UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Arquitectura de Servicios Web basada en modelos: especificación gráfica y derivación automática del código.

Autores: Palumbo María Florencia - Tunessi María Silvina.

Director: Dra. Claudia Pons.

Codirector:

Resumen

MDA provee un conjunto de herramientas para especificar un sistema independientemente de la plataforma de implementación, elegir una plataforma para el sistema, y transformar las especificaciones de los sistemas a la misma. Todo esto se complementa con los objetivos de portabilidad, interoperabilidad y reusabilidad. Dentro de la aproximación MDA, tiene especial relevancia la existencia de transformaciones entre modelos. MDA permite reducir los costos de desarrollo de software, adaptarlo rápidamente a los cambios tecnológicos y a cambios en los requisitos, siempre manteniendo la consistencia entre los modelos y el código del software y así dando el protagonismo de la conducción del diseño y desarrollo a los modelos. Spring es un framework basado en J2EE que implementa el modelo MVC. Su gran ventaja es la modularidad definiendo funcionalidad e integrando diferentes tecnologías. Y los Web Services nos permiten comunicar diferentes aplicaciones de forma distribuida entre sí. Decidimos que MDA nos planteó una ágil y eficiente manera de definir nuestros modelos con la utilización del lenguaje UML, agregando información específica para poder generar la estructura de servicios y así mediante una serie de transformaciones derivar al esqueleto de servicios, interfaces y daos en el código correspondiente. De esta manera se logra reducir tiempos, errores y prestar una disposición para el buen uso que ofrece el framework.

Palabras Claves

- ***Ingeniería de Software***
- ***MDA:*** *Arquitectura dirigida por modelos.*
- ***PIM:*** *Modelo independiente de la plataforma.*
- ***PSM:*** *Modelo dependiente de la plataforma.*
- ***UML:*** *Lenguaje Unificado de Modelado.*
- ***UML Profile:*** *Extender, decorar metamodelado.*
- ***OCL:*** *Consultar y validar un modelo.*
- ***ATL:*** *Lenguaje de transformación de modelos.*
- ***Framework Spring.***
- ***Arquitectura Web Services.***

Trabajos Realizados

- En nuestra tesis hemos analizado y estudiado los conceptos de MDA y puesto en práctica en una arquitectura modular y escalable altamente recomendable, como es la arquitectura SOA, orientada a servicios utilizando el potente Framework Spring para el desarrollo. Realizamos una herramienta de software Implementando un perfil "UService" con perfiles UML2, para definir, decorar y validar nuestros modelos y compusimos un par de transformaciones a PIM y finalmente a código utilizando ATL y MOFScript respectivamente.
- Se implementaron casos de ejemplos que incluyen la definición de modelos uml utilizando la arquitectura de servicios y realizando su transformación automática a código fuente JAVA.

Conclusiones

El objetivo se planteó en función de promover la 'correcta' utilización de la arquitectura y el 'buen diseño' de sistemas orientados a servicios, como también la agilización en reducir tiempos del desarrollo del software, contando con varios pasos resueltos de diseño y organización al comenzar la codificación. MDA nos permitió cumplir y desarrollar el trabajo de una manera modular y con un amplísimo abanico de posibilidades, demostrándonos que finalmente se puede lograr la "Arquitectura dirigida por Modelos", de forma segura, potente y provechosa para el desarrollo de software cotidiano.

Trabajos Futuros

- Optimizar la generación a código JAVA para mantener la trazabilidad con respecto a los cambios realizados.
- Realizar un editor gráfico para generar los diagramas UML y que nos permita visualizar el modelo de salida de una manera más amigable.
- Agregar módulos a la transformación que contengan funcionalidad para las vistas y controladores del MVC utilizados por Spring.
- Analizar la posibilidad de hacer reingeniería inversa de manera que el modelo UML se mantenga actualizado luego de hacer modificaciones en el código fuente.

Fecha de la presentación: **Octubre de 2013**

Agradecimientos

Agradezco en especial a mis padres por darme libertad de elegir el camino en mi carrera, sin exigirme nada y con el eterno apoyo en estos años. Este logro mío es de ustedes...

A los amigos que encontré en la facultad, los que hoy no veo pero los llevo en el corazón con el más lindo recuerdo y aquellos que son entrañables en la compañía de mi vida.. Mis grandes amigas que están siempre... Naty, Noe, Yani, Nancy.

A Ari y al Gordo que fueron las mejores personas que puedo haber encontrado en mi primer trabajo, el cual fue un placer compartirlo juntos, donde había una demostración de compañerismo diario, haciendo el trabajo divertido, agradable y siempre aprendiendo de ustedes, pero lo más importante sembrando una verdadera amistad... A Silvi, mi compañera de todos estos años, estudiando tantas materias juntas, acompañándonos siempre en las buenas y en las malas, gracias porque juntas llegamos hasta acá y las dos sabemos cuán arduo fue el camino...

Al amor de mi vida, por estar cerca en este último tramo y acompañarme en todo con tanto amor...

Florencia.

Agradezco a mi familia, a mis amigos, compañeros de trabajo y de facultad por haberme acompañado y estado presentes en toda mi carrera. A Flor, porque más allá de ser mi compañera de estudio durante gran parte de la carrera, encontré en ella una amiga que me apoyo en todos los momentos buenos y malos que me tocó vivir.

Silvina.

A Claudia Pons por apoyarnos desde el principio con tan buena onda y predisposición, y mostrándonos el camino más simple para desarrollar y finalizar nuestro trabajo… Un placer tenerte como nuestra directora de tesis y una admiración profesional… Y a todos los profesores dedicados en esta carrera que siempre han orientado a darnos las herramientas fundamentales para poder desarrollarnos en esta profesión tan cambiante, aportándonos conocimientos de tan alto valor...

Muchas Gracias.

Indice

Capítulo 4

Indice de Figuras

Capítulo1

Introducción

1.1 Motivación

- Model Driven Architecture (**MDA**) es una aproximación definida por el Object Management Group (**OMG**) [OMG], mediante la cual el diseño de los sistemas se orienta a modelos.

El desarrollo orientado a modelos permite una alta flexibilidad en la implementación, integración, mantenimiento, prueba y simulación de los sistemas. Una de las ideas principales por la que surge MDA es separar la especificación de los sistemas de los detalles de su implementación en una determinada plataforma. MDA provee un conjunto de herramientas para especificar un sistema independientemente de la plataforma de implementación, determinar dichas plataformas, elegir una para el sistema, y transformar las especificaciones de los sistemas a la plataforma elegida. Todo esto se complementa con los objetivos de portabilidad, interoperabilidad y reusabilidad.
La independencia propuesta por MDA se consigue mediante una catalogación de modelos que permiten especificar el sistema desde diferentes puntos de vista.

Asimismo, dentro de la aproximación MDA, tiene especial relevancia la existencia de transformaciones entre modelos. De esta forma, se definen los mecanismos necesarios para convertir un modelo de un tipo a otro, siendo ambos representaciones del mismo sistema. Se puede transformar por ejemplo, un modelo **PIM** (Platform Independent Model) en uno o varios modelos **PSM** (Platform Specific Model).

Como resultado final se obtiene idealmente la implementación del sistema en la plataforma seleccionada (código fuente), entre otros artefactos.

De esta forma, MDA permite reducir los costos de desarrollo de software, adaptarlo rápidamente a los cambios tecnológicos y a cambios en los requisitos, siempre manteniendo la consistencia entre los modelos y el código del software y así dando el protagonismo de la conducción del diseño y desarrollo a los modelos como “debería ser” aunque no estamos acostumbrados a mantener ya que, en la actualidad generalmente, la conexión entre los diagramas y el código se va perdiendo gradualmente mientras se progresa en la fase de codificación sin reflejar los cambios (como por ejemplo de

requerimientos) en el modelo y perdiendo así la documentación del diseño final derivado mediante el proceso de desarrollo como el hilo de la lógica del sistema.

