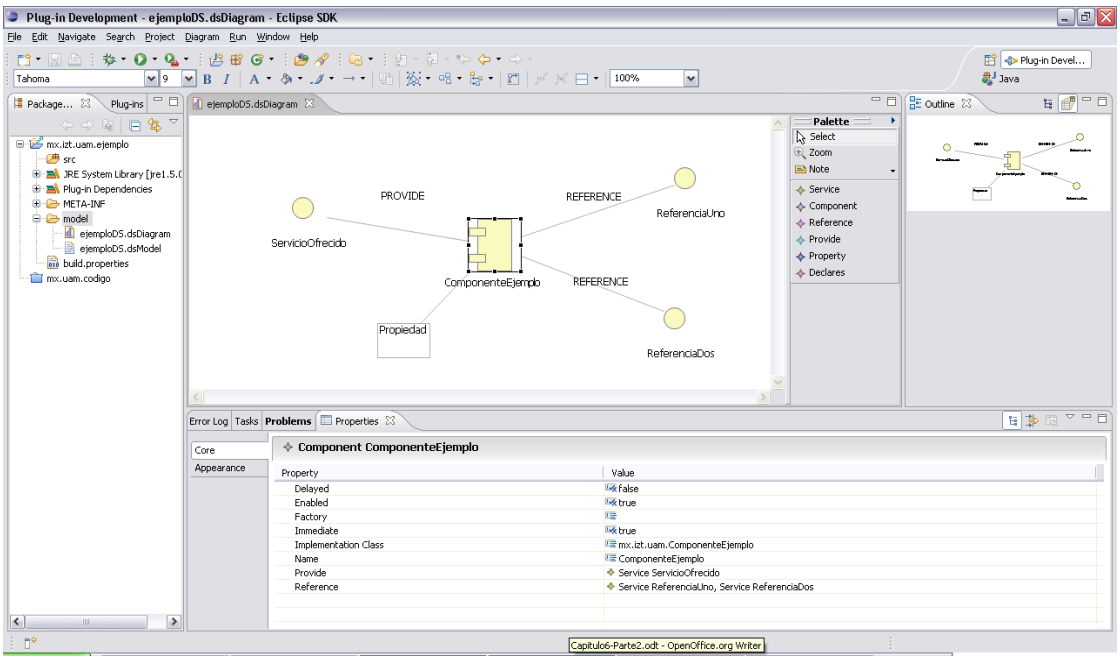

Figura 38: Ejemplo de modelo creado utilizando el editor gráfico

Una vez con el modelo creado el siguiente paso es la generación de código. Para hacer esto es necesario importar el paquete con las plantillas, y modelo de cadena de transformación a la carpeta *src* de nuestro proyecto. Este paquete forma parte de la distribución de la herramienta.

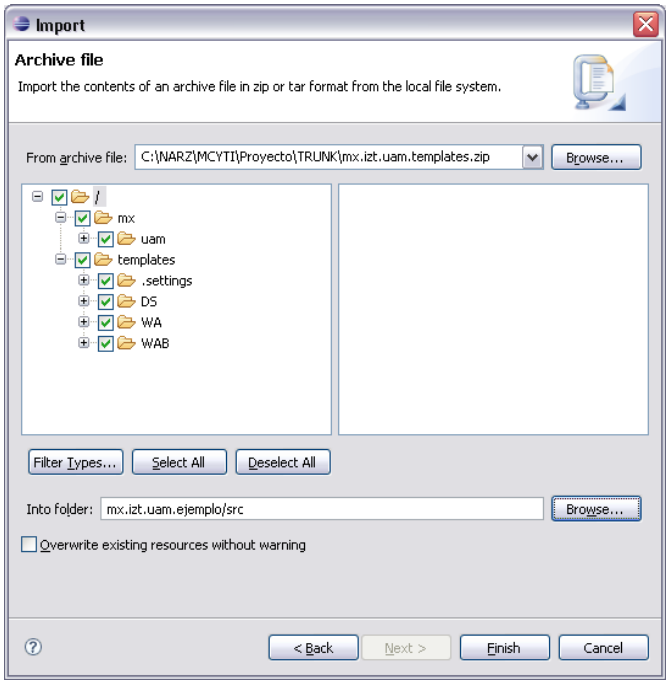

Figura 39: Importar paquete con las plantillas y el modelo de cadena

La siguiente figura muestra las plantillas, librería y modelo importados en nuestro proyecto.

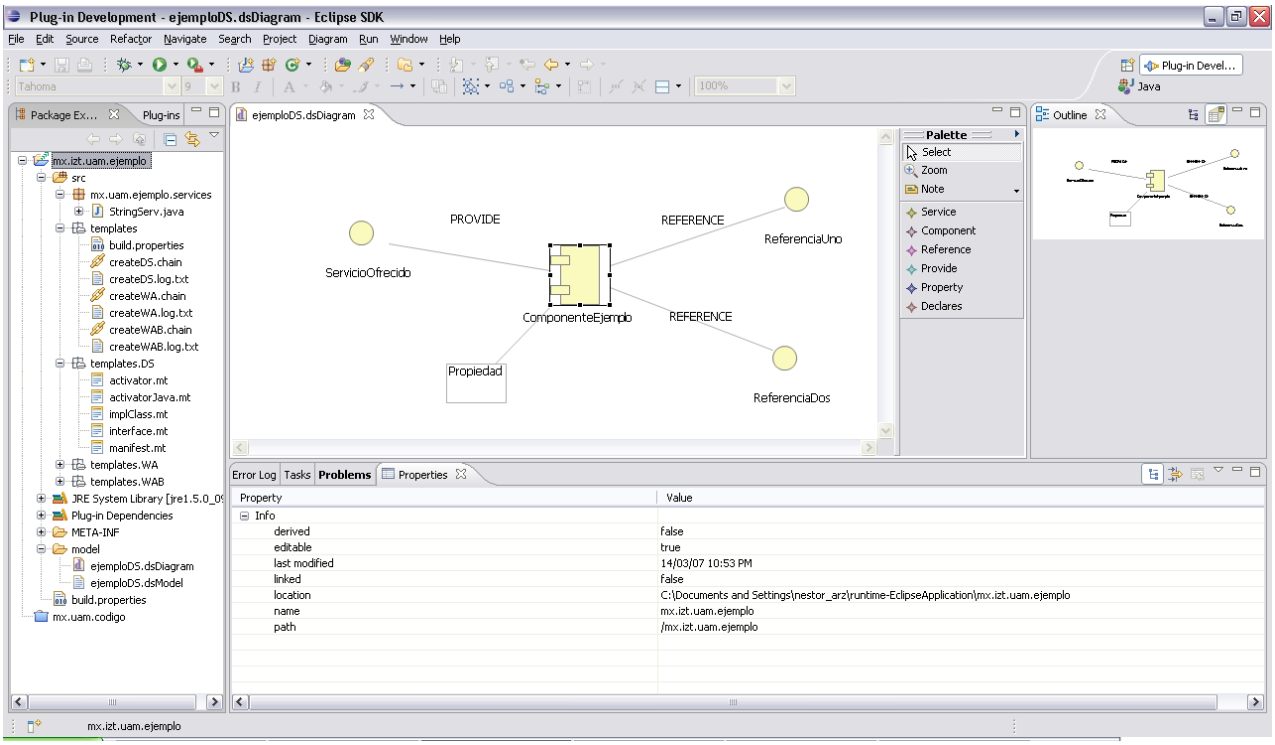

Figura 40: Proyecto con plantillas y el modelo de cadena importados

Para poder generar código es necesario modificar los parámetros del modelo de cadena de transformación, de manera que concuerde con los datos del proyecto en Eclipse. Para esto debe crearse un nuevo proyecto donde quedarán los archivos que serán generados, en nuestro caso particular, se crea un proyecto vacío de tipo *plugin* llamado mx.izt.uam.

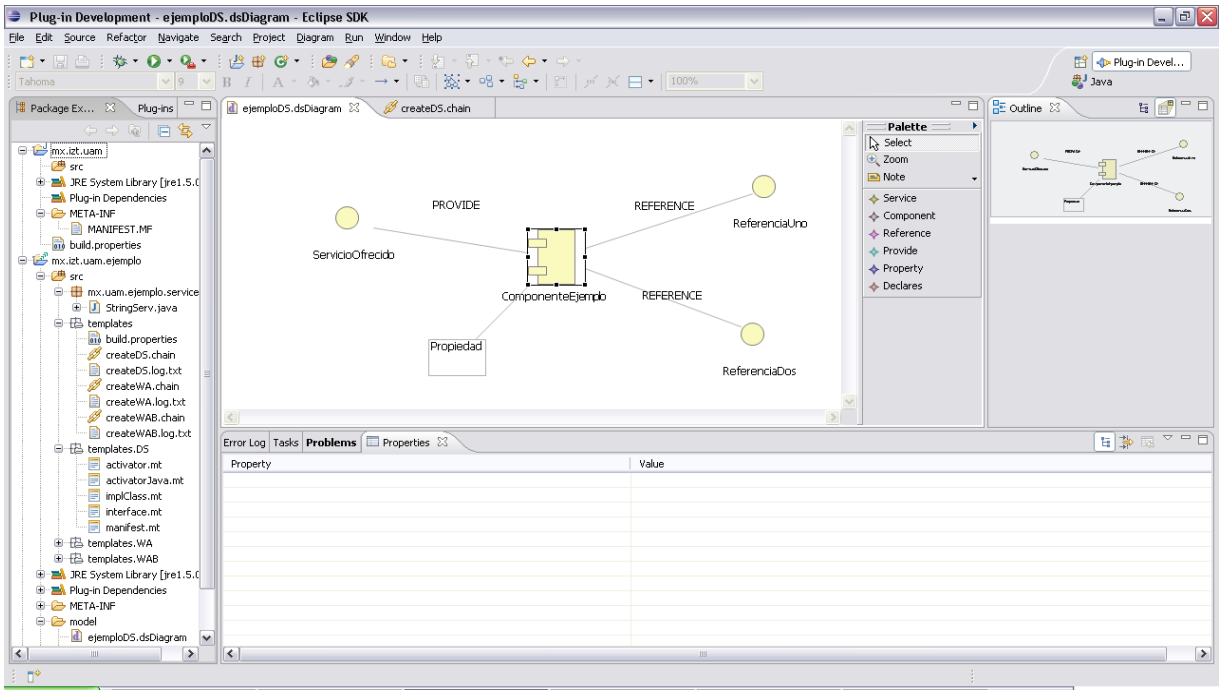

Figura 41: Proyecto mx.izt.uam para crear el código generado

Después de realizar las modificaciones necesarias en el modelo de cadena (que básicamente solo es redefinir las rutas donde ahora se encuentran las plantillas y el diagrama además de las rutas de las carpetas donde se generará el código) se obtiene el resultado mostrado en la figura 42

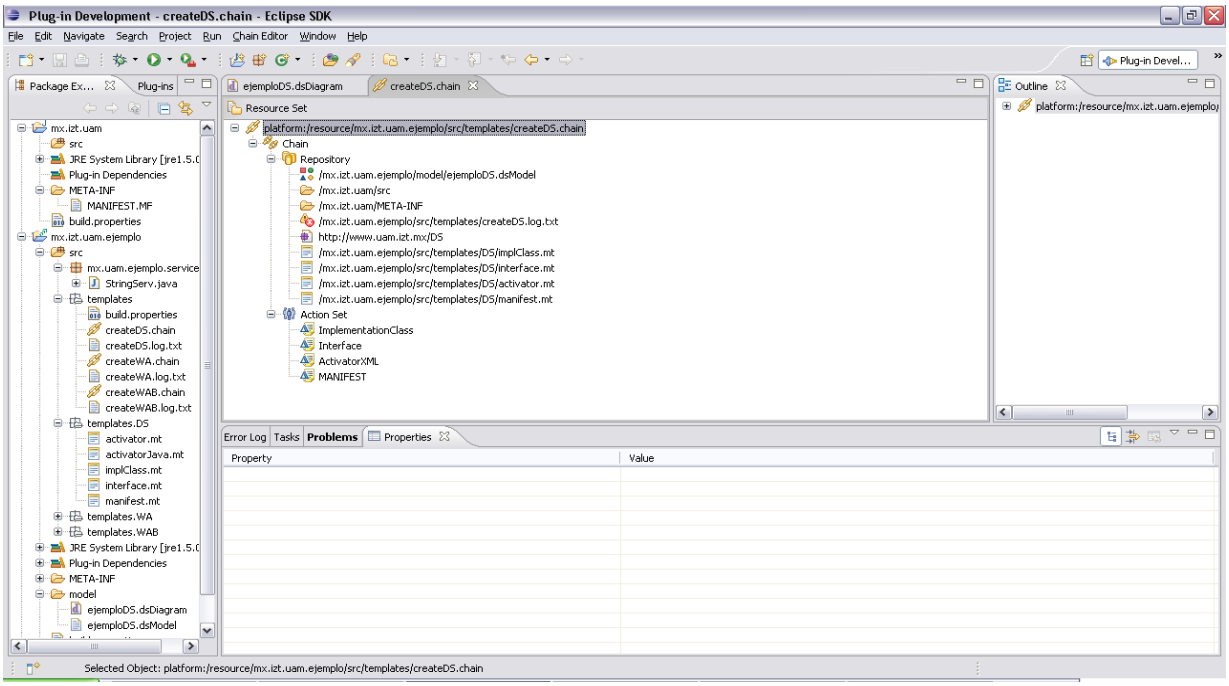

Figura 42: Modelo d cadena modificado para apuntar a los elementos adecuados

Ahora sólo resta ejecutar la cadena de transformación.

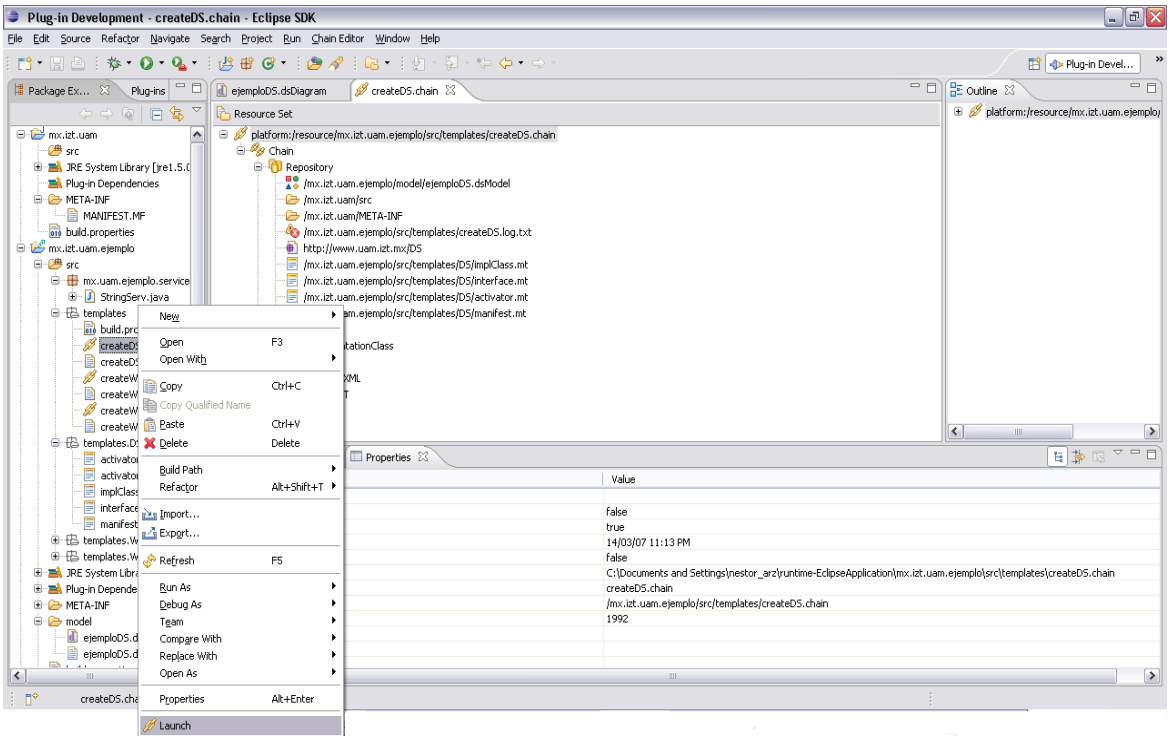

Figura 43: Ejecución de la cadena de transformación

El código es generado en el proyecto mx.izt.uam. La figura 44 muestra el descriptor generado para nuestro ejemplo, la figura 45 muestra la clase de implementación y la figura 46 muestra la interfaz.