- **Spring** es un framework basado en **J2EE** que implementa el modelo **MVC**. Este framework fue lanzado bajo la licencia Apache 2.0 Licence en junio de 2003 por Rod Jonhson, su creador. Durante su lanzamiento los desarrolladores consideraban Spring un avance con respecto al modelo de programación que implementaba Enterprise JavaBeans (EJB). Gran parte de la popularidad de Spring se basa en la implementación del modelo MVC que hace uno de sus módulos. Spring logró popularizar en su momento buenas técnicas de programación al estar implementado usando distintos patrones de diseño. Es código abierto.

- Una ventaja de Spring es su modularidad, pudiendo usar algunos de los módulos sin comprometerse con el uso del resto.

El objetivo central de Spring es permitir que objetos de negocio y de acceso a datos sean reusables, no atados a servicios J2EE específicos. Estos objetos pueden ser reutilizados tanto en entornos J2EE (web o EJB), aplicaciones standalone, entornos de pruebas,... sin ningún problema.
Otra característica especial que tiene Spring es su capacidad para actuar como un "pegamento" tecnológico facilitando la integración de tecnologías que por sí solas sería muy complicado integrar.

Los **WebServices** nos permiten comunicar diferentes aplicaciones de forma distribuida entre sí, esta tecnología empieza a ser muy demandada por las empresas del mundo de la programación, ya que permiten centralizar de forma muy efectiva o distribuir los puntos de acceso de diferentes aplicaciones según convenga de una forma muy dinámica.

A esta tecnología se le puede dar diversas utilidades, pero generalmente pensaremos en un servidor que contiene información, a la que el servidor permite el acceso por parte de otras aplicaciones a través de un WebServices. Esta es la forma que nos encontraremos más habitualmente, pero evidentemente la tecnología nos permite mucho más.

En esta tesis vamos a desarrollar un plugin para nuestro WebServices con Spring-WebServices, por varias razones, la primera la de desarrollar aplicaciones con este Framework, ya que las prestaciones y opciones que nos brinda van a repercutir directamente en la calidad final de nuestra aplicación.

Implementar el diseño de servicios, con las interfaces intervinientes, los archivos de definición y demás elementos que plantea el buen uso de Spring resulta engorroso y siempre la realización manual de los mismos tiende a generar errores y al uso incorrecto de

la arquitectura. Decidimos que **MDA** nos plantea una ágil y eficiente manera de definir nuestros modelos con la utilización del lenguaje **UML** (Unified Modeling Language) que generalmente es elegido para realizar los modelos en **MDD** ya que, se definieron estándares de lenguajes de transformación para el mismo y poder generar la estructura de servicios agregando el perfil en UML, "UML Service" y así mediante una serie de transformaciones derivar al esqueleto de servicios, interfaces y daos en el código correspondiente. De esta manera se logra reducir tiempos, errores y prestar una disposición para el buen uso que ofrece el framework. Dando lugar a poder mantener la dirección del diseño del software por modelos y sostener la consistencia con el código resultante; y así poder modificar y/o agregar nuevas clases y servicios teniendo la independencia de los modelos (**UML** y **UML Service**).

1.2 Objetivos

El objetivo de este trabajo es la realización de una herramienta de software que permita definir modelos de manera gráfica utilizando UML2 y UML2 Service, donde este último es un nuevo perfil de UML2 que se desarrollará para modelar la arquitectura de servicios del Framework Spring, y derivar código fuente a partir de los mismos.

Capítulo2

Comenzamos el capítulo 3, introduciendo a los conceptos de la Arquitectura Orientada a Servicios (**SOA**) y sus beneficios. Continuamos presentando el framework **Spring**, mediante el cual desarrollamos flexiblemente nuestros **webServices** y explicando el desarrollo modular que nos ofrece, la fácil integración de varias tecnologías en cada etapa y el manejo particularmente interesante de las dependencias de los componentes entre sí, mediante el patrón **Inversión de Control** (**IoC**), dando ejemplos del mismo.
Luego introducimos a la arquitectura de software dirigido por modelos (**MDA**) la cual es la columna vertebral de este proyecto. La misma se basa en el modelado standard **UML,** el cual detallamos. Explicamos los **Perfiles**, que se usan como mecanismo de extensión de modelos UML para añadir información semántica y expresar detalles específicos de la plataforma. Fundamentamos OBJECT CONSTRAINT LANGUAGE (**OCL**) que utilizamos para las validaciones del modelo.
Continuamos explicando los diferentes tipos y niveles de modelos en MDA, (**CIM, PIM, PSM**) y el significado de Meta-modelos, la construcción e importancia de los mismos. El mecanismo de transformación entre los modelos y los distintos lenguajes existentes, particularizando **ATL** como referente de nuestro trabajo.
Explicamos también las diferentes herramientas tecnológicas involucradas.
En el capítulo 4, comienza la implementación de la solución al objetivo propuesto, planteando un caso de estudio concreto y detallando paso a paso el desarrollo de la misma. Mostrando las pruebas en cada etapa y concluyendo con la derivación concreta a código Java.
Finalizamos en los capítulos 5 y 6 con las conclusiones obtenidas del estudio y los trabajos a futuro ideales a resolver.

Capítulo 3
Conceptos Básicos

3.1 Arquitectura orientada a servicios

La **arquitectura orientada a servicios de cliente** (en inglés Service Oriented Architecture), es un concepto de arquitectura de software que define la utilización de servicios para dar soporte a los requisitos del negocio.

Permite la creación de sistemas de información altamente escalables que reflejan el negocio de la organización, a su vez brinda una forma bien definida de exposición e invocación de servicios (comúnmente pero no exclusivamente servicios web), lo cual facilita la interacción entre diferentes sistemas propios o de terceros.

SOA define las siguientes capas de software:

Aplicaciones básicas - Sistemas desarrollados bajo cualquier arquitectura o tecnología, geográficamente dispersos y bajo cualquier figura de propiedad;

De exposición de funcionalidades - Donde las funcionalidades de la capa aplicativa son expuestas en forma de servicios (generalmente como servicios web);

De integración de servicios - Facilitan el intercambio de datos entre elementos de la capa aplicativa orientada a procesos empresariales internos o en colaboración;

De composición de procesos - Que define el proceso en términos del negocio y sus necesidades, y que varía en función del negocio;

De entrega - Donde los servicios son desplegados a los usuarios finales.

SOA proporciona una metodología y un marco de trabajo para documentar las capacidades de negocio y puede dar soporte a las actividades de integración y consolidación.

3.1.1 Terminología

Término	Definición / Comentario
Servicio	Una función sin estado, auto-contenida, que acepta una(s) llamada(s) y devuelve una(s) respuesta(s) mediante una interfaz bien definida. Los servicios pueden también ejecutar unidades discretas de trabajo como serían editar y procesar una transacción. Los servicios no dependen del estado de otras funciones o procesos. La tecnología concreta utilizada para prestar el servicio no es parte de esta definición. Existen servicios asíncronos en los que una solicitud a un servicio crea, por ejemplo, un archivo, y en una segunda solicitud se obtiene ese archivo.
Orquestación	Secuenciar los servicios y proveer la lógica adicional para procesar datos. No incluye la presentación de los datos. Coordinación.
Sin estado	No mantiene ni depende de condición pre-existente alguna. En una SOA los servicios no son dependientes de la condición de ningún otro servicio. Reciben en la llamada toda la información que necesitan para dar una respuesta. Debido a que los servicios son "sin estado", pueden ser secuenciados (orquestados) en numerosas secuencias (algunas veces llamadas tuberías o pipelines) para realizar la lógica del negocio.
Proveedor	La función que brinda un servicio en respuesta a una llamada o petición desde un consumidor.
Consumidor	La función que consume el resultado del servicio provisto por un proveedor

3.1.2 Diseño y desarrollo de SOA

La metodología de modelado y diseño para aplicaciones SOA se conoce como análisis y diseño orientado a servicios. La arquitectura orientada a servicios es tanto un marco de trabajo para el desarrollo de software como un marco de trabajo de implementación. Para que un proyecto SOA tenga éxito los desarrolladores de software deben orientarse ellos mismos a esta mentalidad de crear servicios comunes que son orquestados por clientes o middleware para implementar los procesos de negocio. El desarrollo de sistemas usando SOA requiere un compromiso con este modelo en términos de planificación, herramientas e infraestructura.