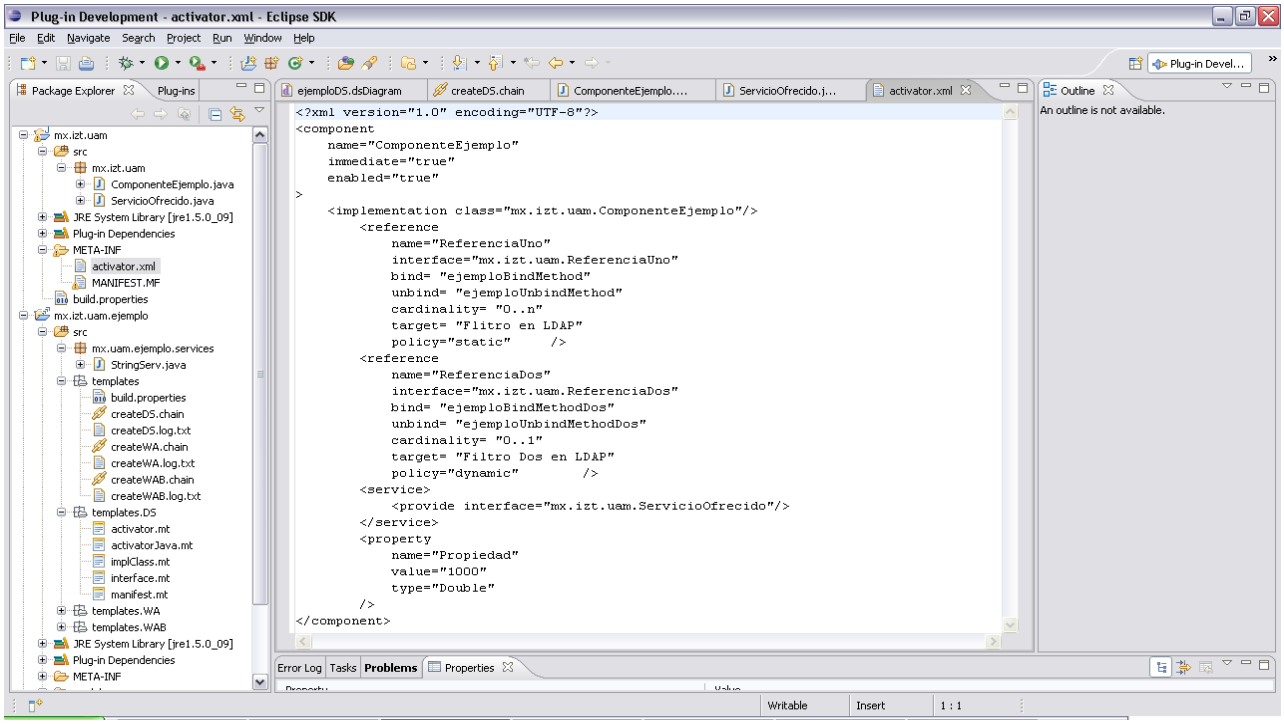

Figura 44: Descriptor generado para el ejemplo de la figura 38

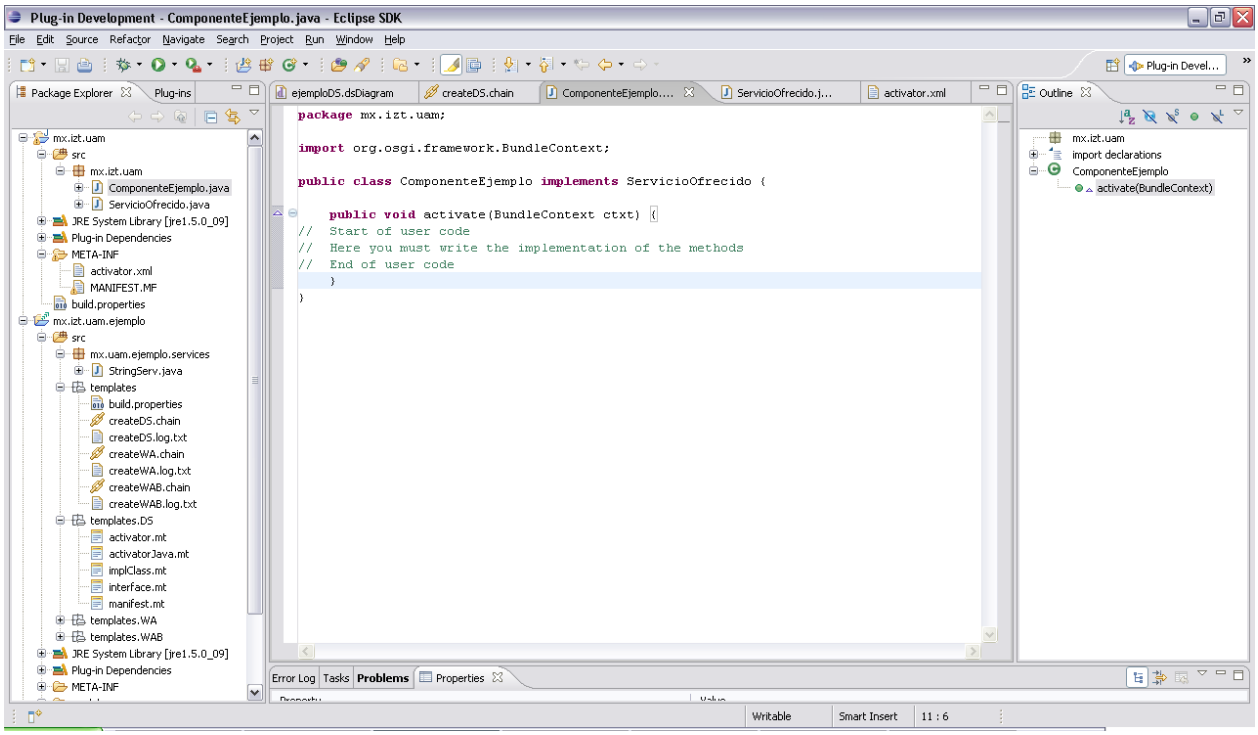

Figura 45: Clase de implementación generada para el ejemplo de la figura 45

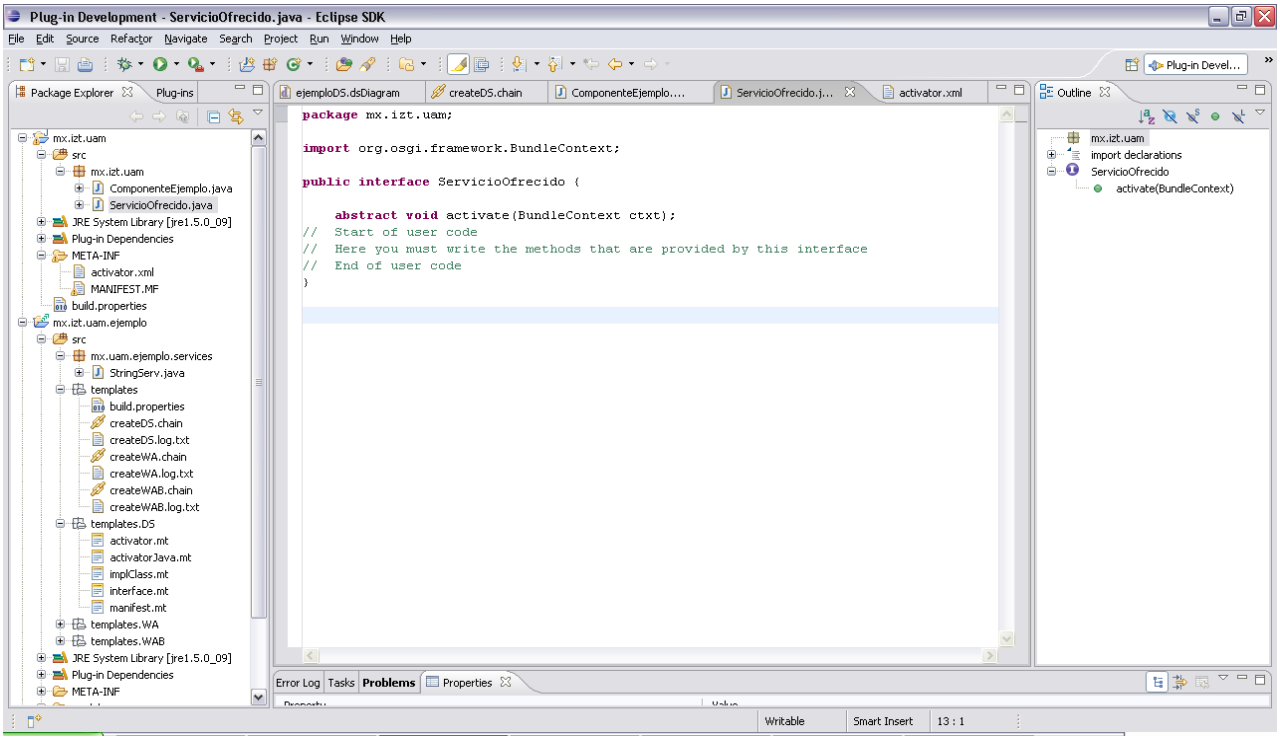

Figura 46: Interfaz generada para el ejemplo de la figura 45

De esta forma se pueden generar todos los archivos requeridos para un *bundle* de OSGi siguiendo el enfoque de los servicios declarativos en cuestión de minutos, y en el caso especial del archivo descriptor del componente en XML, el desarrollador no tiene que modificar ni agregar ninguna línea de código, lo que acelera el proceso de desarrollo de *bundles.* Esto se debe a que el descriptor sólo maneja aspectos no funcionales del componente.

Como punto final se muestra de nuevo la tabla comparativa de las herramientas existentes en el mercado que se encontraron al momento de redacción de este trabajo y comparándolas con la herramienta MDA construida en este proyecto.

Aplicación	***PSM a Código de cualquier plataforma***	***PIM a PSM***	***Editor de metamodelos***	***Editor de reglas de transformación***	***Fácil adaptación a un dominio en particular***	***Comercial***	***Dificultad de uso y aprendizaje***
MDA para DS	***SI***	***NO***	***SI***	***SI***	***SI***	***NO***	***Baja***
UVM-QVT	NO	SI	SI	SI	NO	SI	Alta
OpenMDX	NO	SI	SI	SI	NO	SI	Alta
AndroMDA	SI	SI	SI	SI	NO	SI	Alta
EMF	NO	NO	SI	SI	NO	NO	Media
Acceleo	SI	NO	NO	SI	SI	NO	Baja

Tabla 8: Comparación final de la herramienta construida vs las encontradas en la investigación

Se puede apreciar que la herramienta construida no genera PSM's de PIM's sin embargo es importante comentar que, debido que Acceleo puede generar cualquier tipo de archivo de texto basándose en un modelo, podría generarse una herramienta en la cual se crearan PIM's y tranformarlos a PSM's en formato XMI. Este trabajo estuvo limitado a una plataforma específica, OSGi, y por tanto no se hizo trabajo relacionado con éste punto, pero debe notarse que con el enfoque seguido puede crearse también una herramienta de transformación de PIM's a PSM's.

6.2. Proceso de desarrollo de herramientas MDA

Como parte del proyecto un proceso de desarrollo de herramientas MDA fue definido, éste proceso puede ser reutilizado por cualquier organización de desarrollo de software para crear herramientas que industrialicen algunos de sus procesos de desarrollo acelerando la construcción de software. El proceso es independiente de la plataforma, no es exclusivo para aplicaciones orientadas a servicios, así que, herramientas MDA para distintos dominios y necesidades pueden ser creadas.

El proceso fue descrito implícitamente a lo largo de éste documento y se presenta en la figura 47. La primer fase del proceso consiste en la creación del perfil UML basados en el metamodelo del dominio. Las siguientes dos fases están relacionadas con la implementación del editor visual que utilice el perfil UML como lenguaje visual y use las reglas en OCL para realizar la validación de los modelos generados con el editor.

El generador de código será desarrollado en las dos últimas fases. Primero, se definen las reglas de transformación y posteriormente éstas son implementadas en una herramienta de software que se integre con el editor, de manera que se obtiene una herramienta MDA completa.

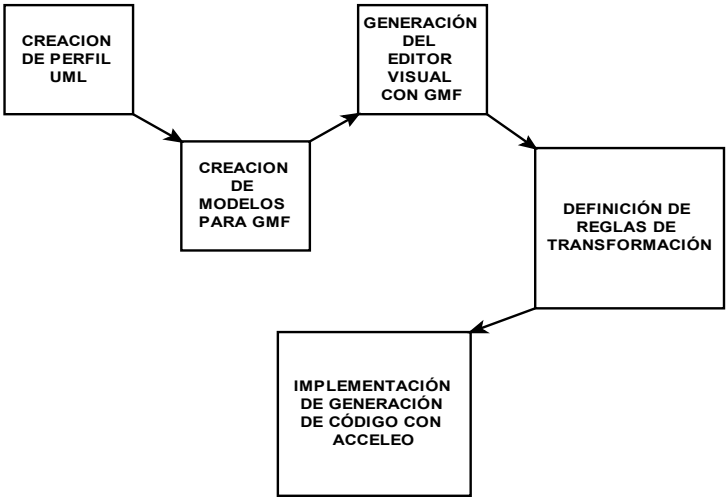

Figura 47: Proceso de desarrollo de herramientas MDA propuesto

Existen varias razones por las cuales no se han adoptado enfoques MDA en el desarrollo de software en la actualidad. El costo de las herramientas existentes es bastante alto, y adaptarlas a las necesidades propias de cada proyecto es muy complejo o imposible, sumado a estos problemas está el hecho de que son herramientas complejas, difíciles de utilizar.

Se estableció un proceso utilizando herramientas *open source* para la creación de herramientas MDA a la medida. Dependiendo de la complejidad de la herramienta, el desarrollo de ésta puede llevar entre 2 y 7 días, obteniendo una herramienta fácilmente extensible, completamente personalizada a nuestras necesidades y basada en tecnologías y estándares ampliamente aceptados. El proceso fue probado en el desarrollo de dos herramientas más, ambas construidas en un lapso menor a una semana (véase Apéndice IV).

Este abre un camino para que herramientas MDA sean cada vez más complejas y completas puedan ser creadas con mayor facilidad, herramientas que en un futuro ayuden a industrializar el proceso de desarrollo de software.

7. Conclusiones finales

Al inicio de este proyecto se tenían planteados 3 objetivos principales: la definición de un lenguaje de modelado de aplicaciones basadas en componentes orientadas a servicios, la construcción de una aplicación que acelerara el desarrollo de éste tipo de aplicaciones y por último la definición de un proceso fácilmente replicable para la creación de aplicaciones MDA para otros dominios.

Se definió un lenguaje de modelado visual basado en un perfil UML para el desarrollo de *bundles* haciendo uso del enfoque de los servicios declarativos. Posteriormente se realizó una investigación sobre herramientas de soporte que nos ayudaran a desarrollar un editor visual dónde crear modelos basados en éste lenguaje visual, validarlos contra la especificación (mediante las reglas en OCL del perfil UML) y generar el código del *bundle*.

Se evaluó la herramienta haciendo pruebas sobre todas las reglas contenidas en la especificación verificando que todas fueran validadas de manera correcta por la herramienta, asegurando con esto que el código generado estuviera libre de errores semánticos.

Se estableció un proceso replicable para la creación de herramientas MDA. Este proceso fue utilizado para la creación de dos aplicaciones más, de manera que el proceso fue probado. Las nuevas herramientas fueron creadas con éxito en un tiempo muy corto (véase apéndice IV). De esta forma una organización de desarrollo de software puede tomar el proceso definido para crear sus propias herramientas MDA de acuerdo a sus necesidades. Desde esta perspectiva, la solución propuesta, es un punto de partida para la adopción de MDA en un mayor número de organizaciones dedicadas al desarrollo de software.

7.1. Comparación con otros trabajos similares

Se encontraron durante este trabajo de investigación muchos esfuerzos para lograr introducir MDA en los procesos de desarrollo de software, sin embargo muchas de las herramientas están enfocadas a plataformas específicas y es complicado o imposible adaptarlas a un dominio en particular.

Durante la investigación se encontró solamente un trabajo similar, pero este no se refiere a una herramienta MDA enfocada al desarrollo de componentes orientados a servicios. Lo encontrado fue un trabajo desarrollado en la Universidad Politécnica de Valencia, en España [41], en el cual se plantea un metamodelo completo para OSGi, y a futuro la creación de herramientas MDA basadas en su metamodelo, sin embargo, al momento de redactar esta tesis, no existe ninguna implementación de su solución.

Cabe señalar que como parte de este proyecto, se realizó una estancia en la Universidad Joseph Fourier en Grenoble, Francia, donde se tuvo la oportunidad de observar el trabajo de otros investigadores en el desarrollo de herramientas para crear herramientas MDA. Se observó que a diferencia de el presente proyecto, donde se hizo uso de *frameworks* existentes para crear nuestra solución, en Grenoble los investigadores estaban construyendo soluciones prácticamente desde cero, complejas de manejar y adaptar. Por experiencia personal, en la industria actual de desarrollo de software se tiene una visión mucha mas práctica, sólo se modifican los procesos cuando las modificaciones facilitan o aceleran considerablemente el desarrollo de aplicaciones, sin llegar a ser complejas ni difíciles de implementar.

7.2. Perspectivas

Durante la fase final de este trabajo de tesis, se observó la posible continuidad de este trabajo en el desarrollo de una herramienta MDA completa, que no sólo genere el esqueleto de algunos componentes del código de la aplicación. El enfoque declarativo está siendo adoptado ampliamente en la actualidad, un ejemplo de ello es un *framework* llamado Spring [42] con el cual se pueden construir la base arquitectónica de una aplicación de forma declarativa de manera similar a como funcionan los servicios declarativos.

Utilizando el proceso definido, generar un archivo en formato XML o descriptor que represente los aspectos no funcionales de la aplicación, no representa ningún problema, si existiesen más *frameworks* como Spring que se encarga de tomar este descriptor y convertirlo en una aplicación funcional, entonces MDA sería completo, al menos en el aspecto no-funcional, esto es debido a que el desarrollo de los aspectos no funcionales de las aplicaciones es mas proclive a ser automatizado. La herramienta desarrollada absorbería toda la complejidad relacionada con la construcción del descriptor de la aplicación, validando todas las reglas utilizando OCL, de acuerdo a las especificaciones de Spring.

Nuestra herramienta se planea será liberada como *open source* para la comunidad OSGi y MDA, de manera que cualquier persona que desee explotar nuestras ideas y extenderlas pueda hacerlo, de manera que el esfuerzo realizado en la construcción de la herramienta y la definición del proceso de desarrollo de la misma pueda ser retomado por otros proyectos y, de esta forma, poder contribuir a que la idea de MDA pueda ser adoptada en un futuro como parte fundamental de un proceso de desarrollo de software.

MDA ha sido desde su concepción una idea que ha quedado siempre más del lado de los sueños que de la realidad, sin embargo con la evolución de las herramientas de desarrollo actuales y el impresionante avance tecnológico en tecnologías MDA, ésta se encuentra cada vez más cerca de volverse una realidad. Este proyecto significa un paso más en el proceso de adopción de MDA como parte del proceso de desarrollo de software en la actualidad y se espera pueda contribuir al lanzamiento de un nuevo proceso de desarrollo basado en herramientas de este tipo.

8. Apéndice I: Modelos GMF

A continuación se presenta un resumen de los modelos GMF que fueron creados para la creación del editor visual para los servicios declarativos explicando en detalle la estructura y funcionamiento de cada uno de ellos.

8.1. Modelo del dominio

Para el desarrollo del editor visual se comenzó por la definición del llamado modelo del dominio. En nuestro caso este modelo es el perfil UML. El modelo del dominio es un modelo EMF [28], almacenado en un archivo con formato XMI (con extensión .ecore). Este modelo puede ser creado de varias maneras:

a) Creando el modelo en UML en alguna herramienta de modelado externa (como *StarUML* o *Rational Rose*) y luego importando el modelo utilizando las facilidades que nos da para esto EMF.

b) Creando el esqueleto de código en Java que represente los elementos de nuestro dominio y luego importando el modelo utilizando EMF.

c) Creando el modelo directamente en un editor tipo árbol que nos ofrece EMF.

d) Creando el modelo directamente en un editor visual que nos ofrece GMF.

La forma más simple de trasladar el metamodelo a EMF es crearlo utilizando el editor visual de GMF o el editor de modelos de árbol de EMF. La figura 48 muestra un ejemplo de estos modelos. Ambos modelos muestran la misma información y por tanto uno puede ser generado a partir del otro [24]. Si se tienen ambos y se realiza una modificación a solo uno de ellos GMF se encarga de mantenerlos sincronizados actualizando los cambios en el otro modelo.

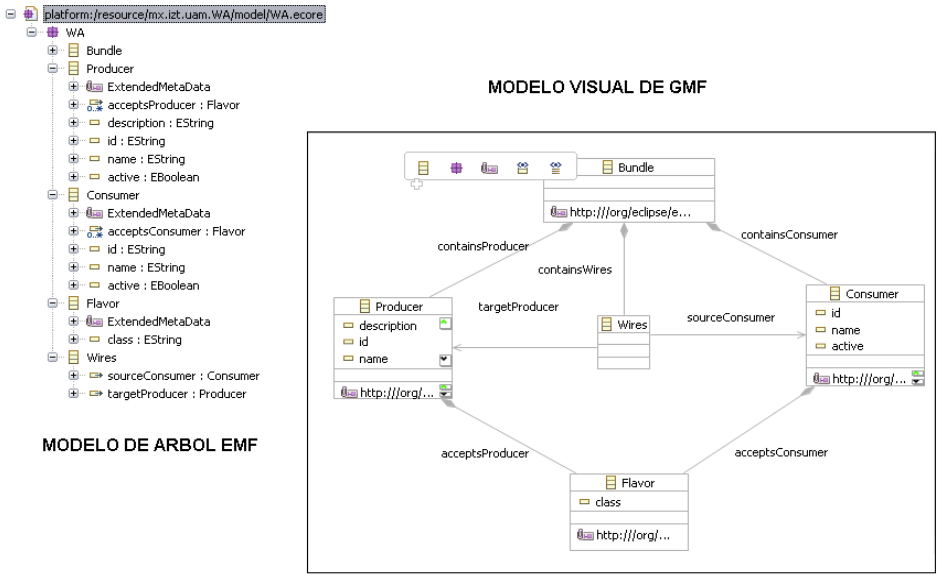

Figura 48: Modelos EMF y GMF para el dominio.

Varios elementos de EMF son utilizados para la definición de éste modelo:

a) ***EClass***. Representa una clase.

b) ***EEnum***. Representa una enumeración.

c) ***EOperation***. Representa un método de una *EClass*.

d) ***EAttribute***. Representa un atributo de una *EClass*.

e) ***EReference***. Representa una relación de una *EClass* con otra.

A continuación en la figura 49 se presenta el modelo del dominio generado en este proyecto. Al compararlo con el perfil UML mostrado en la figura 26, se puede notar que son prácticamente idénticos, sin embargo, existen ciertas diferencias. Se aprecia un paquete que contiene a todas las *EClass* llamado DS (*Declarative Services*). El nombre seleccionado para el paquete posteriormente es utilizado por el *framework* para colocarlo como prefijo en el nombre de todos los elementos generados. Una de las propiedades importantes que se tienen que definir para el paquete DS es *NsURI*, esta define una URI (*Uniform Resource Identifier*) que será utilizada por otras herramientas para hacer referencia al modelo de dominio, en éste caso tiene un valor de http://www.uam.izt.mx/DS.

En el modelo del dominio el elemento central es *Bundle,* éste elemento no aparece en el perfil UML, pues es exclusivo para GMF. A éste elemento *Bundle* se le llama contenedor y representa al modelo (o diagrama), como tal, contendrá a cualquiera de los elementos visuales dentro del modelo. Por ser un contenedor tiene varias relaciones *EReference* de tipo composición a cada uno de los elementos del modelo que podrán ser dibujados en el diagrama. Para indicar que cualquiera de éstas relaciones es de tipo contención, se usa la propiedad *Containment* que por omisión tiene el valor *false.*

En la misma figura 49 se aprecia una *EClass* por cada elemento definido en el perfil UML: *Component*, *Service*, *Provide*, *Reference*, *Property* y *Declares.* Se aprecian *EAtributes* iguales a los valores etiquetados definidos en el perfil para cada elemento, y una *EEnumeration* para la definición de cada uno de las enumeraciones que definen los tipos *JavaTypes*, *Policy* y *Cardinality* del perfil. A cada *EAttribute* se le asigna un tipo de dato específico dentro de los definidos por EMF (*EString*, *EInt*, *EDouble*, *EBoolean*, etc.).

Component posee dos relaciones *EReference: Provide* y *Reference.* Éstas se utilizan para relacionar un componente de servicio con el servicio que provee y con los servicios que requiere respectivamente. Se debe definir para cada *EReferences* la cardinalidad, de manera que se restrinja el numero de elementos pueden conectarse con un elemento en particular, en este caso: *Component.* Para esto, cada *EReference,* posee dos propiedades llamadas *Lower Bound* y *Upper Bound.* En perfil UML existen restricciones en OCL que tienen que ver con la cardinalidad de las relaciones entre los distintos elementos del perfil y algunas de ellas fueron consideradas al definir la cardinalidad.

Para cada tipo de relación (*Declares, Provide y Reference*) se agregaron además dos relaciones: *target* y *source.* Para el caso de *Reference* éstas se llaman *sourceReference* y *targetReference* e indican cual será el extremo origen (*source*) y el extremo destino (*target*)en una relación de tipo *Reference*, en éste caso se puede notar que para una asociación de tipo *Reference* el extremo origen es un *Component* y el extremo destino es un *Service*. Lo mismo sucede con las relaciones *Provide* y *Declares*. Una diferencia más entre el modelo del dominio y el perfil UML es la enumeración *EEnumeration RelationshipType* la cual se utiliza para indicar los tres tipos de asociación que existen (*Declares, Provide y Reference*). Esto fue utilizado posteriormente para validar ciertas restricciones en OCL.

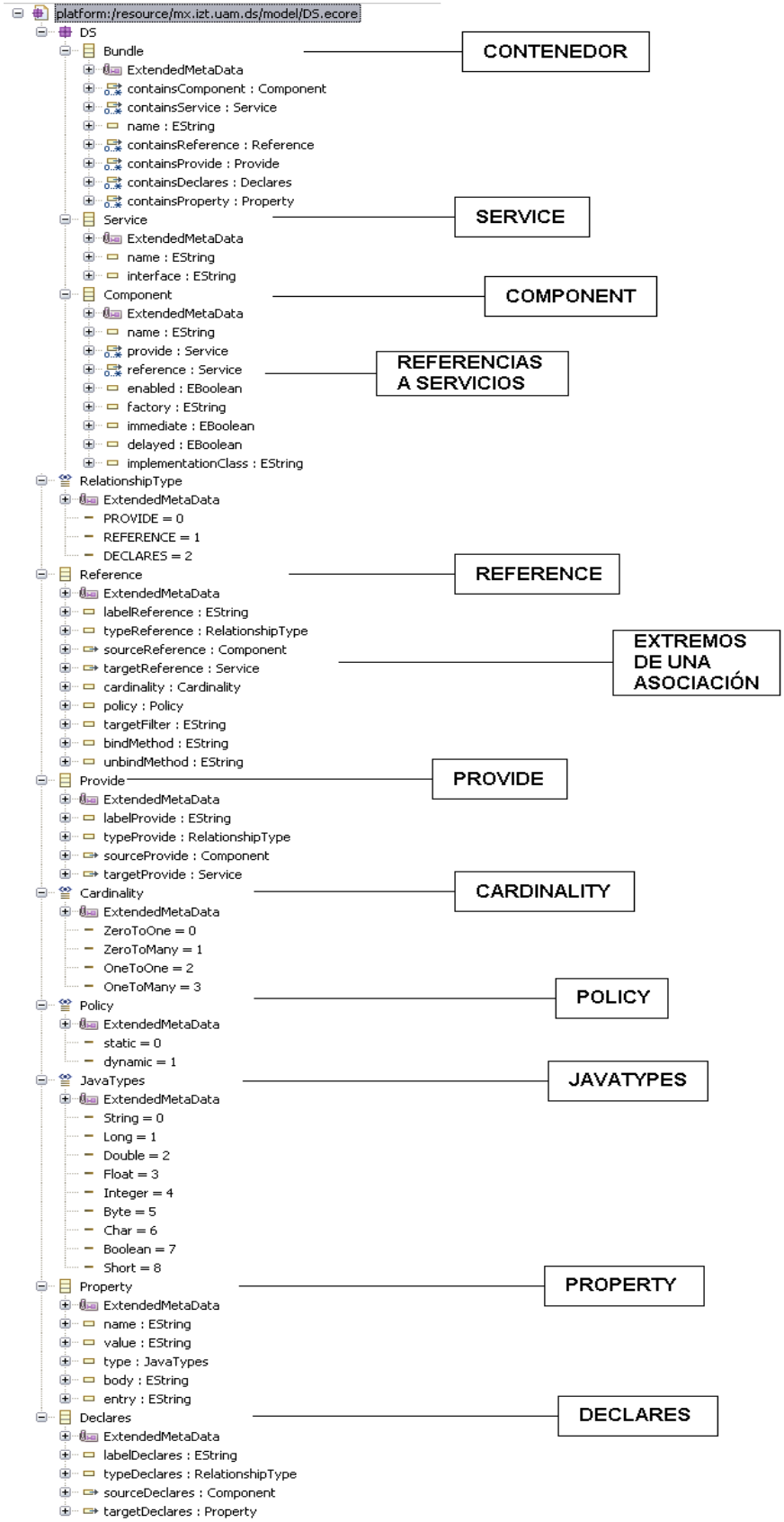

Figura 49: Modelo EMF del dominio para los servicios declarativos basado en el perfil UML

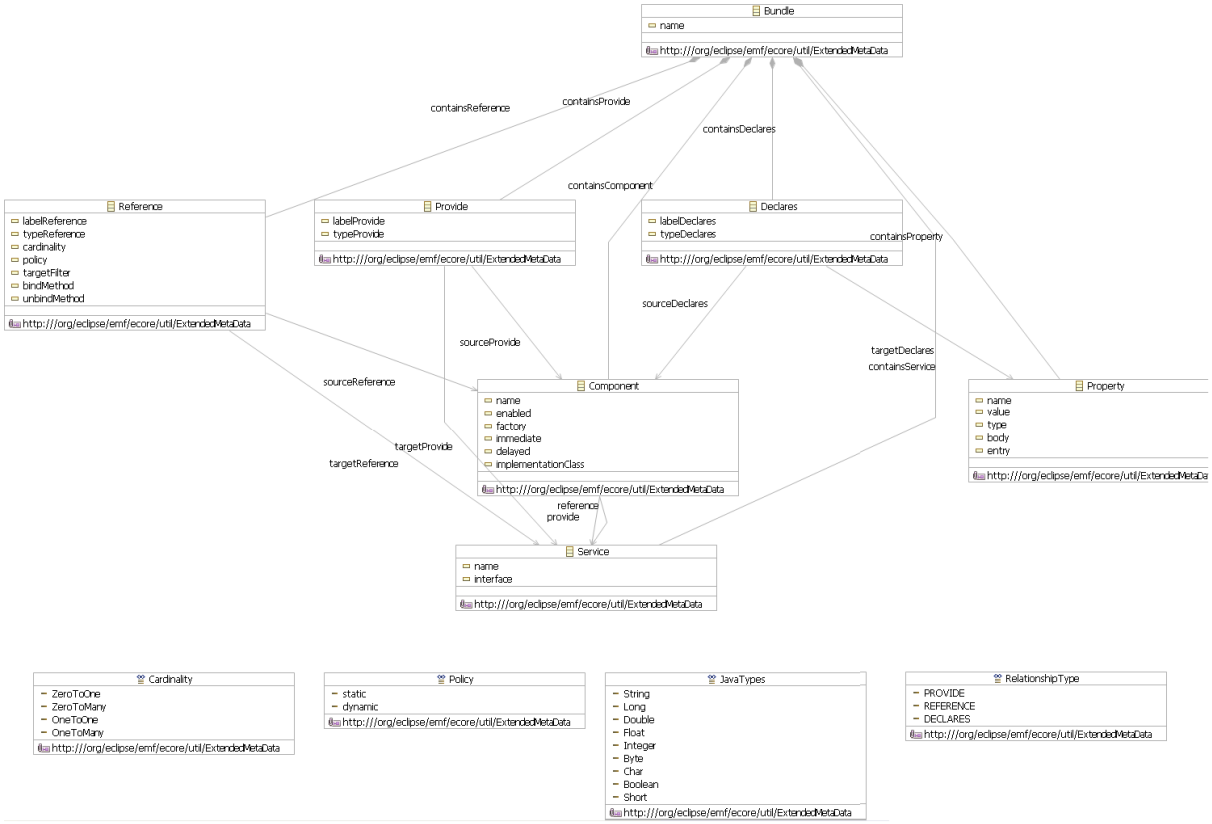

Figura 50: Modelo GMF del dominio para los servicios declarativos

8.2. Modelo de definición gráfica

El modelo del dominio indica qué elementos serán representados de manera visual, sus relaciones, valores etiquetados y propiedades. Sin embargo para presentarlos en pantalla se tiene que definir el comportamiento y aspecto que tendrán éstos elementos en el editor visual. El modelo de definición gráfica (con extensión gmfgraph) define los elementos visuales utilizados para representar los elementos del dominio en el editor gráfico. En éste modelo se definen elementos visuales como son nodos, ligas, figuras, etc. que serán desplegados en el editor.

Los elementos pueden ser definidos con base en la composición de figuras geométricas básicas (triángulo, rectángulo, elipse, línea), definiendo la posición, tamaño, color, etc. de cada figura que será utilizada para representar a los elementos del modelo del dominio. La figura 51 muestra el modelo gráfico final para el editor de los servicios declarativos.

En la primera parte del modelo se aprecia la definición de un conjunto de elementos gráficos, cada uno representando a un elemento del perfil, existen tres lineas de conexión (*Polyline conection*), una para cada relación (*Provide, Reference y Declares*) además de una figura para cada nodo (*ServiceFigure, ComponentFigure y PropertyFigure*) y un par de etiquetas que son utilizadas para colocar el nombre de los elementos debajo de ellos (*ServiceNameFigure y ComponentNameFigure*).

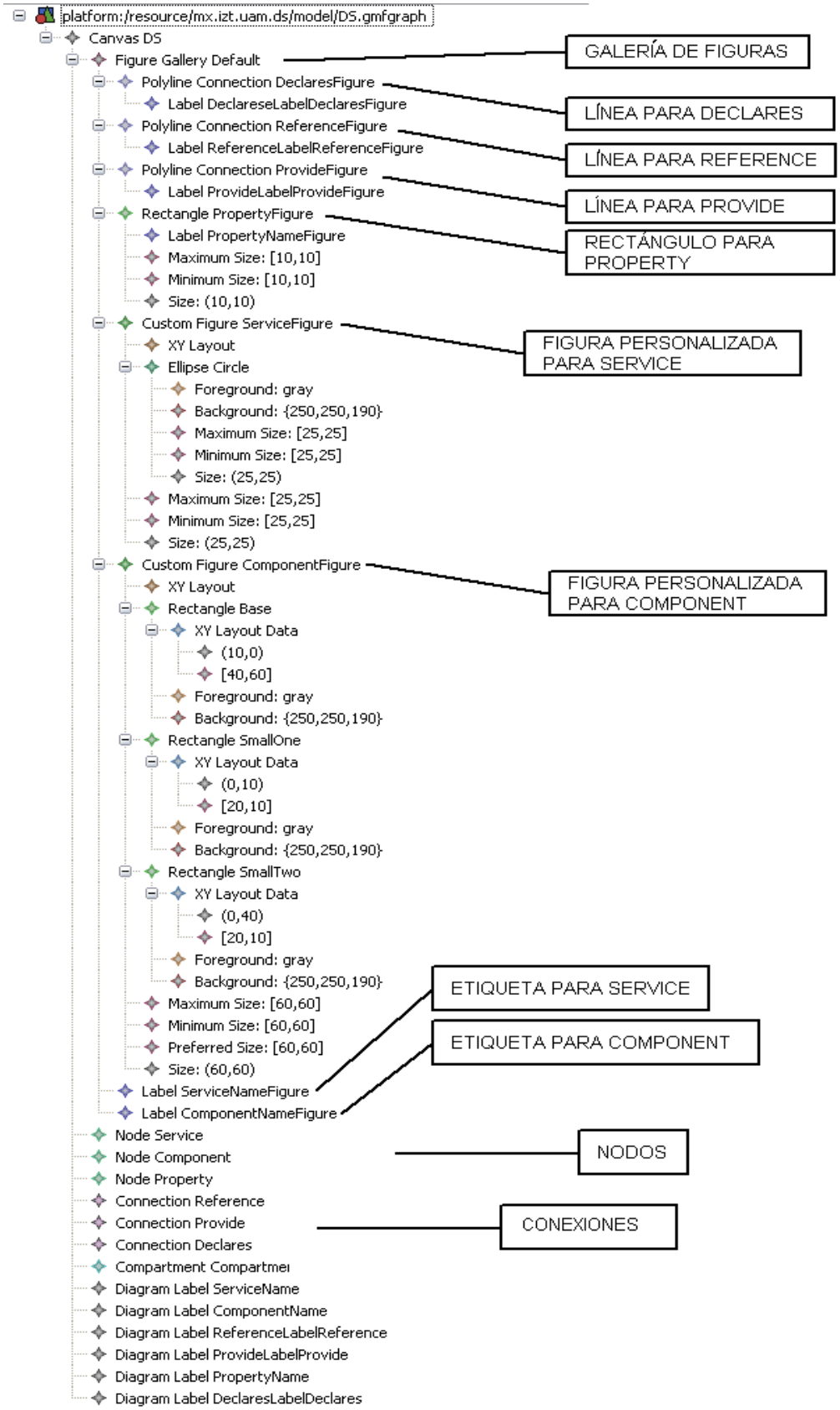

Figura 51: Modelo GMF de definición gráfica para los servicios declarativos

La figura 52 presenta los símbolos seleccionados para cada elemento. La figura para *Property* es un rectángulo de 10x10 pixeles. La figura para *Service* es una círculo con radio de 25 pixeles. La figura para un *Component* es una figura compuesta de 3 rectángulos, que unidos forman el símbolo estándar de UML para un componente.

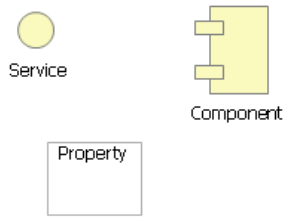

Figura 52: Elementos gráficos que representan elementos de nuestro perfil definidos en el modelo.

Además de las figuras, se definió cuáles de los elementos serán nodos (*Node*) y cuáles vértices o conexiones (*Connection*), asignándoles una figura y una etiqueta. Por ultimo, para cada etiqueta de texto que se quiera presentar en el editor como parte de los modelos debe existir un elemento de tipo *Diagram Label.* En este caso existe uno para el nombre de cada elemento y las etiquetas de las relaciones.

8.3. Modelo de definición de la paleta de herramientas

El tercer modelo (con extensión gmftool) representa los elementos visuales de dibujo que tendrá el editor, tales como menús, botones, etc., éste es, el modelo de definición de la paleta de herramientas.

Éste es el modelo más simple de GMF. Para el editor de los servicios declarativos se llegó al modelo que se muestra en la figura 53. En este modelo existe un grupo de herramientas (*tool group*) llamado DS con 6 herramientas de creación (*Creation tool).* Estas representan botones en la paleta con los cuales se podrán dibujar los elementos del modelo del dominio (*Component, Service, Property, Reference, Provide y Declares*).