Cuando la mayoría de la gente habla de una arquitectura orientada a servicios están hablando de un juego de servicios residentes en Internet o en una intranet, usando servicios web. Existen diversos estándares relacionados a los servicios web. Incluyen los siguientes:

XML, HTTP, SOAP, REST, WSDL, UDDI

Hay que considerar, sin embargo, que un sistema SOA no necesariamente necesita utilizar estos estándares para ser "Orientado a Servicios" pero es altamente recomendable su uso.

En un ambiente SOA, los nodos de la red hacen disponibles sus recursos a otros participantes en la red como servicios independientes a los que tienen acceso de un modo estandarizado. La mayoría de las definiciones de SOA identifican la utilización de Servicios Web (empleando SOAP y WSDL) en su implementación, no obstante se puede implementar SOA utilizando cualquier tecnología basada en servicios.

Lenguajes de alto nivel

Los lenguajes de alto nivel como BPEL o WS-Coordination llevan el concepto de servicio un paso adelante al proporcionar métodos de definición y soporte para flujos de trabajo y procesos de negocio.

3.1.2 .1 Diferencias con otras arquitecturas

Al contrario de las arquitecturas orientadas a objetos, las SOAs están formadas por servicios de aplicación débilmente acoplados y altamente interoperables. Para comunicarse entre sí, estos servicios se basan en una definición formal independiente de la plataforma subyacente y del lenguaje de programación (p.ej., WSDL). La definición de la interfaz encapsula (oculta) las particularidades de una implementación, lo que la hace independiente del fabricante, del lenguaje de programación o de la tecnología de desarrollo (como Plataforma Java o Microsoft .NET). Con esta arquitectura, se pretende que los componentes de software desarrollados sean muy reutilizables, ya que la interfaz se define siguiendo un estándar; así, un servicio C# podría ser usado por una aplicación Java. En este sentido, ciertos autores definen SOA como una Súper-Abstracción.

3.1.2 .2 Beneficios

Los beneficios que puede obtener una organización que adopte SOA son:

- Mejora en los tiempos de realización de cambios en procesos.
- Facilidad para evolucionar a modelos de negocios basados en tercerización.
- Facilidad para abordar modelos de negocios basados en colaboración con otros entes (socios, proveedores).
- Poder para reemplazar elementos de la capa aplicativa SOA sin disrupción en el proceso de negocio.
- Facilidad para la integración de tecnologías disímiles.

3.1.3 SOA: La Tecnología de Información Enfocada al Negocio

Los días en los que una empresa podía operar eficientemente con aplicaciones independientes, corriendo sobre sistemas separados y desconectados, terminaron. La inmediatez, exactitud y seguridad de los datos, de un extremo de un proceso de negocio hasta el otro, es ahora un mandato del negocio. Las organizaciones que pueden cumplirlo, tienen una ventaja competitiva diferente.

Pero la integración, solo por integrar, no es el único objetivo. El objetivo final de un sistema integrado es que las compañías sean capaces de desplazar su enfoque y recursos del mantenimiento de aplicaciones separadas hacia el desarrollo de procesos de negocios que cubran toda la empresa, basados en el servicio al cliente. Más aún, la flexibilidad de un sistema unificado hace posible cambiar esos procesos en respuesta a las rápidamente cambiantes necesidades del negocio.

Software aplicativo construido para ser independiente

El requerimiento para tal infraestructura surge de la evolución del software de negocio. Al comienzo, el desarrollo de aplicaciones de negocio se enfocaba en necesidades específicas del negocio: contabilidad, compras, nómina, pedidos. Cada aplicación era construida sin tener en cuenta otros sistemas de la empresa, ni cómo comunicarse con ellos. Dado que las aplicaciones eran "auto-suficientes", la información común a la empresa completa (tal como la dirección de un cliente) y funciones de negocio comunes (tal como "encontrar un cliente"), aparecían en, y requerían de codificación compleja para todos o varios de los sistemas independientes.

Como resultado, los diferentes sistemas de Tecnología de Información de la mayor parte de las empresas de hoy, no pueden tener acceso ni procesar datos, unos con otros. Un proceso único de negocio (tal como "ventas, pedidos a bodega o entregas, a cuentas por pagar") que se tomaría segundos si los sistemas se pudieran comunicar, se puede tomar semanas.

Pero, ¿qué puede una empresa hacer? Podría llevar a cabo inversiones astronómicas en hardware, software y recurso humano involucrado en ejecutar cada una de sus aplicaciones separadas, sin poner en riesgo la información ni procesos de negocio de ninguna de las aplicaciones que requiere operar.

Con SOA, una empresa puede mantener sus inversiones en sistemas actuales y a las personas necesarias para mantenerlos. Evita continuos y costosos proyectos de "integración", ya que las actualizaciones a cualquiera de las aplicaciones son transparentes para las demás. La información del negocio está disponible al instante, permitiendo mejores decisiones de negocio y relaciones mejoradas con clientes y aliados de negocios.

Para muchos, SOA es una solución prometedora para el problema de la integración. El reto es cómo llegar allá.

¿Qué nos promete esta característica de la orientación a servicios? Pues una vez que dispongamos de un conjunto de servicios básicos, de bajo nivel, que implementen la lógica de negocio de nuestra organización, no tendremos que volver a escribir de nuevo estos programas.

La mejor forma de ver el concepto de composición es con un ejemplo. Imaginemos que estamos desarrollando una aplicación para un banco. Construimos un servicio que nos permite meter dinero en una cuenta corriente. Para ello habrá que implementar un pequeño programa que haga una actualización en varias tablas del modelo de datos. Desarrollamos otro pequeño programa que nos permite sacar dinero de la cuenta. Igualmente tendremos que implementar el acceso a la base de datos para actualizar las tablas.

Bien ¿y qué pasa si ahora nos piden hacer una funcionalidad de transferencia de dinero entre cuentas? ¿Es necesario volver a implementar un programa que accede a las tablas para hacer las actualizaciones necesarias? Pues afortunadamente, con **SOA**, la respuesta es no.

Aquí es donde entra la característica de composición de **SOA**. Podemos hacer un nuevo servicio totalmente nuevo, que usa el servicio “retirar dinero de una cuenta” y el de “meter dinero en la cuenta”. Un nuevo servicio de más alto nivel, de más valor para la empresa y para entendernos “sin programar”. Ya no tendremos que saber, por ejemplo, cómo se accede a la base de datos.

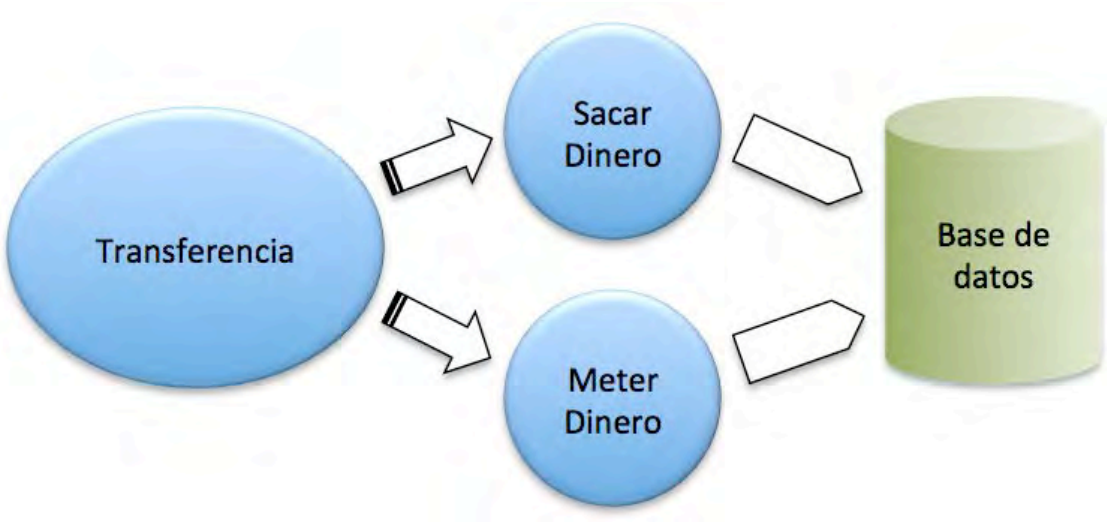


Figura 3.1. Esquema SOA.