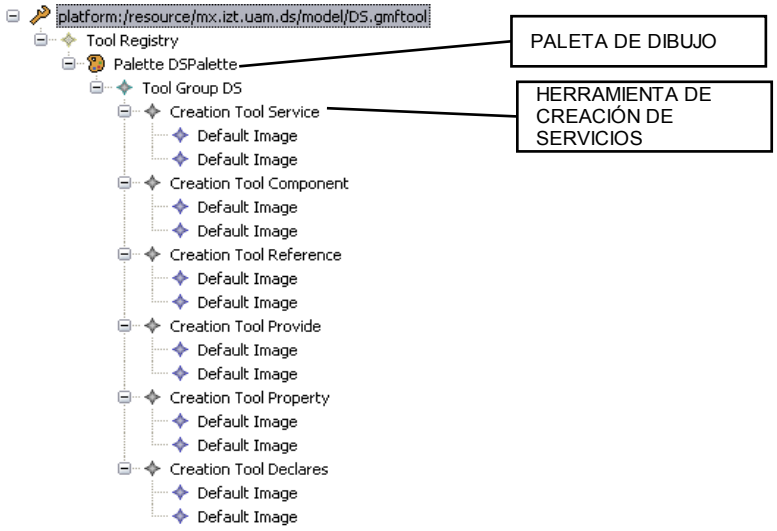

Figura 53: Modelo de definición de la paleta de herramientas para los servicios declarativos

8.4. Modelo de mapeo

En este punto del proceso se tienen tres modelos: del dominio, de definición gráfica y de definición de la paleta de herramientas. Para unir todos los modelos y definir qué elementos corresponden a qué elementos entre los distintos modelos es necesario crear un modelo que una todos los conceptos representados en los tres anteriores. Este es el modelo de mapeo. En el modelo de mapeo utiliza los elementos del modelo del dominio, asignándoles una figura del modelo gráfico y el elemento del modelo de la definición gráfica de la paleta con la que se dibujarán. La figura 54 muestra un fragmento de este modelo, en ésta se aprecia la definición de un *Top Node Reference,* elemento que representa un nodo en el editor visual. En la figura se muestran las propiedades de un nodo y se aprecia el mapeo entre los distintos modelos. La figura sólo muestra el ejemplo para el nodo *Service* y se aprecian las siguientes propiedades:

a) El elemento *Top Node Reference* debe indicar quién será el contenedor de los elementos de este tipo. En el modelo de dominio se definió un elemento llamado *Bundle* que contendrá a todos los elementos del modelo. En este caso existe una relación de composición entre *Bundle* y *Service* que indica que el *Bundle* contendrá a todos los nodos de tipo *Service* que se dibujen, ésta relacion se llama *containsService*.

b) En el elemento *Node Mapping* se realiza el mapeo entre los modelos. La propiedad *Element* define a qué elemento del modelo del dominio representa el nodo. La propiedad *Diagram Node* indica qué figura del modelo de definición gráfica será utilizada para representarlo. Por último la propiedad *Tool* indica con qué elemento del modelo de definición de la paleta de herramientas se dibujará.

c) Por último el elemento *Label Mapping* representa una etiqueta, que corresponde al elemento *Diagram Label* del modelo de definición gráfica y presentará en el nodo la propiedad del nodo indicada en *Features* (en este caso la etiqueta mostrará el valor que la propiedad *name* del elemento *Service*).

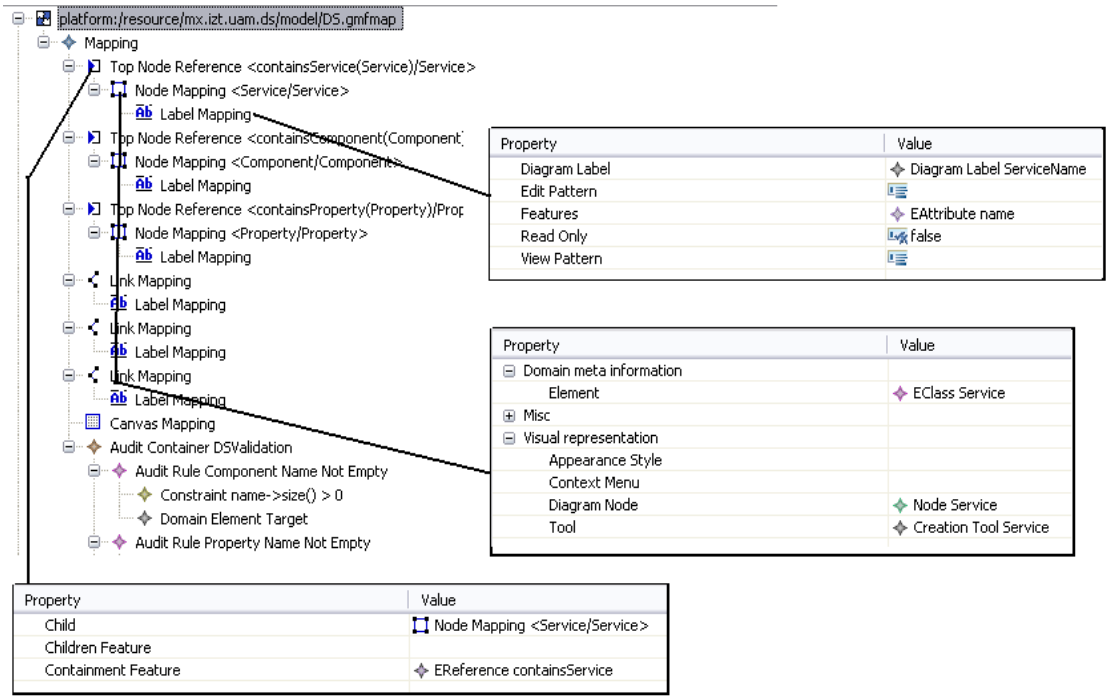

Figura 54: Fragmento del modelo de mapeo para mostrar el mapeo de nodos

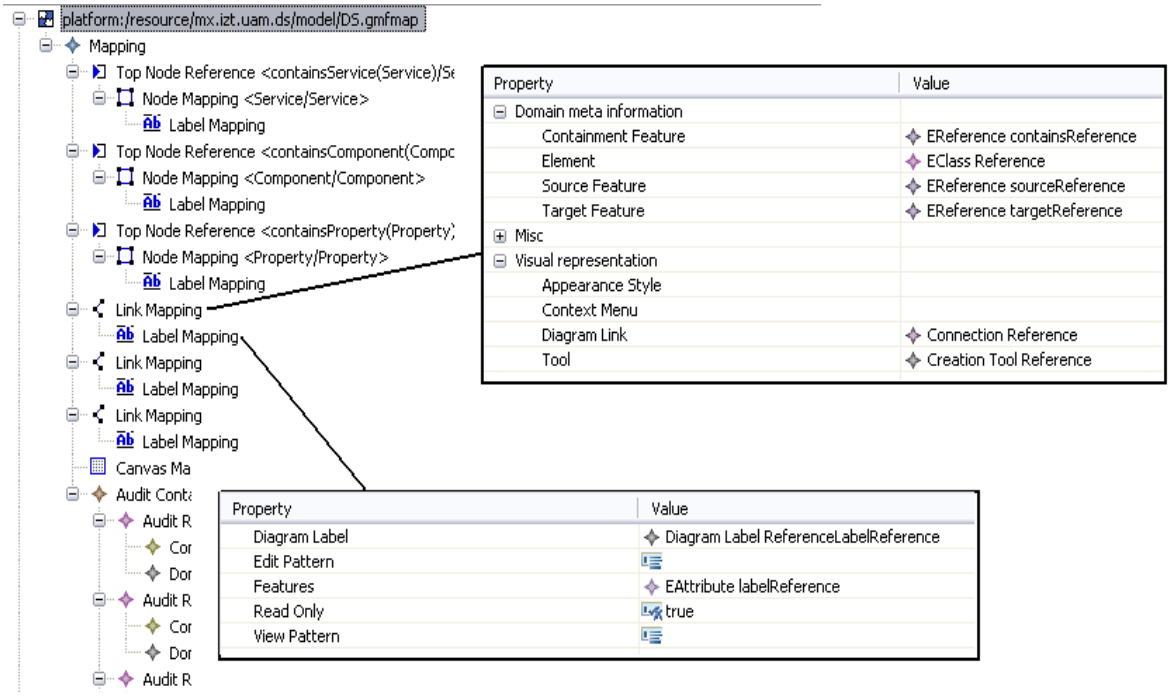

Figura 55: Fragmento del modelo de mapeo para mostrar el mapeo de asociaciones

La figura 55 muestra lo que sucede con las relaciones, en este caso con la relación *Reference*. Se aprecia en la figura lo siguiente:

a) El elemento *Link Mapping* indica el elemento del modelo del dominio (en este caso *Reference*).

La propiedad *Containment feature* representa lo mismo que en un nodo. Las propiedades *Source feature* y *Target feature* apuntan a elementos definidos en el modelo de dominio. Estas indican de nuevo qué elementos serán origen y destino en una asociación, en este ejemplo, si se revisa la figura 55, se podrá ver que, para el caso de *Reference,* el elemento origen es *Component* y el elemento destino es *Service,* lo que indica que esta relación sólo puede establecerse entre estos dos elementos y con el sentido indicado, de *Component* a *Service* (un componente sólo puede ofrecer un servicio).

b) Por último, se aprecia que, como en los nodos, se indica cual figura del modelo de definición gráfica se utilizará para este elemento y que herramienta de la paleta se utilizará para dibujarlo.

La figura 56 muestra el modelo de mapeo final con las restricciones en OCL que fueron definidas en el perfil UML.

A cada restricción en OCL se le asigna un nivel, de manera que una restricción puede categorizarse como un error, una advertencia o un simple mensaje informativo. Un mensaje personalizado del problema puede capturarse para ser desplegado al usuario cuando una restricción se viola.

8.5. Modelo generador de GMF

Una vez que se han definido los cuatro modelos básicos, mediante GMF se genera el llamado modelo generador (con extensión gmfgen). Este modelo es generado por GMF a partir del modelo de mapeo y sólo se modifica para personalizar las opciones de generación del editor visual. Este modelo contiene, toda la información de los cuatro modelos anteriores, y, la información necesaria para generar el código del editor visual.

La figura 57 muestra el modelo generador. En este caso fueron modificadas sólo algunas propiedades del modelo para personalizar el comportamiento del editor que será generado:

a) Los modelos que serán creados con el editor gráfico final serán guardados en archivos con formato XML. En este el modelo generador se puede definir la extensión que tendrán estos archivos de manera que el editor generara dos archivos por modelo, uno con extensión dsDiagram que almacena los elementos visuales del modelo y otro con extensión dsModel en formato XMI que contiene los elementos del dominio.

b) La propiedad *Package Name Prefix* que especifica el paquete de Java donde GMF generará todas las clases del editor visual fue modificada.

c) La propiedad *Validation Enabled* que habilita la validación del modelo de acuerdo a las restricciones establecidas en OCL fue cambiada a *true*. La propiedad *Validation Decotators* que habilita al editor para mostrar de manera visual los errores fue cambiada a *true*. A partir de este modelo, GMF genera el código GEF del editor visual. Sin embargo, para compilar el código del editor visual generado es necesario también generar un paquete de código más: el código de los elementos del metamodelo. El código de los elementos del metamodelo es un paquete de Java que contiene la estructura básica de los elementos que serán dibujados en el editor visual. Para generar éste paquete es necesario un ultimo modelo, llamado modelo generador de EMF. Este modelo no es parte de GMF sino de EMF, y lo que genera son clases en Java que representan los elementos de nuestro modelo de dominio. Este modelo es generado directamente y de manera automática por EMF a partir del modelo del dominio (*.ecore*).

Figura 56: Modelo GMF de mapeo

8.6. Modelo generador de EMF

En la figura 58 se aprecia que la estructura de este modelo (con extensión genmodel) es idéntica a la del modelo *ecore* del dominio, sólo que, en este modelo se especifican las propiedades necesarias para poder realizar la generación del código del:

a) Modelo que representa los elementos del metamodelo y sus atributos.

b) Edición del modelo, que se utiliza para modificar los valores de los atributos de los elementos del

modelo.

c) Prueba del código generado, que son clases que pueden ser utilizadas como base para probar el código generado utilizando JUNIT (*framework* de automatización de pruebas).

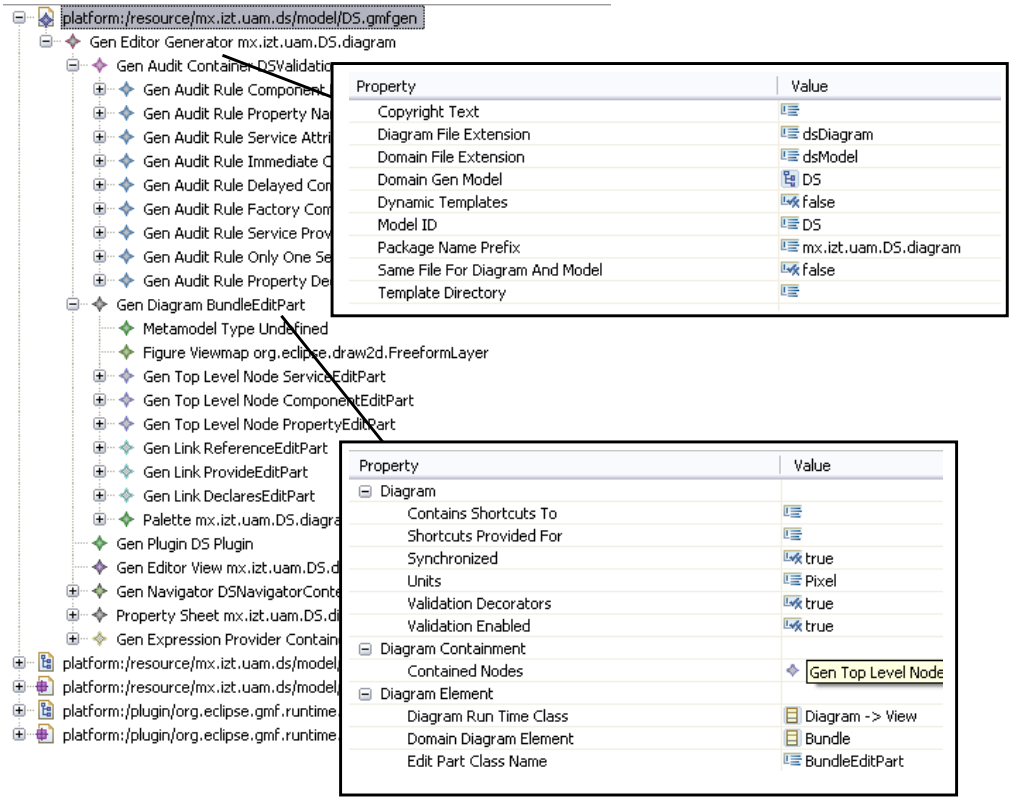

Figura 57: Modelo generador de GMF

GMF requiere para compilar el código del editor visual el código de edición del modelo y el código del metamodelo.

El modelo generador de GMF es creado a partir del modelo del dominio utilizando el *framework* de EMF. Una vez generado se modificaron algunas propiedades, de tal forma que el código que se genere pueda ser utilizado por GMF, estas propiedades son (figura 58):

a) La propiedad *Base Package* se modifica para personalizar el paquete donde será generado todo el código.

b) La propiedad *Prefix* se modifica para definir un prefijo que será utilizado para nombrar todos los elementos dentro del código generado, en este caso DS (*Declarative Services*).

Estos son todos los modelos que se requieren para crear el editor visual en GMF. Una vez generado el código solo queda ejecutar la aplicación como un *plugin* de Eclipse y verificar su funcionamiento.

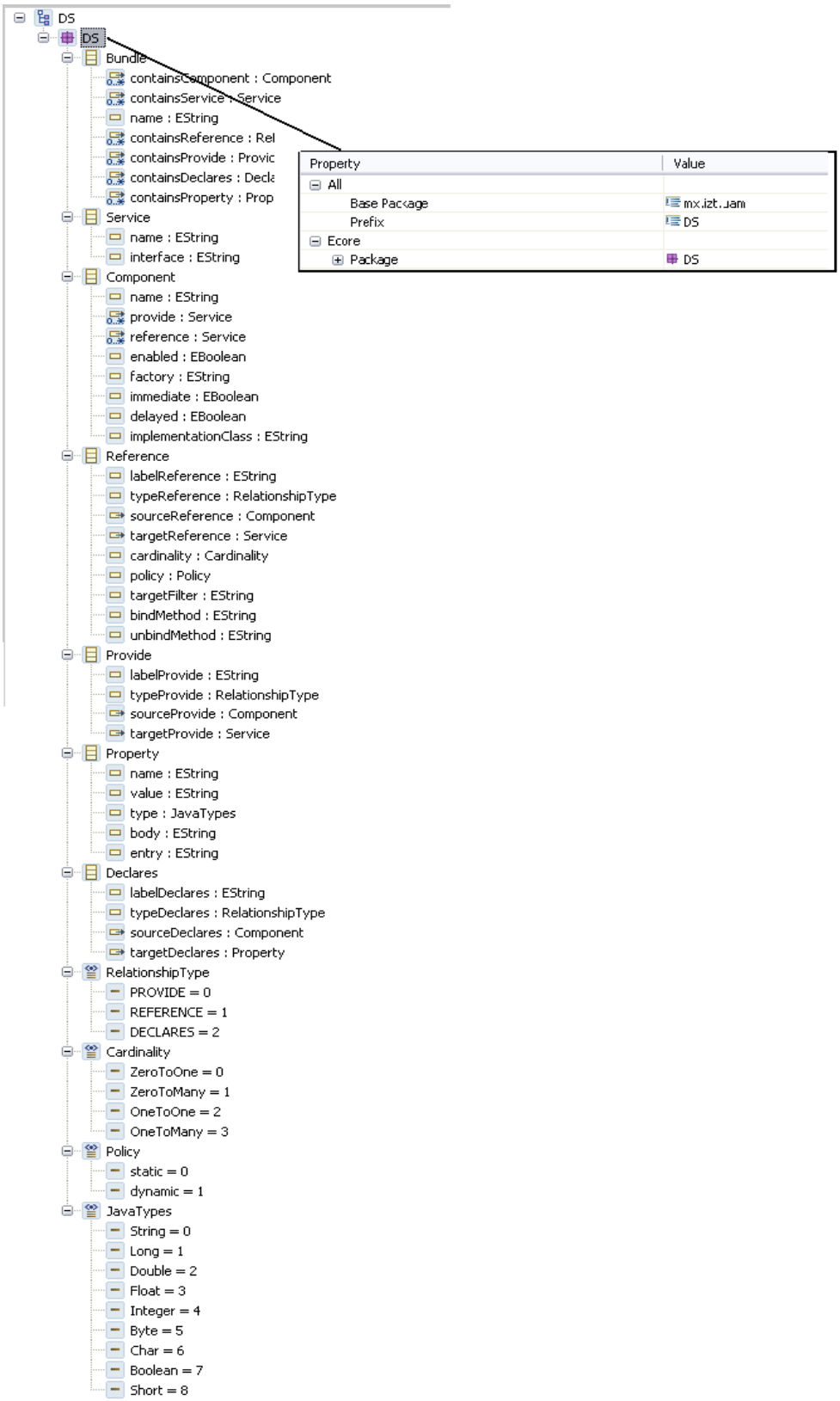

Figura 58: Modelo generador de EMF

9. Apéndice II: Funcionamiento de Acceleo

Los generadores de código son aplicaciones que permiten la generación de código a partir de modelos. Permiten la industrialización de la fase de construcción en un proceso de desarrollo de software. Acceleo es una herramienta basada en dos conceptos: modelos y plantillas. Mediante modelos uno define los parámetros de generación del código y mediante plantillas se expresa el código que será generado [31].

Acceleo está basado en EMF y por lo tanto es compatible con cualquier herramienta basada en este *framework.* Acceleo implementa compatibilidad con el estándar XMI. De esta forma cualquier modelo creado con EMF puede ser entrada para Acceleo. La figura 59 muestra la arquitectura de Acceleo.

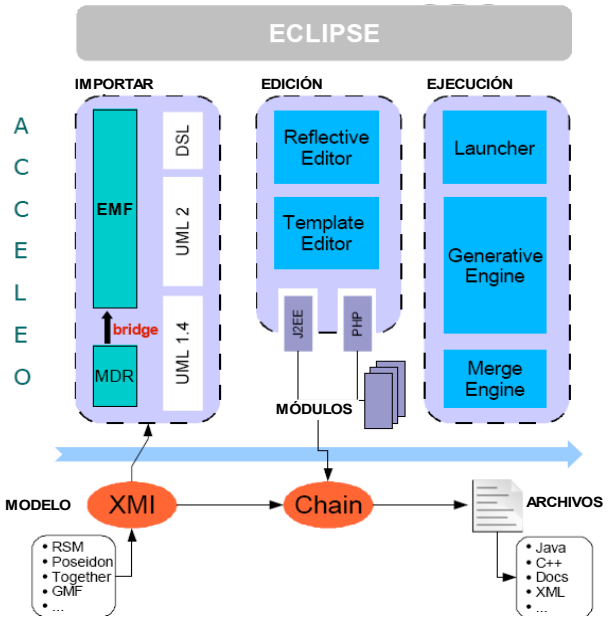

Figura 59: Arquitectura y proceso de desarrollo con Acceleo

El primer paso en el proceso de desarrollo con Acceleo es importar un modelo creado en algún editor visual (como el nuestro), y transferirlo a un formato XMI para ser manipulado por Acceleo. Acceleo requiere conocer el metamodelo utilizado para la creación del modelo. De manera interna Acceleo cuenta con el metamodelo de UML 1.4, UML 2.0 y DSL, pero puede utilizar cualquier otro metamodelo definido de manera externa para explorar el modelo.

Acceleo transforma el modelo en XMI a una representación en árbol como la utilizada por EMF (o los modelos de GMF). Esta tarea la realiza un editor visual que viene con el *framework* de Acceleo llamado *reflective editor* (figura 60). El *reflective editor* permite visualizar cualquier modelo basado en EMF asignándoles plantillas a cada uno de los elementos del modelo y, de ésta forma, previsualizar el código que será generado, de manera que las plantillas puedan probarse antes de utilizarlas.

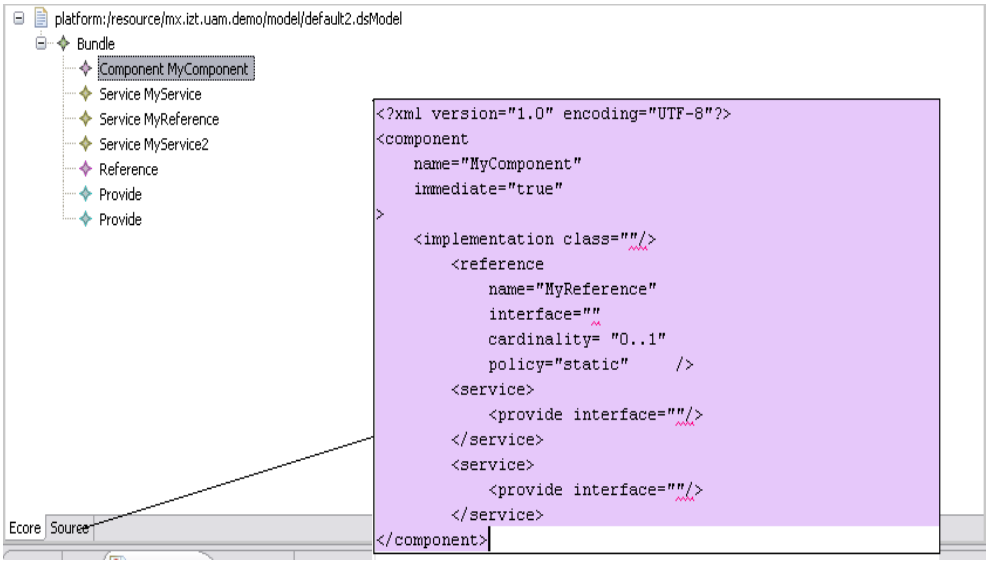

Figura 60: Modelo XMI y previsualización de código de Acceleo visto en el *reflective editor*

Además, Acceleo tiene un editor de plantillas (figura 61) que maneja aspectos como coloreado de sintaxis del lenguaje, detección de errores y autocompletado de código para hacer mas rápido y fácil el desarrollo de plantillas. La característica de autocompletado resulta bastante útil al desarrollar una plantilla ya que Acceleo proveerá de opciones de autocompletado para:

a) Elementos de otras plantillas definidos dentro del proyecto.

b) Atributos de los elementos definidos en el metamodelo.

c) Atributos EMF.

d) Servicios importados de Java.

e) Servicios de Java ofrecidos por Acceleo.

```
<%
metamodel http://www.uam.izt.mx/DS

%>
<%script type="DS.Component" name="activator" file="activator.xml"%>
<?xml version="1.0" encoding="UTF-8"?>
<component
    name="<%name%>"
    <%if (factory != null) {%>
    factory="<%factory%>"
    <%}%>
    <%if (immediate != false) {%>
    immediate="true"
    <%}%>
    <%if (enabled != false) {%>
    enabled="true"
    <%}%>
>
```

Figura 61: Editor de plantillas de Acceleo

Un aspecto importante de Acceleo es que no se limita a los servicios que nos ofrece el *framework* , si se desea extender la funcionalidad de las plantillas, se pueden importar clases de Java para ser utilizadas como servicios dentro de los mismos. En el apéndice III se detalla el funcionamiento del editor de plantillas de Acceleo y se presentan las plantillas creados para este proyecto.

Una vez creadas las plantillas, se debe crear un modelo EMF que contiene toda la información para poder generar el código a partir de las plantillas y el diagrama, este modelo se le llama modelo de cadena de transformación, la figura 62 muestra un ejemplo de este modelo. Se le llama cadena de transformación porque se pueden generar varios archivos al mismo tiempo a partir del mismo modelo. El modelo de cadena de transformación es ejecutado por el *Launcher* (un motor que lo interpreta) y es utilizado por el *Generative Engine* y el *Merge Engine* (figura 59) para generar los archivos especificados en cada plantilla.

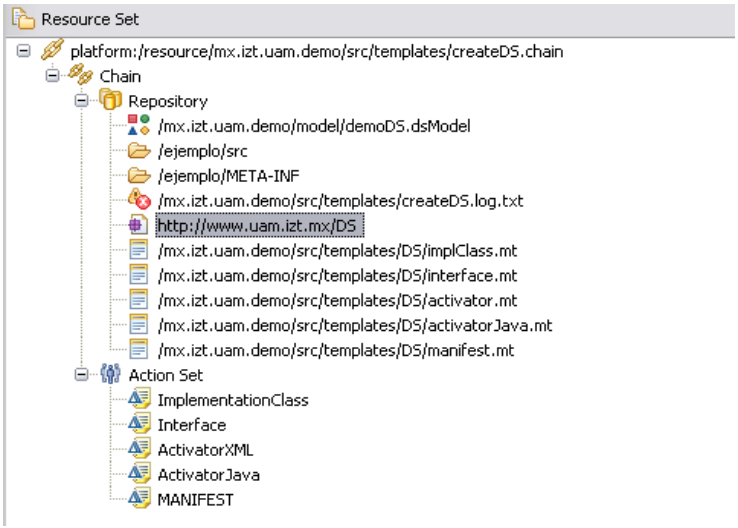

Figura 62: Ejemplo de modelo de cadena de transformación

10. Apéndice III: Plantillas de Acceleo

10.1. Implementación de las reglas de transformación en plantillas de Acceleo

A continuación se presentan las cuatro plantillas creadas para los servicios declarativos.

Las plantillas en Acceleo tienen extensión “mt”. Todas los plantillas comienzan con la linea

```
<% metamodel http://www.uam.izt.mx/DS%>
```

Esta línea indica cual es el metamodelo que será utilizado para navegar por los elementos de los modelos creados con el editor visual, es éste caso éste es el mismo que fue definido en el modelo del dominio de GMF (véase apéndice I). Inmediatamente después aparece la línea

```
<%script type="DS.Component" name="activator" file="activator.xml"%>
```

El atributo *type* de *script* se refiere a que elemento del metamodelo (en este caso *Component*) será utilizado como base para navegar por el modelo. El atributo *name* indica el nombre del *script* y el atributo *file* indica el nombre del archivo que será generado con la ejecución de este *script*.

A continuación se presentan todas los plantillas creadas para el proyecto. Algunos comentarios en negritas han sido agregados para hacer más comprensibles los *scripts*.

Descriptor.mt

```
<% metamodel http://www.uam.izt.mx/DS                    -------> METAMODELO UTILIZADO
%>
<%script type="DS.Component" name="activator" file="activator.xml"%> ----> ELEMENTO DEL METAMODELO
                                                                          UTILIZADO COMO BASE Y NOMBRE
                                                                          DEL ARCHIVO QUE SE GENERARÄ
<?xml version="1.0" encoding="UTF-8"?>
<component                                               -------> COMPONENT
        name="<%name%>"
        <%if (factory != null) {%>                       -------> FACTORY <> null
    factory="<%factory%>"
    <%}%>
        <%if (immediate != false) {%>                    -------> IMMEDIATE = TRUE
    immediate="true"
    <%}%>
        <%if (enabled != false) {%>                      -------> ENABLED = TRUE
    enabled="true"
    <%}%>
>
        <implementation class="<%implementationClass%>"/>
    <%for (self().followingSibling()) {%>                        -------> REVISA CADA ASOCIACIÓN DE COMPONENT
            <%if (self().eClass().name == "Property") {%>
                <%if (self().entry == null) {%>
                        <%if (self().value != null) {%>
            <property                                            -------> PROPERTY
                    name="<%self().name%>"
                    value="<%self().value%>"
                    type="<%self().type%>"
            />
                        <%}else{%>
            <property
                    name="<%self().name%>">
                    <%self().body%>
            </property>
                        <%}%>
                <%}else{%>
            <properties entry="<%self().entry%>"/>
                <%}%>
            <%}%>
            <%if (self().eClass().name == "Reference") {%>
```

```
                <reference                                                      -----> REFERENCE
                        name="<%self().targetReference.name%>"
                        interface="<%self().targetReference.interface%>"
                        <%if (self().bindMethod != null) {%>
                        bind= "<%self().bindMethod%>"
                        <%}%>
                        <%if (self().unbindMethod != null) {%>
                        unbind= "<%self().unbindMethod%>"
                        <%}%>
                        <%if (self().cardinality == "OneToOne") {%>
                        cardinality= "1..1"
                        <%}%>
                        <%if (self().cardinality == "ZeroToOne") {%>
                        cardinality= "0..1"
                        <%}%>
                        <%if (self().cardinality == "ZeroToMany") {%>
                        cardinality= "0..n"
                        <%}%>
                        <%if (self().cardinality == "OneToMany") {%>
                        cardinality= "1..n"
                        <%}%>
                        <%if (self().targetFilter != null) {%>
                        target= "<%self().targetFilter%>"
                        <%}%>
                        policy="<%self().policy%>"           />
                <%}%>
                <%if (self().eClass().name == "Provide") {%>
                <service>                                                                   ----->SERVICE
                        <provide interface="<%self().targetProvide.interface%>"/>
                </service>
                <%}%>
        <%}%>
</component>
```

Interfaz del servicio

```
<%
metamodel http://www.uam.izt.mx/DS          -------> METAMODELO UTILIZADO

import mx.uam.ejemplo.services.StringServ   ----> CLASE DE JAVA IMPORTADA PARA UTILIZAR LOS MÉTODOS
                                                  returnClass y returnPackage
%>
<%script type="Component" name="interface" file="<%provide.interface.returnClass()%>.java"%>
package <% provide.interface.returnPackage()%>;

import org.osgi.framework.BundleContext;

public interface <%provide.interface.returnClass()%> {

        abstract void activate(BundleContext ctxt);
//      <%startUserCode%>                   ----> MARCAS QUE INDICAN UN BLOQUE DE CÓDIGO DEL USUARIO
//  Here you must write the methods that are provided by this interface
//      <%endUserCode%>
}
```

En la interfaz del servicio se utiliza la sentencia *import* para importar librerías externas escritas en Java. La librería llamada *StringServ* (listada a continuación) fue el único código que se escribió en todo el proceso de desarrollo del proyecto y contiene dos métodos muy simples de manipulación de cadenas. El método *returnClass* obtiene el nombre de la clase en una cadena del tipo "mx.izt.uam.MiClase", es decir, "MiClase" y *returnPackage* regresa el paquete donde se encuentra esta clase, es decir, "mx.izt.uam". Estos métodos requieren como parámetro una cadena con el valor de alguna propiedad del modelo, en el caso de esta plantilla, la propiedad *interface* de *Service*.

Los métodos de esta clase requieren implementar la excepción *EnodeCastException*, que es una excepción de Acceleo que se presenta cuando no puede transformarse al tipo *String* la propiedad que se le pasa al método.

StringServ.java

```
package mx.uam.ejemplo.services;

import fr.obeo.acceleo.gen.template.eval.ENodeCastException;

public class StringServ {

		/**
		 * Return the String with the name of the Class in a path.
		 */
		public String returnClass(String s) throws ENodeCastException {
			return s.substring(s.lastIndexOf('.')+1);
		}

		/**
		 * Return the String with the name of the package in a path.
		 */
		public String returnPackage(String s) throws ENodeCastException {
			return s.substring(0, s.lastIndexOf('.'));
		}
}
```

Clase Java de implementación del servicio

```
<% metamodel http://www.uam.izt.mx/DS        -------> metaMODELO UTILIZADO

   import mx.uam.ejemplo.services.StringServ ----> CLASE DE JAVA IMPORTADA PARA UTILIZAR LAS FUNCIONES
                                                   returnClass y returnPackage
%>
<%script type="Component" name="implClass" file="<%implementationClass.returnClass()%>.java"%>
package <%implementationClass.returnPackage()%>;

import org.osgi.framework.BundleContext;

public class <%implementationClass.returnClass()%> implements <%provide.interface.returnClass()%> {

	public void activate(BundleContext ctxt) {
//	<%startUserCode%>
//	Here you must write the implementation of the methods
//	<%endUserCode%>
	}
}
```

Manifest.mf

```
<%
metamodel http://www.uam.izt.mx/DS

import mx.uam.ejemplo.services.StringServ
%>

<%script type="DS.Component" name="manifest" file="MANIFEST.MF"%>
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: <%name%>
Bundle-SymbolicName: <%name%>
Bundle-Version: 1.0.0
Service-Component: metaINF/activator.xml
Bundle-Activator: <% provide.interface.returnPackage()%>.Activator
Bundle-Localization: plugin
Import-Package: org.osgi.framework;version="1.3.0"
```

Estas plantillas fueron probadas utilizando el *reflective editor* de Acceleo. Después de tener las plantillas completas y probadas el último paso en la automatización de la generación de código fue la creación del un modelo llamado modelo de cadena de transformación.

10.2. Modelo de cadena de transformación para los servicios declarativos

El modelo de cadena de transformación (*chain),* es un modelo EMF de Acceleo donde se definen los parámetros necesarios para la generación de código a partir del modelo. El modelo de cadena se presenta en la figura 63.

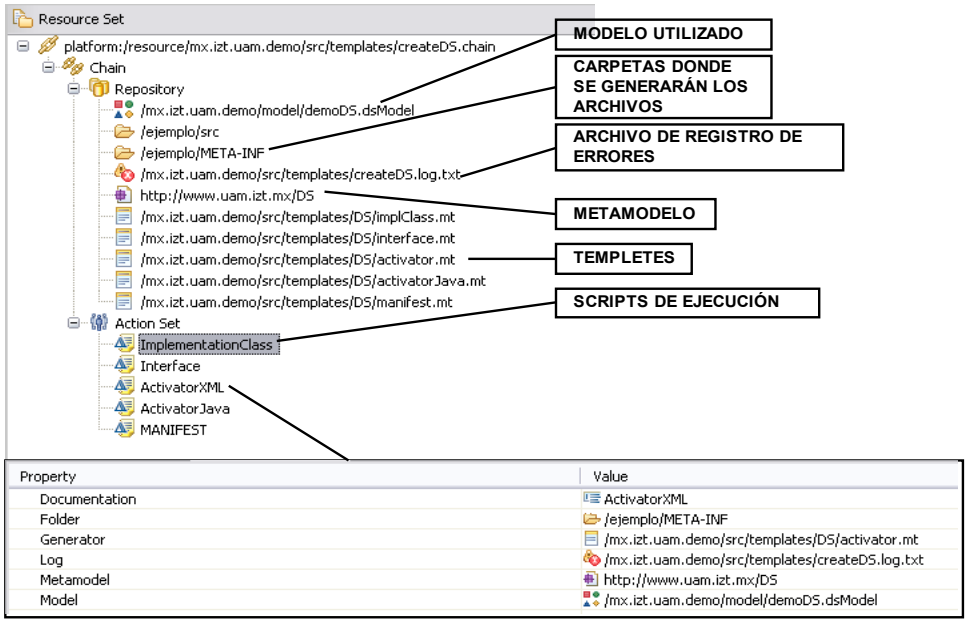

Figura 63: Modelo de cadena de transformación para los servicios declarativos

El modelo define:

a) Qué modelo serán utilizados como entrada para la generación de código.

b) Las distintas carpetas en las que serán generados los archivos. En este caso *src* para el código en Java y META-INF para el descriptor XML y el *manifest*.

c) Ruta de un archivo o archivos de registro de errores donde serán registradas las excepciones.

d) El metamodelo utilizado para navegar por los modelos. En este caso http://www.uam.izt.mx/DS

e) Las distintas plantillas que se utilizarán para la generación de código.

f) *Scripts* de ejecución. Acceleo permite crear *scripts* para generar código, generar respaldos y borrar archivos. Estos se ejecutan en el orden en que son colocados en el modelo. Los *scripts* de generación de código utilizan los parámetros antes definidos para generar un archivo. En la figura 63 se muestran las propiedades del *script* ActivatorXML, utilizado para crear el descriptor XML:

- *Documentation*. Es el nombre del *script*.
- *Folder*. De las carpetas definidas anteriormente se selecciona una donde será generado el archivo, en este caso, META-INF.
- *Generator*. Es la plantilla que se utilizará.
- *Log*. Es el archivo que se utilizará para llevar el registro de problemas.
- *Metamodel*. Es el metamodelo que se utilizará para navegar por el modelo que se

transformará.

– *Model*. Es el diagrama creado con el editor visual, el archivo con extensión dsModel en este caso.

Una vez creado el modelo de cadena de transformación, este puede ser ejecutado para generar el código del *bundle*.