Claro que esto es mucho prometer. Nada más y nada menos que la reutilización de los activos software de la empresa. Algo que se ha perseguido quizás desde el inicio de la programación estructurada, allá por los años 70, y que nunca se había hecho realmente realidad. Pero con SOA, tenemos una oportunidad de llevarlo a la práctica ya que tenemos los medios para ello.

3.1.4 Servicios Web

Existen numerosas definiciones de *Servicios Web* y esto demuestra, en parte, la gran complejidad de los servicios que se agrupan bajo este término y las implicaciones asociadas a ellos. Hasta ahora la definición más general y convincente es decir que los Servicios Web son el conjunto de aplicaciones o tecnologías con capacidad para interoperar en la Web. Estas tecnologías intercambian datos entre ellas con el fin de ofrecer unos servicios.

La World Wide Web no es sólo un espacio de información, también es un espacio de interacción. Utilizando la Web como plataforma, los usuarios de forma remota, pueden solicitar un servicio que algún proveedor ofrezca en la red. Pero para que esta interacción funcione, deben existir unos mecanismos de comunicación estándares entre diferentes aplicaciones. Estos mecanismos deben poder interactuar entre sí para presentar la información de forma dinámica al usuario. Se precisa, pues, una arquitectura de referencia

estándar que haga posible la interoperabilidad y extensibilidad entre las distintas aplicaciones y que permita su combinación para realizar operaciones complejas.

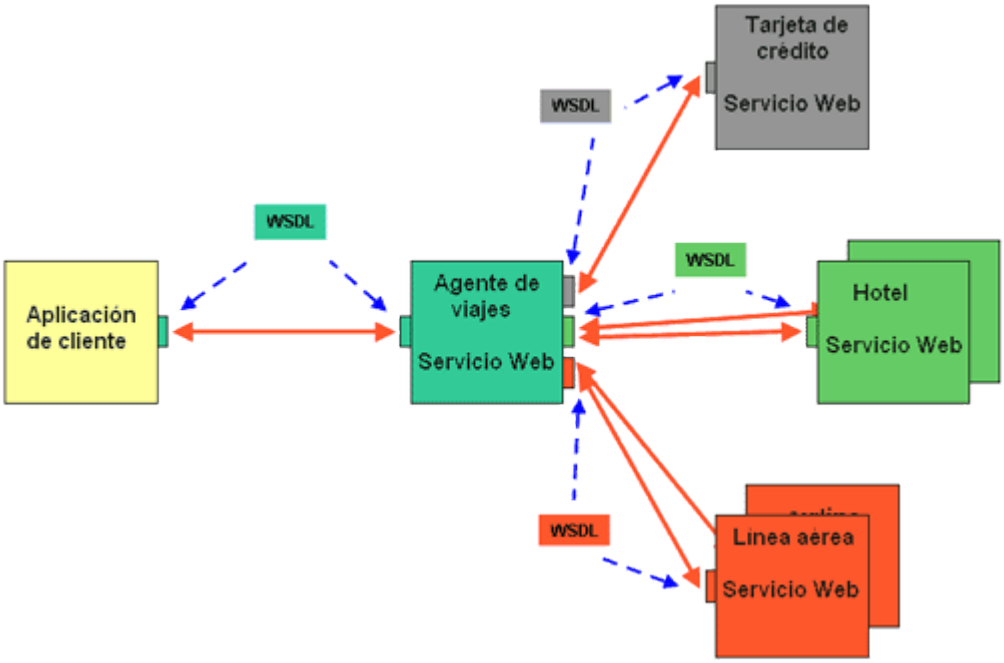


Figura 3.2. Fuente: W3C Oficina Española. "Los Servicios Web en funcionamiento". .

Web Services **(WS)** ofrece un significado estándar para interoperar entre diferentes aplicaciones de software corriendo en diferentes plataformas y/o marcos de trabajo. El W3C pretende diseñar la arquitectura, definirla y crear el núcleo de tecnologías que hagan posible los Servicios Web. Esta arquitectura se basa en los siguientes componentes:

3.1.4.1 Diseñar un marco de mensajería

Simple SOAP: Simple Object Access Protocol es un protocolo simple para intercambiar información estructurada en un ambiente descentralizado y distribuido. "Messaging Framework" define, usando tecnologías XML, un marco extensible de mensajería que contiene una construcción del mensaje que se pueda intercambiar con una variedad de protocolos subyacentes.

Web Services Addressing (WS-Addressing): Direccionamiento de Servicios Web. La dirección de los servicios Web proporciona mecanismos neutrales para transportar los servicios web y los mensajes. Define un sistema de características abstractas y una representación de XML para referirse a servicios de la Web y para facilitar la dirección final de los mensajes. Esta especificación permite a los sistemas de mensajería soportar la transmisión del mensaje a través de redes que incluyen el procesado de nodos diferentes mediante una forma de transporte neutro.

SOAP Message Transmission Optimization (MTOM) Descripción de la Optimización de la Transmisión del Mensaje. Describe una característica abstracta y una puesta en práctica concreta para optimizar el formato de la transmisión y/o de la vía de los mensajes SOAP.

3.1.4.2 Descripción de los Servicios

Los servicios Web, aunque sean independientes entre sí, pueden vincularse para realizar una tarea. Por ejemplo, Google, utiliza un Servicio Web -Google Web APIs- basado en los estándares **SOAP** y **WSDL** que permite programar en Java, Perl óVisual Studio.NET y que sirve para la recuperación de información permitiendo utilizar este buscador en distintas plataformas y Servicios Web.

Como se ha afirmado anteriormente, los servicios web se componen de varias capas entre las que se destacan: **servicios de transporte** (constituidos por los protocolos del nivel más bajo, que codifican la información independientemente de su formato, y que pueden ser comunes a otros servicios), **de mensajería, de descripción y de descubrimiento**.

En la capa inferior se encuentran los servicios de transporte que son los encargados de establecer la conexión y el puerto utilizado. Lo más común es emplear el protocolo de hipertexto HTTP, pero también se pueden usar otros protocolos como SMTP (Simple Mail Transfer Protocol o Protocolo de Transmisión de Correo Simple que es el protocolo que nos permite recibir correos electrónicos), o el protocolo FTP(File Transfer Protocol). En la capa siguiente se encuentran los servicios de mensajería que especifican cómo se tiene que codificar el mensaje que contiene los datos que se intercambian entre el ordenador cliente y el ordenador servidor. Como se ha afirmado, el protocolo más utilizado en esta capa es SOAP que permite utilizar cualquiera de los protocolos de transporte antes mencionados y que se basa en el lenguaje XML para especificar los mensajes.

Por su parte, la función del lenguaje **WSDL** **(Web Service Description Language)** es decirle a una aplicación qué formato usar para comunicarse, especificando por medio de un lenguaje estándar, tanto la dirección del servicio como la interfaz que se va a utilizar. WSDL es un lenguaje basado en XML.

WSDL se usa para describir *qué puede hacer un servicio web*, *dónde reside*, y *cómo invocarlo*. El protocolo concreto y las especificaciones del formato de datos para un tipo particular de puerto constituye un enlace reutilizable. Un puerto se define por asociación a una dirección de red con un enlace reutilizable; una colección de puertos define un servicio.

Por último, en la capa superior se encuentra **UDDI** **(Universal Description, Discovery and Integration)**, un protocolo que permite no sólo describir servicios web, sino también describir productos, compañías, transacciones, etc. UDDI es uno de los principales edificios construidos para llevar a cabo los servicios Web. Usando la interfaz de UDDI, pueden conectarse dinámicamente las empresas con los servicios proporcionados por socios externos. Para ello es necesario registrarse en UDDI y los registros pueden tener diversos propósitos y usarse en distintos contextos. Se monta sobre SOAP y asume que las consultas y las respuestas son objetos de UDDI enviados como mensajes de SOAP.