11. Apéndice IV: Aplicación de nuestro proceso

Dentro del mismo proyecto existían otros problemas que requerían como solución, la implementación de una herramienta para acelerar el desarrollo de bundles. Utilizamos nuestro proceso propuesto para desarrollar de manera rápida herramientas MDA de una forma simple y rápida. El proceso que se siguió para crear la herramienta para los servicios declarativos. Ambos problemas tienen su base en otra forma de interacción entre servicios que se encuentra también dentro de la especificación de OSGi [37], llamado el Servicio *WireAdmin*.

La primera herramienta MDA fue construida para un enfoque de tipo declarativo. El proceso se completó en 1 día. La segunda herramienta MDA tenía que generar la estructura básica del código en Java que implementara los requerimientos no funcionales de la aplicación. Este segundo proceso se completó en 3 días.

11.1. Servicio *WireAdmin*

11.1.1. Problemática

El servicio *WireAdmin* es un servicio administrativo que es utilizado para controlar una topología de conexión dentro de la plataforma OSGi siguiendo un enfoque productor-consumidor. *WireAdmin* se encarga de conectar entre ellos tanto a los productores como a los consumidores de datos siguiendo un enfoque orientado a servicios. Los *bundles* que participan en este proceso se deben registrar para funcionar como servicios consumidores o servicios productores. El servicio *WireAdmin* conecta los servicios que producen datos con los servicios que consumen esos datos y su propósito es permitir la cooperación de *bundles* [37]. Por ejemplo un sensor de temperatura puede ser conectado a un calefactor para proveer un sistema de control de temperatura.

La descripción completa del funcionamiento de este servicio está fuera del alcance de este documento, sin embargo para comprender éste apéndice, se requiere conocer las siguientes definiciones:

a) **Productor**. Es un componente de servicio que genera información a ser utilizada por un servicio consumidor.

b) **Consumidor**. Es un componente de servicio que recibe la información generada por un servicio productor.

c) **Alambre** (*Wire*). Es un objeto creado por el servicio *WireAdmin* que define una asociación entre un servicio productor y un servicio consumidor. Múltiples objetos de este tipo pueden existir entre el el mismo par de consumidores y productores.

d) **Sabores** (*Flavors*). Los distintos tipos de datos que pueden ser utilizados para intercambiar información entre un productor y un consumidor.

El desarrollo de aplicaciones basadas en este servicio puede resultar bastante complejo, muchos factores tienen que considerarse, sin embargo, muchas partes del código de estas aplicaciones tiende a ser repetitivo y debido a esto, una herramienta MDA que genere el esqueleto básico de una aplicación basada en un modelo simple de la misma es de gran utilidad.

Se siguió el proceso propuesto para el desarrollo de la herramienta para los servicios declarativos

mostrado en la figura 47 para generar la herramienta para el servicio *WireAdmin*. Las siguientes secciones de este capítulo presentan los resultados finales obtenidos para este problema.

11.1.2. Perfil UML para el servicio *WireAdmin*

La figura 64 muestra el metamodelo y la figura 65 muestra el perfil UML finales creados para modelar aplicaciones utilizando el servicio *WireAdmin*.

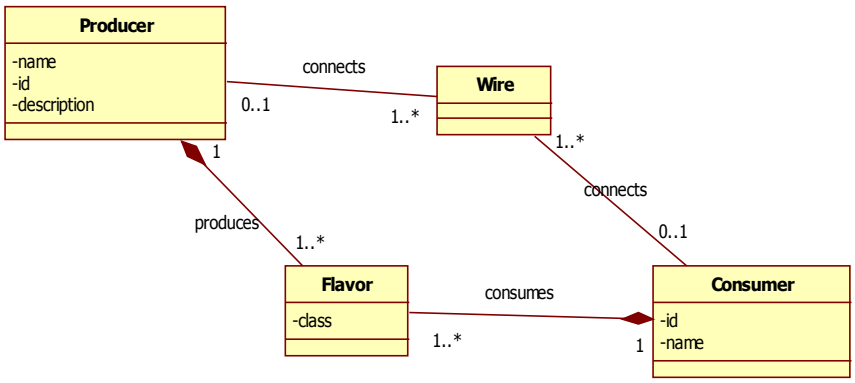

Figura 64: metamodelo del servicio WireAdmin

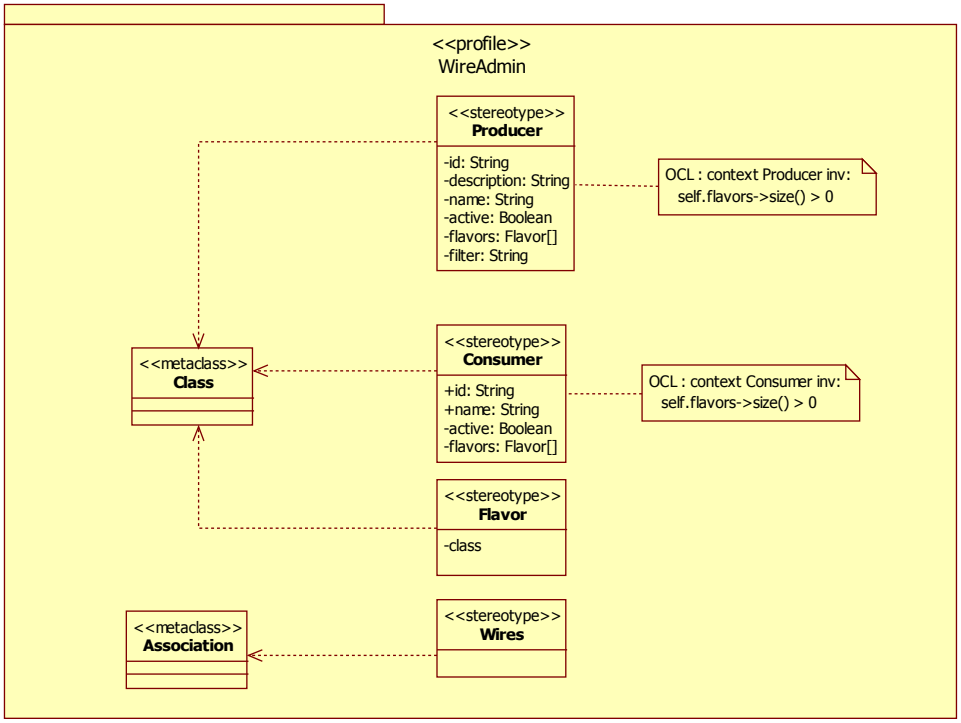

Figura 65: Diagrama del perfil UML para el servicio WireAdmin

11.1.3. Modelos GMF creados para el servicio *WireAdmin*

Se crearon todos los modelos que se especificaron en detalle en el capítulo 6 de este trabajo para el caso de los servicios declarativos, estos son:

1. El modelo EMF del dominio (figura 66) .
2. El modelo de definición gráfica (figura 67) .
3. El modelo de definición de la paleta de dibujo (figura 68).
4. El modelo de mapeo con las restricciones OCL (figura 69).

Los modelos generador de GMF y generador de EMF no se muestran debido que estos son generados a partir de los modelos de mapeo y de dominio respectivamente.

11.1.4. Editor gráfico generado para el servicio *WireAdmin*

Una vez con los modelos completos se generó el editor gráfico para modelar aplicaciones basadas en el servicio *WireAdmin*. En la figura 70 se muestra el editor generado, en esta imagen se puede apreciar un consumidor y un productor unidos a través de un alambre, cada uno es capaz de manejar distintos sabores.

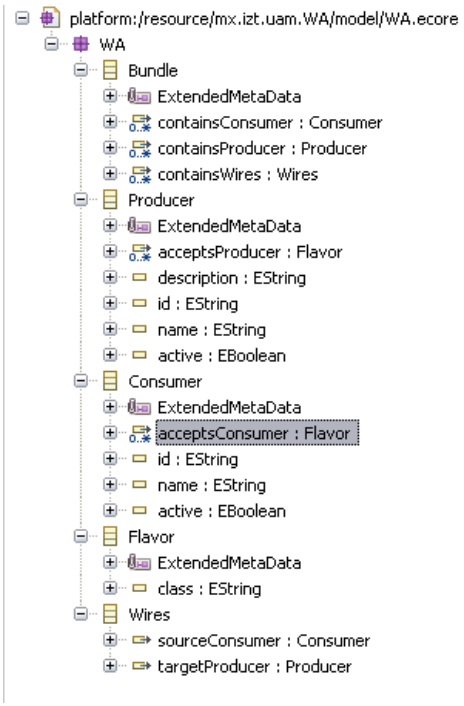

Figura 66: Modelo de dominio GMF para el servicio WireAdmin

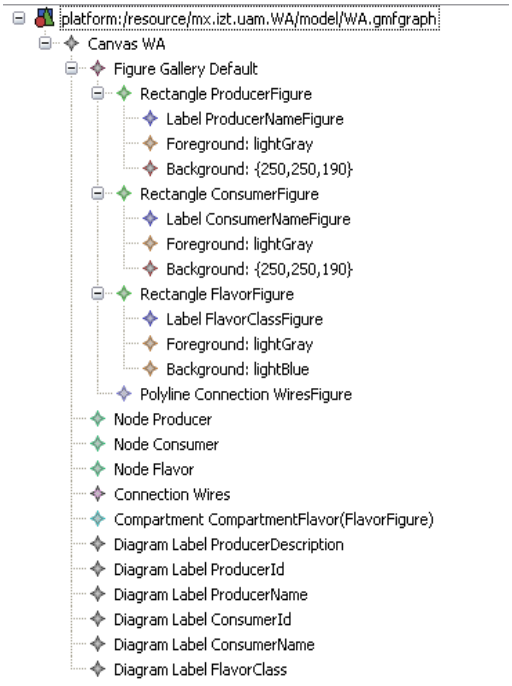

Figura 67: Modelo de definición gráfica GMF para el servicio WireAdmin

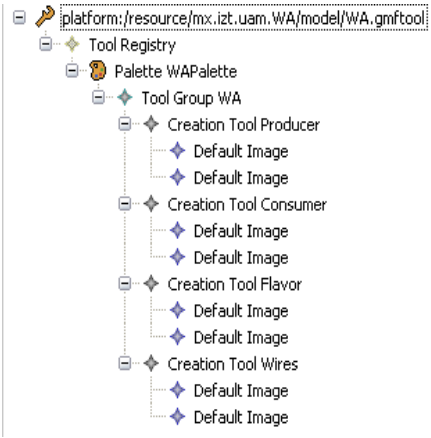

Figura 68: Modelo de definición de la paleta de herramientas de GMF para el servicio WireAdmin

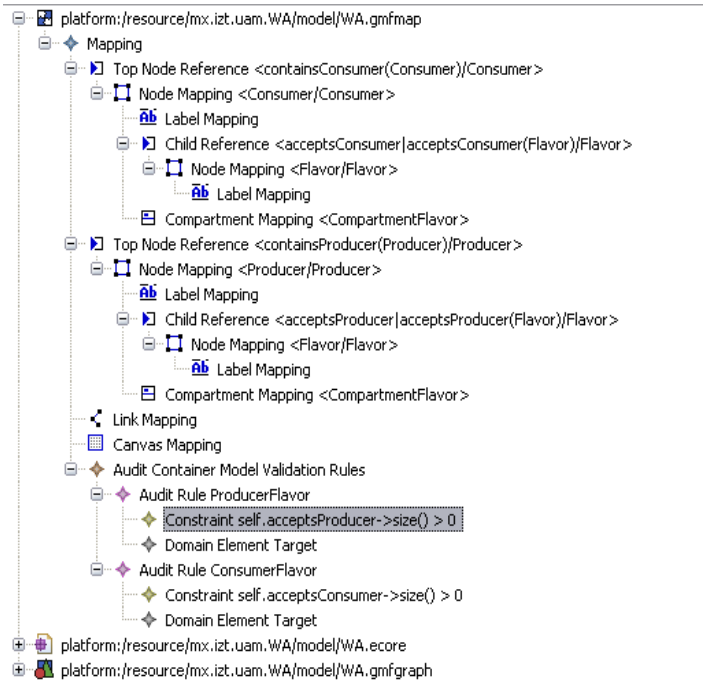

Figura 69: Modelo de mapeo GMF para el servicio WireAdmin

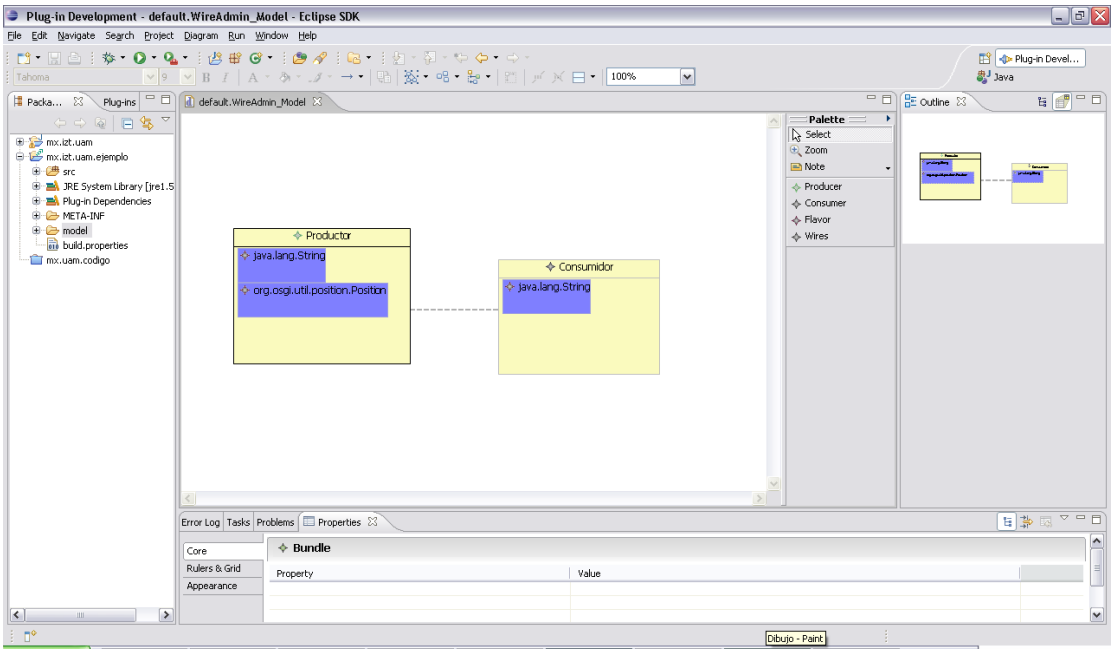

Figura 70: Editor visual para modelar aplicaciones basadas en el servicio WireAdmin

11.1.5. Reglas de transformación y plantillas de Acceleo

En este caso gran cantidad de código es generada a partir del modelo que en esencia es muy simple, esto muestra un ejemplo donde una herramienta MDA es de gran ayuda en la construcción de aplicaciones. A continuación se presenta el conjunto de reglas de transformación definidas para este problema.

Elemento del modelo	*Código generado*

Consumer	**Consumer.java**
	`import java.util.Dictionary;`
	`import java.util.Hashtable;`
	
	`import org.osgi.framework.BundleActivator;`
	`import org.osgi.framework.BundleContext;`
	`import org.osgi.framework.Constants;`
	`import org.osgi.framework.ServiceRegistration;`
	`import org.osgi.service.wireadmin.Consumer;`
	`import org.osgi.service.wireadmin.Wire;`
	`import org.osgi.service.wireadmin.WireConstants;`
	
	`public void stop(BundleContext bundleContext) throws Exception {`
	`serviceRegistration.unregister();`
	`}`
	`public void producersConnected(Wire[] wires) {`
	`this.wires = wires;`
	`}`

Producer	**Producer.java**
	`import java.util.Hashtable;`
	`import org.osgi.framework.BundleContext;`
	`import org.osgi.framework.ServiceRegistration;`
	`import org.osgi.service.wireadmin.Producer;`
	`import org.osgi.service.wireadmin.Wire;`
	
	`context.registerService(Producer.class.getName(), this, ht);`
	
	`public void end() {`
	`this.quit = true;`
	`}`
	
	`public synchronized void consumersConnected(Wire wires[]) {`
	`this.wires = wires;`
	`}`

Consumer.acceptsConsumer, Flavor.class	**Consumer.java**
	`import <Flavor.class>;`
	
	`public void updated(Wire wire, Object obj) {`
	`if ( obj instanceof Date )`
	`process<Flavor.class>( (<Flavor.class>) obj );`
	`}`
	
	`properties.put(WireConstants.WIREADMIN_CONSUMER_FLAVORS,`
	`new Class[] {`
	`<Flavor.class>.class,`
	`});`
	
	`serviceRegistration=bundleContext.registerService(Consumer.class.getName(), this, properties);`
	
	`public void process<Flavor.class> (<Flavor.class> input) {`
	
	`}`

Producer.acceptsProducer, Flavor.class	**Producer.java**
	`import <Flavor.class>;`
	
	`public synchronized void run() {`
	`Hashtable ht = new Hashtable();`
	
	`ht.put(org.osgi.service.wireadmin.WireConstants.WIREADMIN_PRODUCER_FLAVORS,new Class[] {`
	`<Flavor.class>.class,`
	`});`
	
	`public Object polled(Wire wire) {`

Elemento del modelo	*Código generado*

	Class clazzes[] = wire.getFlavors();
	for (int i = 0; i < clazzes.length; i++) {
	Class clazz = clazzes[i];
	if (clazz.isAssignableFrom(<Flavor.class>.class)){
	<Flavor.class> obj = new <Flavor.class>();
	return obj;
	}
	}
	return null;
	}

Consumer.name	**Consumer.java**
	public class **<**Consumer.name**>**
	implements Consumer, BundleActivator {
	
	private Wire[] wires = null;
	private ServiceRegistration serviceRegistration;
	
	public <Consumer.name> ( ) {
	
	}

Producer.name	**Producer.java**
	public class <Producer.name> implements Producer, Runnable {
	
	private Wire wires[];
	private BundleContext context;
	private ServiceRegistration servreg;
	private boolean quit;
	
	public <Producer.name>(BundleContext context) {
	this.context = context;
	
	servreg = null;
	quit = false;
	new Thread(this).start();
	}

Producer.active = true	**Producer.java**
	while (!quit)
	try {
	for (int i = 0; wires != null && i < wires.length; i++){
	Wire wire=wires[i];
	if(!wire.isConnected() || !wire.isValid())
	continue;
	wire.update(polled(wire));
	}
	wait(150000);
	} catch (InterruptedException ie) {
	}

Consumer.id	**Consumer.java** public void start(BundleContext bundleContext) throws Exception { Dictionary properties = new Hashtable(); properties.put(Constants.SERVICE_PID,"<Consumer.id>");
Producer.id	**Producer.java** ht.put(org.osgi.framework.Constants.SERVICE_PID, "<Producer.id>");
Producer.description	**Producer.java** ht.put(org.osgi.framework.Constants.SERVICE_DESCRIPTION, "<Producer.description>");

Tabla 9: Reglas de transformación para el servicio WireAdmin

Con base en estas reglas se crearon las plantillas de Acceleo, en este caso se generan dos archivos

en código Java, uno para el productor y otro para el consumidor. A continuación se presentan ambas plantillas.

Consumer.mt

```
<%
metamodel http://www.uam.izt.mx/WA
%>

<%script type="WA.Consumer" name="consumer" file="<%name%>.java"%>

package // <%startUserCode%>
                // Write the package path

                // <%endUserCode%>

import java.util.Dictionary;
import java.util.Hashtable;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.Constants;
import org.osgi.framework.ServiceRegistration;
import org.osgi.service.wireadmin.Consumer;
import org.osgi.service.wireadmin.Wire;
import org.osgi.service.wireadmin.WireConstants;

//<%startUserCode%>
// Here write all the imports that's you require for your code

<%for (acceptsConsumer) {%>
import <%class%>;
<%}%>

//<%endUserCode%>

public class <%name%>
        implements Consumer, BundleActivator {

        private Wire[] wires = null;
        private ServiceRegistration serviceRegistration;
        //<%startUserCode%>
        // Here write all the attributes you will required;
        //<%endUserCode%>


        public <%name%> ( ) {
                //<%startUserCode%>
                // Here write all the inicialization code;
                //<%endUserCode%>
        }


        public void start(BundleContext bundleContext) throws Exception {
                Dictionary properties = new Hashtable();
                properties.put(
                        Constants.SERVICE_PID,"<%id%>");
                properties.put(
                        WireConstants.WIREADMIN_CONSUMER_FLAVORS,
                        new Class[] {
                        <%for (acceptsConsumer) {%>
                                <%class%>.class,
                        <%}%>
                        });
                serviceRegistration=bundleContext.registerService(Consumer.class.getName(),
                        this, properties);
        }

        public void stop(BundleContext bundleContext) throws Exception {
                serviceRegistration.unregister();
```

```
        }

        public void updated(Wire wire, Object obj) {
                <%for (acceptsConsumer) {%>
                if ( obj instanceof Date )
                        process<%class%>( (<%class%>) obj );
                <%}%>
        }

        public void producersConnected(Wire[] wires) {
                this.wires = wires;
        }

        <%for (acceptsConsumer) {%>
        public void process<%class%> (<%class%> input) {
        // <%startUserCode%>
        // Write the code for the manupilation of the object of type <%class%>
        // <%endUserCode%>
        }
        <%}%>

        //<%startUserCode%>
        // Write your own code;
        //<%endUserCode%>

}
```

Producer.mt

```
<%
metamodel http://www.uam.izt.mx/WA
%>

<%script type="WA.Producer" name="producer" file="<%name%>.java"%>

package //<%startUserCode%>
                //Write the package path

                //<%endUserCode%>

import java.util.Hashtable;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import org.osgi.service.wireadmin.Producer;
import org.osgi.service.wireadmin.Wire;

//<%startUserCode%> for import
// Here write all the imports that's you require for your code ;

<%for (acceptsProducer) {%>
import <%class%>;
<%}%>

//<%endUserCode%> for import

public class <%name%> implements Producer, Runnable {

        private Wire wires[];
        private BundleContext context;
        private ServiceRegistration servreg;
        private boolean quit;

        public <%name%>(BundleContext context) {
                this.context = context;

                servreg = null;
                quit = false;
                new Thread(this).start();
        }

        public synchronized void run() {
```

```
                Hashtable ht = new Hashtable();
                ht.put(
                        org.osgi.service.wireadmin.WireConstants.WIREADMIN_PRODUCER_FLAVORS,
                        new Class[] {
                        <%for (acceptsProducer) {%>
                                <%class%>.class,
                        <%}%>
                        });
                ht.put(org.osgi.framework.Constants.SERVICE_PID,
                        "<%id%>");
                ht.put( org.osgi.framework.Constants.SERVICE_DESCRIPTION,
                        "<%description%>");

                context.registerService(Producer.class.getName(), this, ht);

                <%if (active == true) {%>
                while (!quit)
                        try {
                                for (int i = 0; wires != null && i < wires.length; i++){
                                        Wire wire=wires[i];
                                        if(!wire.isConnected() || !wire.isValid())
                                                continue;
                                        wire.update(polled(wire));
                                }
                                wait(150000);
                        } catch (InterruptedException ie) {
                        }
                <%}%>
        }

        public synchronized void consumersConnected(Wire wires[]) {
                this.wires = wires;
        }

        public Object polled(Wire wire) {
                Class clazzes[] = wire.getFlavors();
                for (int i = 0; i < clazzes.length; i++) {
                        Class clazz = clazzes[i];
                        <%for ( acceptsProducer){%>
                        if (clazz.isAssignableFrom(<%class%>.class)) {
                                <%class%> obj = new <%class%>();
                                //<%startUserCode%>
                                // Here write your code for this type;
                                //<%endUserCode%>
                                return obj;
                        }
                        <%}%>
                }
                return null;
        }

        public void end() {
                this.quit = true;
        }
}
```

A continuación se presenta el código generado para el modelo mostrado en la figura 70.

Consumidor.java

```
package // Start of user code
                // Write the package path

                // End of user code

import java.util.Dictionary;
import java.util.Hashtable;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
```

```
import org.osgi.framework.Constants;
import org.osgi.framework.ServiceRegistration;
import org.osgi.service.wireadmin.Consumer;
import org.osgi.service.wireadmin.Wire;
import org.osgi.service.wireadmin.WireConstants;

//Start of user code
// Here write all the imports that's you require for your code

import java.lang.String;

//End of user code

public class Consumidor
	implements Consumer, BundleActivator {

	private Wire[] wires = null;
	private ServiceRegistration serviceRegistration;
	//Start of user code
	// Here write all the attributes you will required;
	//End of user code


	public Consumidor ( ) {
		//Start of user code
		// Here write all the inicialization code;
		//End of user code
	}


	public void start(BundleContext bundleContext) throws Exception {
		Dictionary properties = new Hashtable();
		properties.put(
			Constants.SERVICE_PID,"ConID");
		properties.put(
			WireConstants.WIREADMIN_CONSUMER_FLAVORS,
			new Class[] {
				java.lang.String.class,
			});
		serviceRegistration=bundleContext.registerService(Consumer.class.getName(),
			this, properties);
	}

	public void stop(BundleContext bundleContext) throws Exception {
		serviceRegistration.unregister();
	}

	public void updated(Wire wire, Object obj) {
		if ( obj instanceof Date )
			processjava.lang.String( (java.lang.String) obj );
	}

	public void producersConnected(Wire[] wires) {
		this.wires = wires;
	}

	public void processjava.lang.String (java.lang.String input) {
	// Start of user code
	// Write the code for the manupilation of the object of type java.lang.String
	// End of user code
	}

	//Start of user code
	// Write your own code;
	//End of user code

}
```

Productor.java

```
package //Start of user code
```

```
		//Write the package path

		//End of user code

import java.util.Hashtable;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import org.osgi.service.wireadmin.Producer;
import org.osgi.service.wireadmin.Wire;

//Start of user code for import
// Here write all the imports that's you require for your code ;

import java.lang.String;
import org.osgi.util.position.Position;

//End of user code for import

public class Productor implements Producer, Runnable {

	private Wire wires[];
	private BundleContext context;
	private ServiceRegistration servreg;
	private boolean quit;

	public Productor(BundleContext context) {
		this.context = context;

		servreg = null;
		quit = false;
		new Thread(this).start();
	}

	public synchronized void run() {
		Hashtable ht = new Hashtable();
		ht.put(
			org.osgi.service.wireadmin.WireConstants.WIREADMIN_PRODUCER_FLAVORS,
			new Class[] {
				java.lang.String.class,
				org.osgi.util.position.Position.class,
			});
		ht.put(org.osgi.framework.Constants.SERVICE_PID,
			"ProdID");
		ht.put( org.osgi.framework.Constants.SERVICE_DESCRIPTION,
			"ProductorEjemplo");

		context.registerService(Producer.class.getName(), this, ht);

		while (!quit)
			try {
				for (int i = 0; wires != null && i < wires.length; i++){
					Wire wire=wires[i];
					if(!wire.isConnected() || !wire.isValid())
						continue;
					wire.update(polled(wire));
				}
				wait(150000);
			} catch (InterruptedException ie) {
			}
	}

	public synchronized void consumersConnected(Wire wires[]) {
		this.wires = wires;
	}

	public Object polled(Wire wire) {
		Class clazzes[] = wire.getFlavors();
		for (int i = 0; i < clazzes.length; i++) {
			Class clazz = clazzes[i];
			if (clazz.isAssignableFrom(java.lang.String.class)) {
				java.lang.String obj = new java.lang.String();
```

```
                        //Start of user code
                        // Here write your code for this type;
                        //End of user code
                        return obj;
                }
                if (clazz.isAssignableFrom(org.osgi.util.position.Position.class)) {
                org.osgi.util.position.Position obj = new org.osgi.util.position.Position();
                        //Start of user code
                        // Here write your code for this type;
                        //End of user code
                        return obj;
                }
        }
        return null;
    }

    public void end() {
        this.quit = true;
    }
}
```

11.2. Servicio *WireAdminBinder*

11.2.1. Problemática

Debido a la complejidad del desarrollo de aplicaciones utilizando el servicio *WireAdmin*, fue desarrollado un servicio llamado *WireAdminBinder* para implementar una manera declarativa de construir aplicaciones utilizando el servicio *WireAdmin* [43].

Así como en los servicios declarativos fue creado un lenguaje declarativo llamado WA-DCL (*WireAdmin Declarative Composition Language*). Este está diseñado para soportar la introducción y remoción de productores y consumidores durante la ejecución, así como la composición de productores y consumidores.

Con el *WireAdminBinder* las aplicaciones son descritas de manera declarativa en un descriptor XML. En este trabajo no se detallará el funcionamiento completo de el *WireAdminBinder*, sin embargo es necesario conocer algunos conceptos principales para entender el modelo propuesto:

a) **WireApp**. Representa una aplicación compuesta por un conjunto dinámico de productores, consumidores y sus conexiones.

b) **Bundle**. Es una unidad de despliegue utilizada para entregar *WireApps*. Una o más *WireApps* pueden ser entregadas dentro de un bundle.

c) **WireSet**. Representa un conjunto de alambres (*wires*), que conectan a los productores con los consumidores. Los *WireSets* están definidos en torno a un par de filtros que restringen la selección de los consumidores y productores. Los *WireSets* también definen políticas de remoción del alambrado asociado a ellos.

d) **Property**. Las propiedades especifican propiedades de calidad de servicio utilizado por los productores y alambres para controlar el flujo y aliviar la carga de consumidores.

11.2.2. Perfil UML para el *WireAdminBinder*.

La figura 71 muestra el metamodelo y la figura 72 muestra el perfil UML finales creados para modelar aplicaciones utilizando el servicio *WireAdminBinder*.

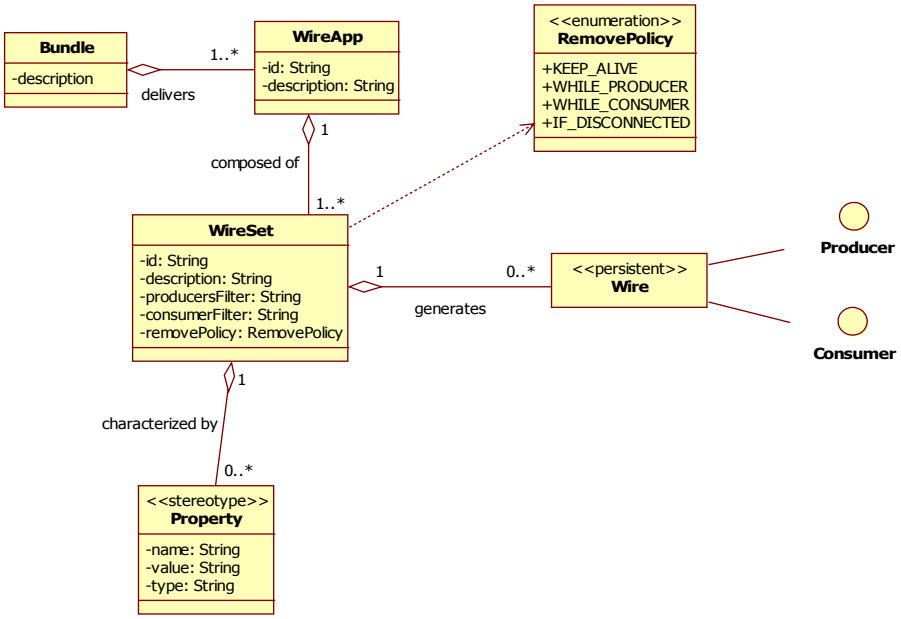

Figura 71: metamodelo del WireAdminBinder

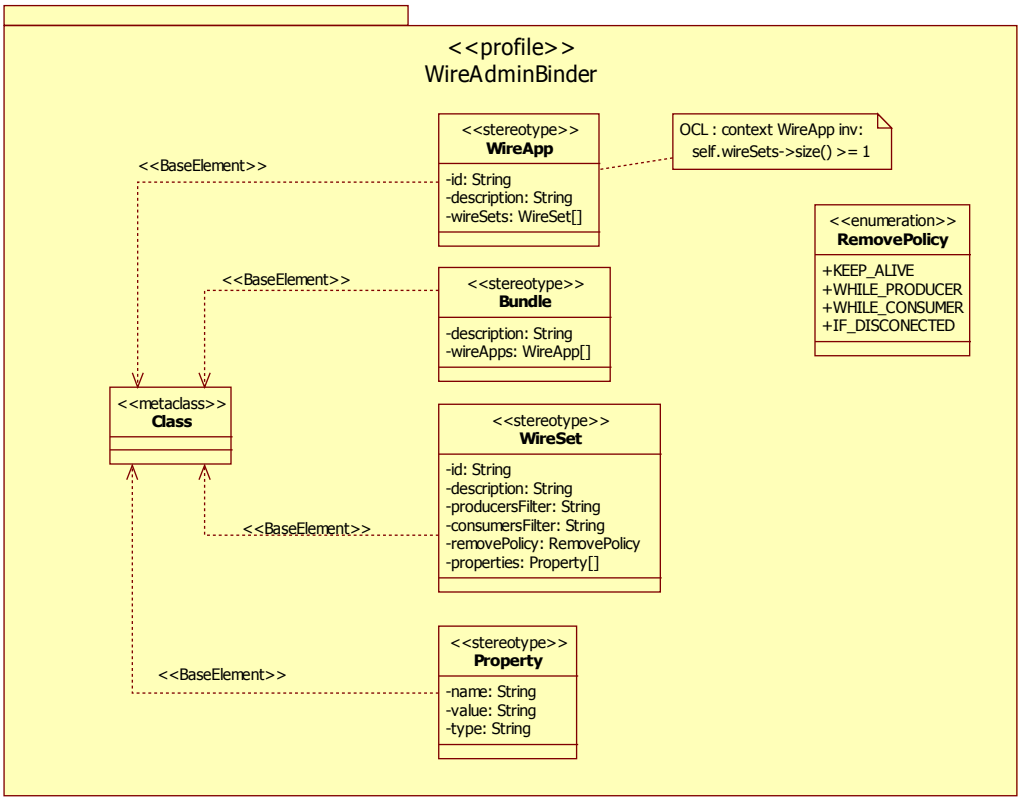

Figura 72: Perfil UML para el WireAdminBinder

11.2.3. Modelos GMF para el *WireAdminBinder*

Se tienen los siguientes modelos:

1. El modelo EMF del dominio (figura 73).
2. El modelo de definición gráfica (figura 74).
3. El modelo de definición de la paleta de herramientas (figura 75).
4. El modelo de mapeo con las restricciones OCL (figura 76).

Los modelos generador de GMF y generador de EMF no se muestran debido que estos son generados a partir de los modelos de mapeo y de dominio respectivamente.

11.2.4. Editor gráfico generado para el *WireAdminBinder*.

Una vez con los modelos completos se generó el editor gráfico para modelar aplicaciones basadas en el servicio *WireAdmin* utilizando el *WireAdminBinder*. En la figura 77 se muestra el editor generado, en esta imagen se puede apreciar un par de *WireApps* cada uno con sus respectivos *WireSets*. A su vez cada *WireSet* puede tener una propiedad.

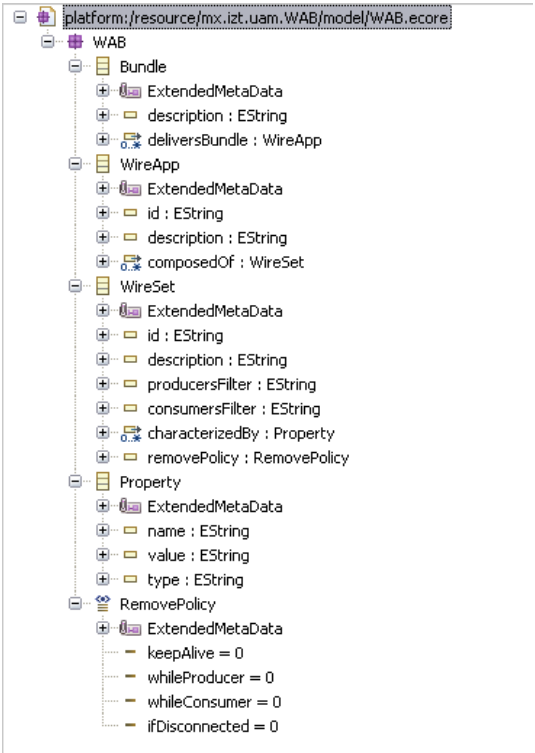

Figura 73: Modelo EMF del dominio para el WireAdminBinder

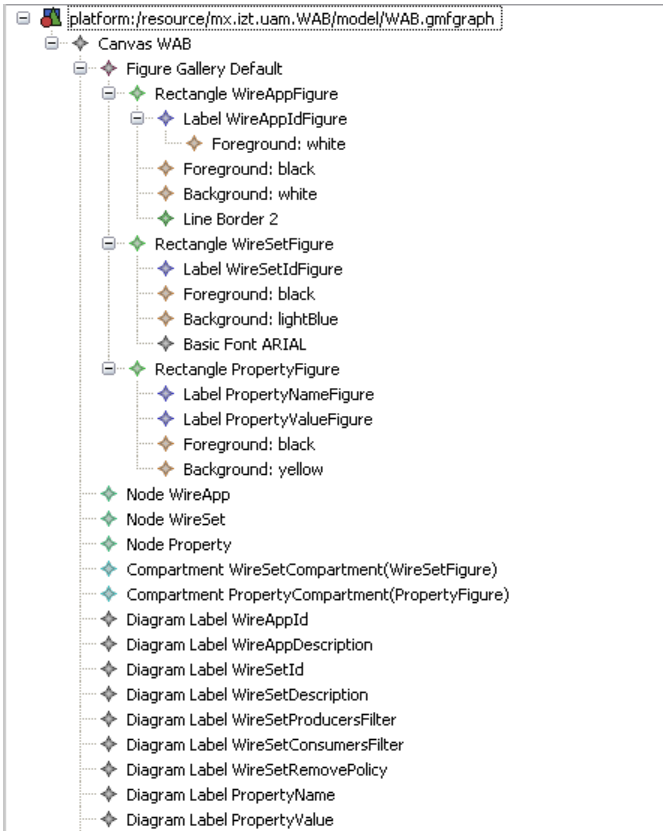

Figura 74: Modelo GMF de definición gráfica para el WireAdminBinder

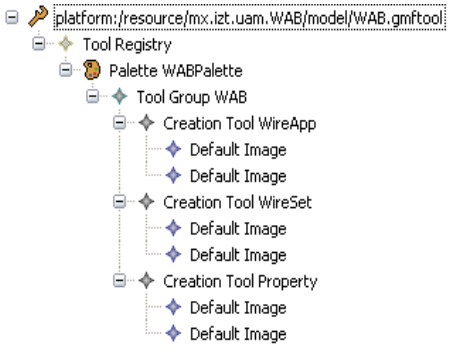

Figura 75: Modelo GMF de definición de la paleta de herramientas para el WireAdminBinder