Así pues, la plataforma básica de los Servicios Web es el lenguaje XML construido sobre el protocolo de hipertexto HTTP y para el intercambio de esta información estructurada en un entorno descentralizado y distribuido, se utiliza el protocolo SOAP (Simple Object Access Protocol), pero en los Servicios Web también intervienen otros mecanismos, lenguajes y tecnologías entre las que se encuentran el lenguaje WSDL, el protocolo UDDI y otros

lenguajes como WSFL, WSML, WSMO, WSMX, etc.

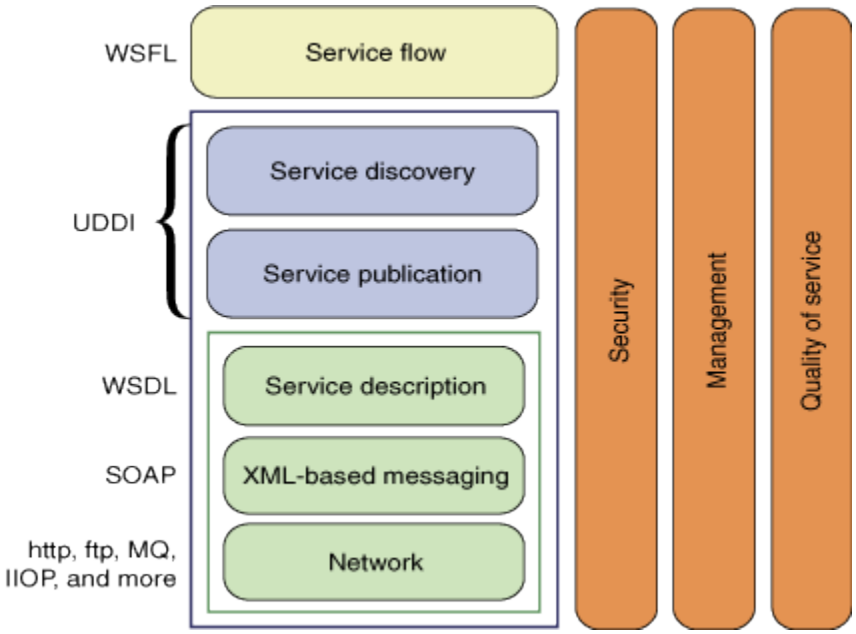


Figura 3.3 La estructura de Servicios Web en capas.

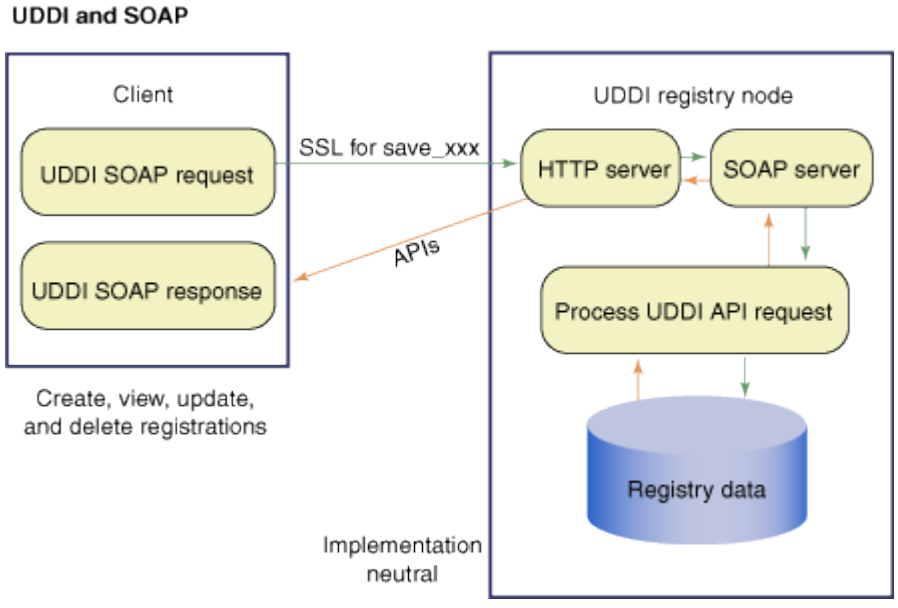


Figura 3.4. UDDI and SOAP

3.1.4.3 "WSDL: El contrato de un Servicio"

Aquí va una pequeña historia práctica.

"Pablo es un desarrollador que está realizando un portal corporativo para una empresa farmacéutica. Una de las funcionalidades del portal es incluir un buscador de internet. Como Pablo es muy listo, sabe que el mejor buscador que existe es Google, y por tanto decide navegar por la web para ver si encuentra alguna información sobre el algoritmo de búsqueda que utiliza este famoso buscador.

Después de interminables horas de búsqueda, descubre que Google ofrece su buscador en forma de Servicio Web. En ese momento Pablo piensa... ¿Por qué construir algo que ya existe, es bueno, y puedo utilizarlo aunque mi portal esté desarrollado en Java?

Por lo tanto Pablo decide utilizar el Servicio Web de Google. Pero ¿Cómo utilizarlo? ¿Qué datos debe enviar? ¿Qué datos va a recibir? ¿Cuáles son las funcionalidades que ofrece el Servicio de Google? Toda esta información que necesita Pablo debería estar en algún lugar, ya que es imprescindible para acceder al Servicio."

Desde el punto de vista de los negocios, todas las respuestas a las preguntas de Pablo definen el contrato entre el cliente (el que usa el Servicio) y el proveedor (quien lo implementa), ya que indica las pautas a seguir por cada una de las partes.

Pues bien, WSDL (Web Service Description Language) es el lenguaje estándar definido por el W3C para describir un Servicio Web y crear ese contrato. No es un documento obligatorio, pero es muy importante que sea estándar ya que así se podrá acceder de manera dinámica a los Servicios.

Es muy importante entender WSDL porque es la parte fundamental para desarrollar Servicios. No es necesario saber construir un documento WSDL (ya que lo construyen automáticamente las herramientas de desarrollo), pero sí entenderlo.

WSDL es un lenguaje basado en XML creado para definir el interfaz de los servicios.

Como conclusión, el interfaz de un Servicio es el componente fundamental ya que define lo que ofrece el Servicio. Es muy importante dedicarle todo el tiempo necesario, ya que cualquier modificación provocará cambios en el resto de nuestras aplicaciones.

Para ello en nuestro caso de estudio hemos elegido y utilizado una excelente herramienta open source, como es el framework Spring, que permite una gran capacidad de manejo de servicios e integración con otras tecnologías casi de manera transparente.

3.2 Vamos a introducir y hablar un poco de Spring

"Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application."

Traducido, Spring Framework es una plataforma que nos proporciona una infraestructura que actúa de soporte para desarrollar aplicaciones Java. Spring maneja toda la infraestructura y así te puedes centrar en tu aplicación. Diciéndolo más coloquialmente, Spring es el "pegamento" que une todos los componentes de la aplicación, maneja su ciclo de vida y la interacción entre ellos.

Spring Framework es un contenedor ligero ("lightweight container") en contraposición a un servidor de aplicaciones J2EE. En el caso de una aplicación web, te basta con un contenedor de servlets como Tomcat o Jetty. Pero Spring no solo se puede usar para crear aplicaciones web, se podría usar para cualquier aplicación java, aunque su uso habitual sea en entornos web, nada te impide utilizarlo para cualquier tipo de aplicación.

3.2.1 ¿Por qué surgió Spring Framework?

En la actualidad el desarrollo de aplicaciones empresariales es más ambicioso y por lo tanto más complejo. También se vuelve un poco más complicado porque nuestras aplicaciones deben ser capaces de conectarse con otras aplicaciones y servicios. Además, como desarrolladores, somos los responsables de coordinar cada una de las partes de que la forman para que todo funcione correctamente.

Por si no fuera suficiente, debemos tomar en cuenta que debemos darle mantenimiento a nuestras aplicaciones, y que en algunos casos será necesario que cambiemos módulos o capas enteras de la misma para mejorarla; como por ejemplo sustituir nuestras consultas JDBC con Hibernate en los casos en los que sea prudente.

Afortunadamente existen APIs, y Frameworks; los cuales aunque son opcionales, el aprender a usarlos nos ayudará a desarrollar nuestras aplicaciones en menos tiempo y a que estas sean más robustas y contengan menos errores. **Spring** es el más popular de estos super-frameworks Java. Nos proporciona varios módulos los cuales abarcan la mayor parte de las cosas que debemos hacer en cualquiera de las capas de nuestras aplicaciones, desde plantillas para trabajar con JDBC o invocación de Web Services y JMS, pasando por sus propias soluciones, ORM o MVC (web), hasta integración con otros frameworks, como Struts 2, Hibernate, JSF, etc. Todo esto de una forma elegante y haciendo uso de muchos buenos principios de programación. Además Spring maneja la infraestructura de la aplicación, por lo que nosotros solo deberemos preocuparnos de la lógica de la misma (y de la configuración de Spring).

Aunque Spring se encuentra dividido en distintos módulos, cada uno de los cuales se encarga de partes diferentes de nuestra aplicación, no deja de ser un monstruo, ya que es tan grande que alguien podría nunca usar todos estos módulos en aplicaciones pequeñas o medianas; pero en aplicaciones grandes o realmente grandes puede ahorrarnos mucho trabajo ya que puede coordinar todas las partes de la aplicación. Esta separación en módulos nos permite usar solo las partes que necesitamos, sin tener la carga de los que no usemos.

Spring está diseñado para no ser intrusivo, esto significa que no es necesario que nuestra aplicación extienda o implemente alguna clase o interface de Spring (si no lo queremos), por

lo que nuestro código de lógica quedará libre y completamente reutilizable para un proyecto sin Spring, o por si debemos quitarlo de una aplicación que ya lo esté usando. Gracias a esto es posible usar un POJO o un objeto Java para hacer cosas que antes solo podían hacerse con EJBs.

Hoy en día Spring ha crecido mucho, tiene mucha demanda. Incluso ya no es exclusivo de Java, pues ya hay versión para .NET, bautizada como Spring.NET. Si usamos Spring de la forma correcta (lo cual no es difícil) nuestra aplicación quedará dividida en capas bien delimitadas, y con buenas prácticas de programación.

El núcleo de Spring está basado en un principio o patrón de diseño llamado **Inversión de Control** (**IoC**). Las aplicaciones que usan el principio de **IoC** se basan en su configuración (que en este caso puede ser en archivos XML o con anotaciones como en **Hibernate**) para describir las dependencias entre sus componentes, esto es, los otros objetos con los que interactúa. En este caso **"inversión"** significa que **la aplicación no controla su estructura**; permite que sea el framework de IoC (en este caso Spring) quien lo haga.

Por ejemplo, supongamos que tenemos una clase "**AlmacenUsuario**", que depende de una instancia de una clase "**UsuariosDAO**" para realizar su tarea. "**AlmacenUsuario**" crea una instancia de "**UsuariosDAO**" usando el operador "**new**" u obtiene una de algún tipo de Fabrica. Usando la técnica de **IoC**, una instancia de "**UsuariosDAO**", o una subclase de esta, es proporcionada a "**AlmacenUsuario**" en tiempo de ejecución por el motor de Spring. En este caso "**UsuariosDAO**" también podría ser una interface y Spring se encargará de proporcionarnos una instancia de una clase que implemente esa interface. Esta **inyección de dependencia** en tiempo de ejecución ha hecho que a este tipo de IoC se le dé el nombre más descriptivo de **inyección de dependencia** (**DI**). El concepto importante es que los componentes no saben cuál implementación concreta de otros componentes están usando; solo ven sus interfaces.

El uso de interfaces y DI son mutuamente benéficos, ya que hace más flexible y robusta nuestra aplicación y es mucho más fácil realizar pruebas unitarias. Pero la complejidad de escribir código que maneje las dependencias entre los componentes de una aplicación diseñada para usar interfaces puede llegar a ser mucho y esto, además, hace que los desarrolladores tengamos que escribir aún más código. Afortunadamente, usando DI reducimos la cantidad de código extra que debemos escribir, para un diseño basado en

interfaces, casi a cero.

En el contexto de DI, Spring actúa como un **contenedor** que proporciona las instancias de las clases de nuestra aplicación todas las dependencias que necesita, pero en una forma no intrusiva y automática. Todo lo que debemos hacer es crear un archivo de configuración que describa las dependencias; Spring se hará cargo del resto.

Como dijimos antes: **Spring** es un contenedor ya que no solo crea los componentes de nuestra aplicación, sino porque contiene y maneja al ciclo de vida y configuración de estos componentes. En Spring, podemos declarar cómo debe ser creado cada uno de los objetos de nuestra aplicación, cómo deben ser configurados, y cómo deben asociarse con los demás.

La implementación de DI de Spring se enfoca en el acoplamiento débil: los componentes de nuestra aplicación deben asumir lo menos posible acerca de otros componentes. La forma más fácil de lograr este bajo acoplamiento en Java es mediante el uso de Interfaces. Como cada componente de la aplicación solo está consciente de la interface de otros componentes, podemos cambiar la implementación del componente sin afectar a los componentes que lo usan.

3.2.2 Beneficios del uso de Inyección de dependencia (DI)

- **Reduce el código pegamento:** Esto quiere decir que reduce dramáticamente la cantidad de código que debemos escribir para unir los distintos componentes. Aunque algunas veces esté código puede ser tan simple como usar el operador “new” para instanciar un nuevo objeto, otras puede ser más complejo, como realizar una búsqueda de dicha dependencia en un repositorio a través de JNDI, como en el caso de los recursos remotos. En este caso, el uso de **DI** puede proporcionar búsquedas automáticas.
- **Externaliza dependencias:** Como es posible colocar la configuración de dependencias en archivos XML podemos realizar una reconfiguración fácilmente, sin necesidad de recompilar nuestro código. Gracias a esto es posible realizar el cambio de la implementación de una dependencia a otra.
- **Las dependencias se manejan en un solo lugar:** Toda la información de dependencias es responsabilidad de un solo componente, el contenedor de IoC de Spring, haciendo que este manejo de dependencias más simple y menos propenso a errores.

- **Hace que las pruebas sean más fáciles:** Nuevamente, como nuestras clases serán diseñadas para que sea fácil el reemplazo de dependencias, podemos proporcionar mocks o dummies, que regresen datos de prueba, de servicios o cualquier dependencia que necesite el componente que estamos probando.

Como podemos ver, el uso de **DI** nos proporciona muchos beneficios, pero no sin sus correspondientes desventajas. En particular, es difícil ver qué implementación de una dependencia está siendo usada para qué objeto, especialmente para alguien que no está familiarizado con esta forma de trabajo. ¿Y por qué tanto hablar de DI? Pues porque estos dos conceptos (**IoC y DI**) son los puntos centrales alrededor del cual gira todo en Spring, así que es mejor entenderlos desde el principio.

3.2.3 Módulos

Spring es bastante grande, por ello el proyecto está dividido en módulos. No siempre se utiliza en un proyecto todo lo que tiene spring. Por poner un ejemplo, podrías utilizar Struts para la parte web, en vez de Spring MVC.