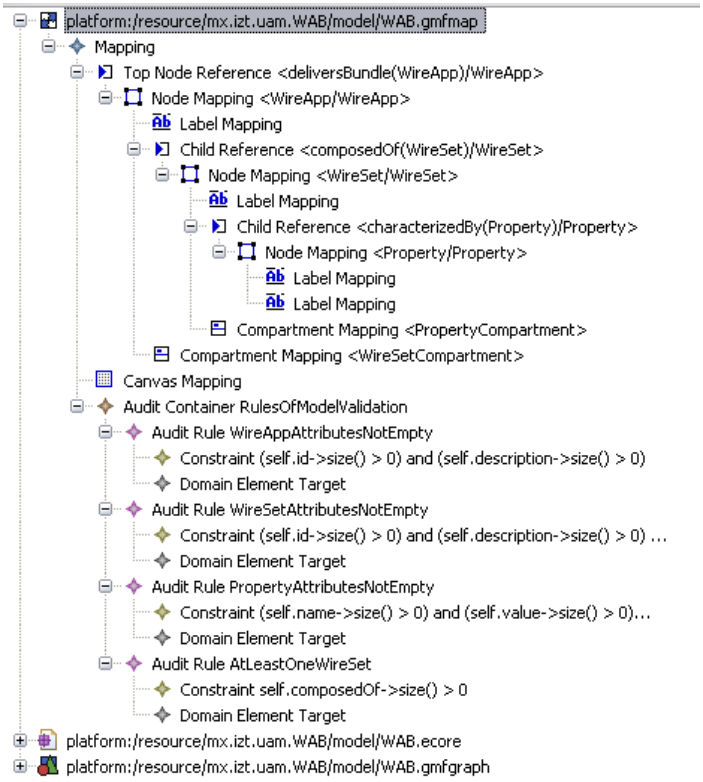

Figura 76: Modelo GMF de mapeo para el WireAdminBinder

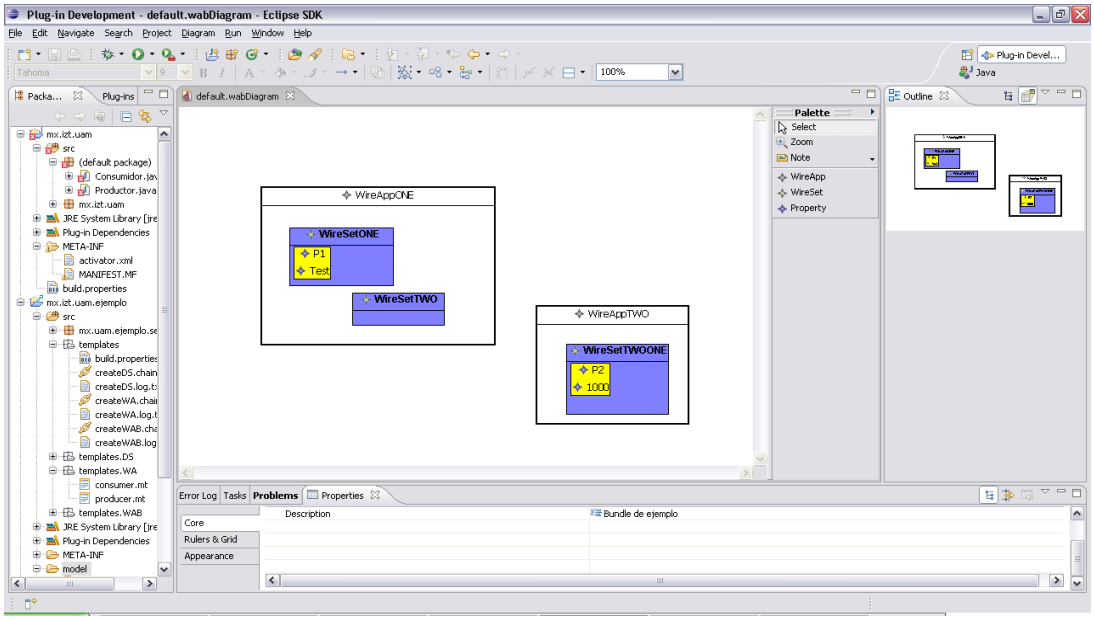

Figura 77: Editor generado con GMF para el WireAdminBinder

11.2.5.Reglas de transformación y plantillas de Acceleo.

En este caso se generará únicamente el descriptor en XML siguiendo la especificación de WA-DCL a partir del modelo. A continuación se presenta el conjunto de reglas de transformación definidas para este problema.

Elemento del modelo	*Código generado*

Bundle	`<?xml version="1.0" encoding="UTF-8"?>`
	
	`<!--`
	`<!DOCTYPE wireadminbinder SYSTEM "wireadminbinder.dtd">`
	`-->`
	
	`<!-- Metadata used by WireAdminBinder, http://www-adele.imag.fr/users/Didier.Donsez/dev/osgi/wireadminbinder/wireadminbinder.jar -->`

Bundle.description	`<bundle description="<Bundle.description>">` `</bundle>`
WireApp.id, WireApp.description	`<wireapp description="<WireApp.description>" id="<WireApp.id>">` `</wireapp>`
WireSet.id	`<wireset id="<WireSet.id>">` `</wireset>`
WireSet.description	`description="<WireSet.description>"`
WireSet.producersFilter	`producersfilter="<WireSet.producersFilter>"`
WireSet.consumersFilter	`consumersfilter="<WireSet.consumersFilter>"`
WireSet.removePolicy	`removepolicy="<WireSet.removePolicy>"`
Property	`<property`
Property.name	`name="<Property.name>"`
Property.value	`value="<Property.value>"`
Property.type	`type="<Property.type>"`

Tabla 10: Reglas de transformación para el WireAdminBinder

Con base en estas reglas se creó una plantilla para generar el descriptor en XML.

Descriptor.mt

```
<%
metamodel http://www.uam.izt.mx/WAB
%>

<%script type="WAB.Bundle" name="descriptor" file="activator.xml"%>
<?xml version="1.0" encoding="UTF-8"?>

<!--
<!DOCTYPE wireadminbinder SYSTEM "wireadminbinder.dtd">
-->

<!-- Metadata used by WireAdminBinder, http://www-
adele.imag.fr/users/Didier.Donsez/dev/osgi/wireadminbinder/wireadminbinder.jar -->
<bundle description="<%description%>">
        <%for (deliversBundle){%>
        <wireapp description="<%description%>" id="<%id%>">
                <%for (composedOf){%>
                <wireset
                        id="<%id%>"
                        description="<%description%>"
```

```
                producersfilter="<%producersFilter%>"
                consumersfilter="<%consumersFilter%>"
                removepolicy="<%removePolicy%>"
        >
                <%for (characterizedBy){%>
                <property
                        name="<%name%>"
                        value="<%value%>"
                        type="<%type%>"
                />
                <%}%>
        </wireset>
        <%}%>
    </wireapp>
    <%}%>
</bundle>
```

El descriptor generado por la aplicación a partir del modelo de la figura 77 es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
<!DOCTYPE wireadminbinder SYSTEM "wireadminbinder.dtd">
-->

<!-- Metadata used by WireAdminBinder, http://www-
adele.imag.fr/users/Didier.Donsez/dev/osgi/wireadminbinder/wireadminbinder.jar -->
<bundle description="Bundle de ejemplo">
    <wireapp description="Aplicacion de prueba uno" id="WireAppONE">
        <wireset
            id="WireSetONE"
            description="Set de alambres UNO"
            producersfilter="Filtro LDAP para productores"
            consumersfilter="Filtro LDAP para consumidores"
            removepolicy="keepAlive"
        >
            <property
                name="P1"
                value="Test"
                type="String"
            />
        </wireset>
        <wireset
            id="WireSetTWO"
            description="Set de alambres DOS"
            producersfilter="Filtro LDAP para productores"
            consumersfilter="Filtro LDAP para consumidores"
            removepolicy="keepAlive"
        >
        </wireset>
    </wireapp>
    <wireapp description="Aplicacion de prueba dos" id="WireAppTWO">
        <wireset
            id="WireSetTWOONE"
            description="Set de alamabres UNO"
            producersfilter="Filtro LDAP para productores"
            consumersfilter="Filtro LDAP para consumidores"
            removepolicy="keepAlive"
        >
            <property
                name="P2"
                value="1000"
                type="Integer"
            />
        </wireset>
    </wireapp>
</bundle>
```

12. Referencias.

[1] Humberto Cervantes y Richard S. Hall, *"Services Oriented Concepts and Technologies",* (ISBN 1-59140-426-6) editado por Zoran Stojanovic and Ajantha Dahanayake, Idea Group Publishing, Holanda, 2005, pag. 1-26.

[2] Ed Ort, *"Service-Oriented Architecture and Web Services: Concepts, Technologies, and Tools",* Sun Microsystems, http://java.sun.com/developer/technicalArticles/WebServices/soa2/, E.U., 2005, última fecha de consulta 01/JUL/07.

[3] Duane Nickull, coautores: Mike Connor, Matthew MacKenzie, *"Service Oriented Architecture",* Adobe Systems Inc. Whitepaper, E.U., 2005.

[4] James McGovern, Sameer Tyagi, Michael Stevens, Sunil Mathew, *"Java Web Services Architecture",* editado por Elsevier Science, E.U., 2003, pag. 35-62

[5] Jon Shemitz, *".Net Architecture"*, paper en línea, http://www.midnightbeach.com/dotNetArchitecture.2002.html, E.U., 2006, última fecha de consulta 01/JUL/07.

[6] Guy Bieber, Lead Architect, Jeff Carpenter, *"Introduction to Service-Oriented Programming*", whitepaper en línea, http://www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf, E.U., 2001, última fecha de consulta 01/JUL/07.

[7] Mike P. Papazoglou y Willem-Jan van den Heuvel, *"Service-Oriented Computing: State of the Art and Open Research Issues"*, publicado por Telematica Instituut, Holanda, 2003.

[8] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, Frank Leymann, "*Service-Oriented Computing: Research Roadmap",* artículo en línea, ftp://ftp.cordis.europa.eu/pub/ist/docs/directorate_d/st-ds/services-research-roadmap_en.pdf, Holanda, 2006, última fecha de consulta 01/JUL/07.

[9] BEA Systems, IBM, IONA, Oracle, SAP AG, Siebel Systems, Sybase, "*Service Component Architecture",* artículo en línea, http://www-128.ibm.com/developerworks/library/specification/ws-sca/ , E.U., 2005, última fecha de consulta 01/JUL/07.

[10] Object Managment Group, "*MDA Guide Version 1.0.1",* editado por la OMG, E.U., 2003.

[11] Alan Brown, *"An introduction to Model Driven Architectures"*, artículo en línea, IBM, http://www.ibm.com/developerworks/rational/library/3100.html, E.U., 2004, última fecha de consulta 01/JUL/07.

[12] Joaquin Miller y Jishnu Mikerji, "*Model Driven Architecture",* MDA adopted as OMG's Technical Architecture, OMG, http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01, E.U., 2001, última fecha de consulta 01/JUL/07.

[13] Mike Rosen, "*MDA, SOA and Technology Convergence",* whitepaper en línea, Azora technologies, http://www.azoratech.com/PDF/AZORA_WP-MDA_and_SOA-(English).pdf, E.U., 2004, última fecha de consulta 01/JUL/07.

[14] Lidia Fuentes y Antonio Vallecillo, "*Una Introducción a los Perfiles UML",* Novática: Revista de la asociación de técnicos de informática, ISSN 0211-2124, Nº. 168, (Ejemplar dedicado a: UML e Ingeniería de Modelos), España, 2004 , pags. 6-11.

[15] Anneke Kleppe, Jos Warmer, Wim Bast, "*MDA Explained: The Model Driven Architecture: Practice and Promise",* publicado por Addison-Wesley, ISBN:032119442X, Holanda, 2003.

[16] Mikko Kontio, "*Architectural manifesto: Choosing MDA tools",* artículo en línea, IBM, http://www.ibm.com/developerworks/webservices/library/wi-arch18.html, E.U., 2005, última fecha de consulta 01/JUL/07.

[17] Object Management Group, "*Unified Modeling Language: Superstructure",* version 2.0, editado por la OMG, E.U., 2004, pags. 633-654.

[18] Eclipse Development Resources, "*The Eclipse Modeling Framework (EMF)"*, documentación en línea, Eclipse, http://dev.eclipse.org/viewcvs/indextools.cgi/*checkout*/org.eclipse.emf/doc/org.eclipse.emf.doc/references/overview/EMF.html, E.U., 2005, última fecha de consulta 01/JUL/07.

[19] Eclipse Development Resources, "*GMF Tutorial",* documentación en línea, Eclipse http://wiki.eclipse.org/index.php/GMF_Tutorial, E.U., 2006, última fecha de consulta 01/JUL/07.

[20] Jon Oldevik, "*UMT. UML Model Transformation Tool. Overview and user guide documentation",* documentación en línea, http://umt-qvt.sourceforge.net/, E.U., 2004, última fecha de consulta 01/JUL/07.

[21] ModelBased.net, "*MDA Tools*", documentación en línea, http://www.modelbased.net/mda_tools.html, E.U., 2006, última fecha de consulta 01/JUL/07.

[22] AndroMDA.org. http://www.andromda.org/ , última fecha de consulta 01/JUL/07.

[23] OpenMDX.org. http://www.openmdx.org/index.html , última fecha de consulta 01/JUL/07.

[24] Eclipse Development Resources, "*GMF GenModel Hints",* documentación en línea, Eclipse, http://wiki.eclipse.org/index.php/GMF_GenModel_Hints, E.U., 2006, última fecha de consulta 01/JUL/2007.

[25] Frederic Plante, "*Introducing the GMF Runtime",* artículo en línea, IBM, http://www.eclipse.org/articles/Article-Introducing-GMF/article.html, E.U., 2006, última fecha de consulta 01/JUL/07.

[26] International conference on object oriented programming, systems, languages and applications, OOPSLA 2006, "*GMF Principles*", Tutoriales online, http://www.oopsla.org/2006/submission/tutorials/introduction_to_tutorials.html, E.U., 2006, última fecha de consulta 01/JUL/07.

[27] Adrian Powell, "*Model with the Eclipse Modeling Framework, Part 1",* artículo en línea, IBM, http://www-128.ibm.com/developerworks/opensource/library/os-ecemf1/, E.U., 2004, última fecha de consulta 01/JUL/07.

[28] Eclipse Development Resources, "*Generating an EMF Model",* artículo en línea, Eclipse, http://dev.eclipse.org/viewcvs/indextools.cgi/*checkout*/org.eclipse.emf/doc/org.eclipse.emf.doc/tutorials/clibmod/clibmod.html, E.U., 2006, última fecha de consulta 01/JUL/07.

[29] Chris Aniszczyk, "*Using GEF with EMF*", artículo en línea, IBM, http://www.eclipse.org/articles/Article-GEF-EMF/gef-emf.html, E.U., 2005, última fecha de consulta 01/JUL/07.

[30] Cédric Brun, "*Using Acceleo with GMF",* documentación en línea, OBEO,

http://www.acceleo.org/pages/using-acceleo-with-gmf/en, Francia, 2006, última fecha de consulta 01/JUL/07.

[31] Jonathan Musset, Etienne Juliot, Stephan Lacrampe, "*Acceleo 1.1 User Guide",* publicado por OBEO, Francia, 2006, pags. 6-52.

[32] Adrian Powell, "*Model with the Eclipse Modeling Framework, Part 2: Generate code with Eclipse's Java Emitter Templates",* artículo en línea, IBM, http://www-128.ibm.com/developerworks/ibm/library/os-ecemf2/, E.U., 2004, última fecha de consulta 01/JUL/07.

[33] Eclipse Graphical Editing Framework (GEF), http://www.eclipse.org/gef/, última fecha de consulta 01/JUL/07.

[34] Merlin Generator, http://sourceforge.net/projects/merlingenerator/, última fecha de consulta 01/JUL/07.

[35] OSGi Alliance, "*OSGi Service Platform, Core Specification. Release 4"*, publicado por OSGi Alliance, E.U., 2005, pags. 27-78.

[36] OSGi Alliance, "*OSGi Service Platform, Service Compendium. Release 4*", publicado por OSGi Alliance, E.U., 2005, pags. 277-312.

[37] OSGi Alliance, "*OSGi Service Platform, Service Compendium. Release 4*", publicado por OSGi Alliance, E.U., 2005, pags. 151-194.

[38] Humberto Cervantes, Richard Hall, "*ServiceBinder. Automatic service dependency manager in OSGi",* documentación online, http://gravity.sourceforge.net/servicebinder/, Francia, 2004, última fecha de consulta 01/JUL/07.

[39] Ivar Jacobson, Grady Booch y James Rumbaugh, "*The Unified Software Development Process",* publicado por Addison Wesley, E.U., 1999.

[40] Object Managment Group, "*UML 2.0 OCL Specification",* editado por la OMG, E.U., 2003.

[41] Javier Muñoz, "*Pervasive Systems Development with Model Driven Architecture",* conferencia: UML 2004, Portugal, 2004, versión en línea http://ctp.di.fct.unl.pt/UML2004/DocSym/JavierMunozUML2004DocSym.pdf .

[42] Spring Framework, http://www.springframework.org/, última fecha de consulta 01/JUL/07.

[43] Humberto Cervantes, Didier Donsez, Lionel Touseau, "*A declarative composition language for dynamic sensor-based applications",* artículo no publicado, México-Francia, 2006.

[44] Clemens Szyperski, "*Component Software: Beyond Object-Oriented Software"*, publicado por ACM Press, E.U., 1998.

[45] Javier Muñoz, Vicente Pelechano, "*MDA vs factorías de software"*, Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones (DSDM), España, 2005.

[46] Bran Selic, *"The pragmatics of model driven development",* IBM Rational Software, publicado por la IEEE Computer Society, Canadá, 2003.

UNIVERSIDAD AUTÓNOMA METROPOLITANA

Casa abierta al tiempo

División de Ciencias Básicas e Ingeniería

Un enfoque MDA para el desarrollo de aplicaciones basadas en un modelo de componentes orientados a servicios

Idónea comunicación de resultados que presenta

Néstor A. Riba Zárate

PARA OBTENER EL GRADO DE MAESTRO EN CIENCIAS

Asesor:

Dr. Humberto Cervantes Maceda

Julio del 2007

Sinodales

Presidente: Dr. Héctor A. Durán Limón

Secretario: Dra. Elizabeth Pérez Cortés

Vocal: Dr. Humberto Cervantes Maceda