Spring está dividido en alrededor de **20 módulos** y colocados en los siguientes grupos:

- Contenedor Central (Core Container)
- Acceso a Datos / Integración
- WEB
- AOP (Programación Orientada a Aspectos)
- Instrumentación
- Pruebas (TEST)

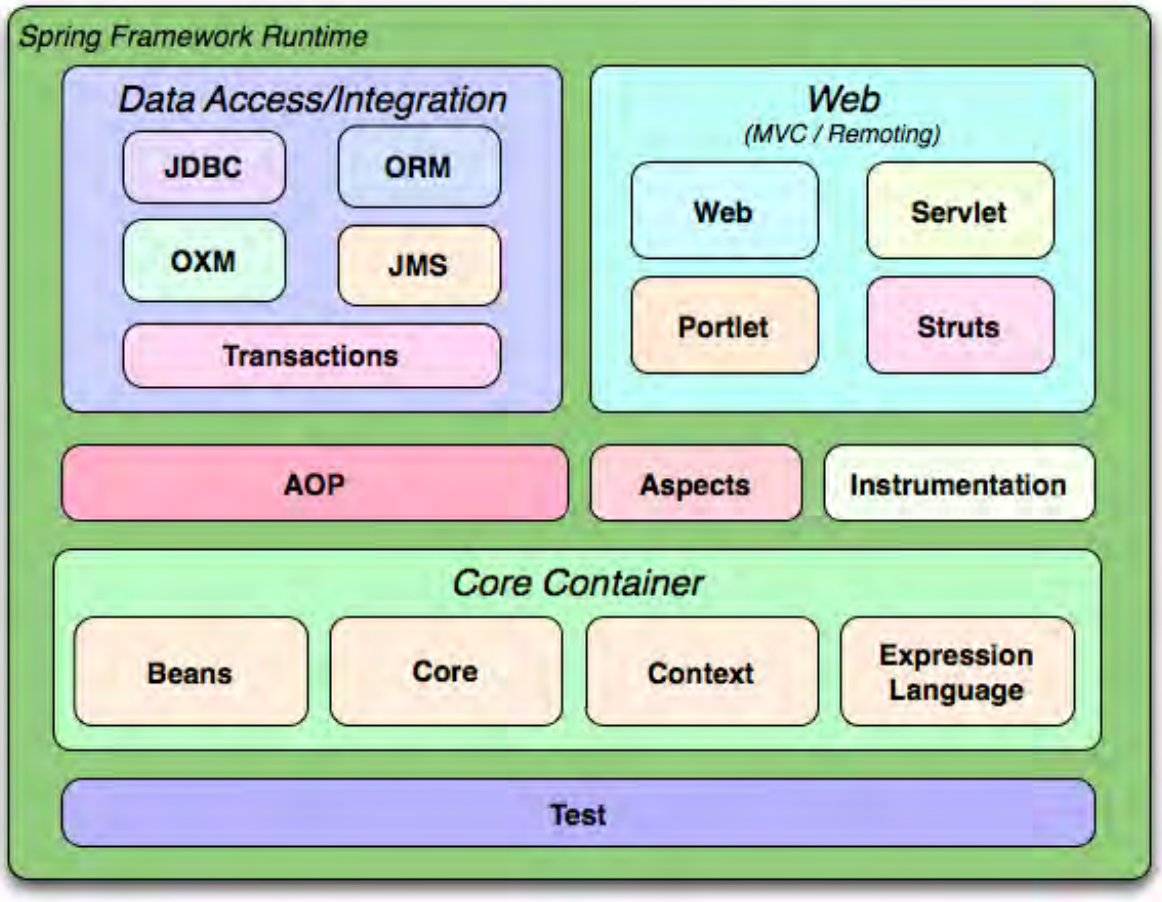


Figura 3.5. Estructura Framework Spring

En general, estas son algunas de las características de Spring:

- Simplificación de la programación orientada a aspectos.
- Simplificación del acceso a datos.
- Simplificación e integración con JEE
- Soporte para planificación de trabajos.
- Soporte para envió de mail.
- Interacción con lenguajes dinámicos (como BeanShell, JRuby, y Groovy).
- Soporte para acceso a componentes remotos.
- Manejo de Transacciones.
- Su propio framework MVC.
- Su propio Web Flow.
- Manejo simplificado de excepciones.

3.2.4 Spring - Contenedores de IoC e Inyección de Dependencias

Como vimos, el core de **Spring** se basa en el concepto de Inversión de Control (**IoC**), más específicamente en la Inyección de Dependencias (**DI**). Este es un proceso en el cual los objetos definen sus dependencias (o sea, los otros objetos con los que trabajan) solo a través de los argumentos de su constructor, argumentos a un método de factory, o métodos setter que son invocados después de que el objeto se ha construido. En este caso en vez de ser el mismo objeto quien se encargue de instanciar, o localizar, las dependencias con las que trabaja (usando directamente su constructor o un localizador de servicios), **es el contenedor el que inyecta estas dependencias cuando crea al bean**. Este proceso, como podemos observar, es el inverso al que normalmente se hace, y de ahí el nombre de Inversión de Control (es el framework el que hace el trabajo, no el programador).

Spring coloca las clases básicas de su contenedor de **IoC** en dos paquetes:

- **org.springframework.beans**
- **org.springframework.context**

Estos paquetes contienen **dos interfaces** que son las que realizan la "**magia**" de la instanciación de objetos.

La primera es "org.springframework.beans.factory.BeanFactory", que proporciona un mecanismo de configuración avanzada capaz de manejar cualquier tipo de objeto. Además proporciona una sub-interface, "org.springframework.context.ApplicationContext", la cual agrega una integración más fácil con algunas de las características más interesantes de **Spring**, como su módulo de Programación Orientada a Aspectos, manejo de recursos de mensajes (para la internacionalización), publicación de eventos, y contextos específicos para ciertas capas de aplicaciones (como aplicaciones web), entre otras.
"BeanFactory" proporciona el framework de configuración y la funcionalidad básica, mientras que **"ApplicationContext"** agrega más funcionalidad específica para ciertos productos empresariales. Ambas interfaces representan **contenedores de beans**, solo que de tipos distintos.

En Spring, los objetos que forman la columna vertebral de las aplicaciones, y que son administradas por el contenedor de IoC de Spring, son llamados **"Beans".** Un bean es un objeto que es instanciado, ensamblado (cuando sus dependencias son inyectadas), y en general, administrado por el contenedor de IoC. Dicho de otra forma: un bean es simplemente uno de muchos objetos de nuestra aplicación. Los Beans, y las dependencias entre ellos, las declaramos en los metadatos del contenedor de IoC (lo cual puede ser mediante *anotaciones* o *archivos de mapeo*).

Spring proporciona muchas implementaciones de "ApplicationContext".
En particular en las aplicaciones que NO son web,
usamos "org.springframework.context.support.ClassPathXmlApplicationContext", que busca los archivos de configuración dentro del **classpath**, o
"org.springframework.context.support.FileSystemXmlApplicationContext", que los busca en el sistema de archivos de la máquina donde se ejecute la aplicación.

Para ejemplificar todo esto, haremos una aplicación la cual tendrá una clase de prueba que invocará a un "**ServicioRemoto**". Este servicio no hará algo especial, solamente regresará un número aleatorio entre **0** y **10**; lo interesante de esto es que el cliente obtendrá una instancia del ServicioRemoto sin tener que crearla de manera directa, sino usando la DI de Spring.

Crearemos nuestra clase "**ServicioRemoto**", dentro del paquete "**beans**". Esta clase será muy sencilla, y solo tendrá un método llamado "consultaDato" el cual no recibirá ningún parámetro y regresará un "int" aleatorio entre 1 y 10. La clase "ServicioRemoto" queda de esta forma:

```
public class ServicioRemoto
{    public int consultaDato()
   {
        return (int)(Math.random()*10.0);
   }
}
```

Como se ve es una clase sumamente sencilla, no hará falta explicar mucho.

Normalmente para obtener una nueva instancia de "ServicioRemoto" tendríamos que colocar la siguiente línea en alguna parte de la clase de prueba:

```
servicioRemoto = new ServicioRemoto();
```

Como habíamos dicho antes: el cliente no debería ser responsable de la instanciación de este “ServicioRemoto”, ya que es una tarea que no le corresponde. Gracias a la **DI** podremos evitarnos el escribir la línea anterior, y por lo tanto muchos problemas potenciales que pueden surgir con ella.

El acto de crear estas asociaciones entre objetos de la aplicación, usando **DI**, es llamado "**wiring**" (**cableado** en español). A continuación veremos la configuración básica de este wiring con Spring, y veremos las posibilidades que este nos ofrece.

3.2.5 Configuración de Beans en Spring

Existen, básicamente, dos formas de configurar los beans y sus dependencias en **Spring**, la primera es mediante un archivo de configuración en XML, la segunda es mediante anotaciones proporcionadas por Spring. Como siempre, cada una tiene sus ventajas y sus desventajas. Aquí veremos la configuración en XML que es la más utilizada.

3.2.5.1 Configuración con archivos XML

La configuración de Spring consiste de, al menos, la definición de un bean que será administrado por el contenedor.

Estas definiciones de beans se corresponden con los objetos que componen nuestra aplicación. Cuando creamos una aplicación, típicamente definimos una **capa de servicios**, una **capa de acceso a datos** (o **DAOs**) una **capa de presentación**, y una **capa de infraestructura** (con objetos como el "SessionFactory" de Hibernate). En Spring normalmente declaramos objetos que definen estas 4 capas (y no objetos muy específicos del dominio), ya que son las capas anteriores quienes se encargan de crear y manejar los objetos del dominio.

Crearemos el archivo XML de configuración en nuestro proyecto. Le damos un nombre al archivo, en nuestro caso será "applicationContext", con eso aparecerá en nuestro editor el archivo "**applicationContext.xml**" con el siguiente contenido:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">  </beans>
```

El archivo contiene únicamente el elemento raíz "**<beans>**" dentro del cual declararemos cada uno de nuestros beans usando un elemento **"<bean>"** para cada uno.

Lo siguiente que haremos es declarar nuestro bean **"ServicioRemoto"** para que el contenedor de **Spring** lo administre. Para esto declaramos un elemento "<bean>" en nuestro archivo de configuración:

El elemento "<bean>" es la unidad de configuración más básica en Spring, le dice que cree y administre un objeto.

Debemos darle un identificador a cada uno de nuestros beans dentro de la aplicación; esto no es obligatorio pero si no lo hacemos no podremos distinguir a los beans con los que trabajamos. Estos deben estar identificados de manera única dentro del contenedor que tiene al bean. Un bean por lo regular solo tiene un identificador. En esta configuración basada en XML usamos el atributo "id" o el atributo "name" para especificar el identificador del bean.

Para este bean, usaremos el identificador "servicioRemoto**":**

```
<bean id="servicioRemoto" />
```

Ahora nuestro bean puede ser referenciado por el nombre de "servicioRemoto".

El último paso que debemos hacer en nuestro archivo de configuración es decirle a Spring de qué tipo es el bean, esto es, la clase de la que queremos que cree los objetos. Para hacer esto usamos el atributo "**class**". En este caso el bean será de tipo

"**ejemplos.spring.ioc.beans.ServicioRemoto**". Por lo que la configuración de nuestro bean queda de la siguiente forma:

```
<bean id="servicioRemoto" class="ejemplos.spring.ioc.beans.ServicioRemoto" />
```

Y el archivo de configuración queda así:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="servicioRemoto" class="ejemplos.spring.ioc.beans.ServicioRemoto" />
</beans>
```

3.2.6 Inyección de Dependencias

Una aplicación empresarial típica no consiste de un solo objeto o bean (como en el caso de nuestra prueba con "ServicioRemoto"). Aún la aplicación más simple tiene unos cuantos objetos que trabajan juntos para realizar sus procesos de una manera coherente. Como ya dijimos antes, **Spring** nos proporciona la forma de unir estos objetos sin tener que indicarlo de manera explícita en el código. Ahora veremos cómo hacer esto de una manera rápida y sencilla.

Cuando usamos el mecanismo de DI de Spring, el código es más limpio, y está desacoplado de una forma más efectiva. El objeto no busca sus dependencias, de hecho ni siquiera conoce la ubicación o clase de su dependencia.

En Spring, la DI existe en dos formas principales:

- Inyección de dependencia basada en constructor
- Inyección de dependencia basada en setter

Ahora explicaremos la primera forma, ya que es la utilizada en nuestra práctica. Para esto, modificaremos un poco la clase "ServicioRemoto" para que haga uso de un par de dependencias y por lo tanto la cosa se vuelva más interesante.

Lo primero que haremos es crear una nueva interface llamada "Proceso". Esta interface representará el proceso que se realizará dentro del servicio remoto. Tendrá un método llamado "ejecuta ()", que será el que se encargará de realizar el proceso. Colocamos esta interface en el paquete de nuestros beans. "Proceso" queda de la siguiente forma:

```
public interface Proceso

{    Object ejecuta();    }
```

Como vemos, "ejecuta" regresa un "Object". Este irá cambiando de acuerdo a las implementaciones que realicemos de esta interface, de las cuales por cierto haremos 3. La primer implementación será la clase "Calculo". La cual solo regresará un número al azar entre 0 y 100 (que es lo que ya actualmente está haciendo la clase "ServicioRemoto"), quedando de la siguiente forma:

```
public class Calculo implements Proceso

{    public Object ejecuta()

    {        return (int)(Math.random()*100.0);    }   }
```

La siguiente implementación de "Proceso" se llamará "Concatenacion" y simplemente concatenará las cadenas "Hola " y "mundo" y nos regresará esta cadena. La clase queda de la siguiente forma:

```
public class Concatenacion implements Proceso

{    public Object ejecuta()

    {        return new StringBuilder().append("Hola ").append(" mundo");    }  }
```

La tercer y última implementación de "Proceso" se llamará "Ordenamiento" y lo que hará es regresar una lista de enteros de forma ordenada. La clase "Ordenamiento" queda de la siguiente forma:

```
public class Ordenamiento implements Proceso

{    public Object ejecuta()

   {   List< Integer> listaEnteros = new ArrayList<Integer>();

       listaEnteros.add(9);

       listaEnteros.add(3);

       listaEnteros.add(1);

       listaEnteros.add(6);

       listaEnteros.add(5);

       listaEnteros.add(10);

       Collections.sort(listaEnteros);

       return listaEnteros;    }}
```

Ahora que tenemos nuestros tres "Procesos" modifiquemos un poco nuestra clase "ServicioRemoto" para que en vez de ser él quien realice la funcionalidad, delegue este trabajo a una de las implementaciones de "Proceso". "ServicioRemoto" podrá recibir el Proceso que ejecutará de dos formas, la primera es mediante su constructor, y la segunda es pasando este Proceso mediante un setter.
Como ahora "ServicioRemoto" delegará la ejecución de los Procesos a otras clases, su única función será invocar el método "ejecuta" del Proceso correspondiente, y como estos regresan Objects, cambiará su tipo de retorno.

Con las modificaciones hechas, la clase "ServicioRemoto" queda de la siguiente forma:

```
public class ServicioRemoto
{    private Proceso proceso;
     public ServicioRemoto()    {    }
    public ServicioRemoto(Proceso proceso)     {        this.proceso = proceso;    }
    public Object consultaDato()
    {        return proceso.ejecuta();    }}
```

Ahora sí, veremos cómo hacer que Spring inyecte estas dependencias de forma automática, haciendo uso de DI.

3.2.6.1 Inyección de Dependencia basada en Constructor

En este tipo de inyección de dependencia, **el contenedor invoca a un constructor** de la clase, que recibe cierto número de argumentos, cada uno de los cuales representa una dependencia.
Spring invocará al constructor correspondiente dependiendo del número y tipo de dependencias que pasemos, ya sea a través de los archivos de configuración en XML o haciendo uso de anotaciones.

3.2.6.1.1 Inyección de Dependencia basada en Constructor con Archivos XML

Lo primero que haremos es indicar, en el archivo de configuración "applicationContext.xml", que usaremos un bean llamado "**proceso**", que en este caso será de tipo "**Ordenamiento**":

```
<bean id="proceso" class="ejemplos.spring.ioc.beans.Ordenamiento" />
```

Ahora modificaremos la declaración actual de nuestro bean "servicioRemoto" que actualmente está de esta forma:

```
<bean id="servicioRemoto" class="ejemplos.spring.ioc.beans.ServicioRemoto" />
```

Para indicar que haremos inyección de dependencia basada en constructor, usamos el elemento "<constructor-arg>". Si no indicamos este elemento se usa el constructor por default de la clase (como en los ejemplos anteriores), pero si se usa, **Spring** buscará un constructor que reciba el tipo de objeto indicado en él.

```
<constructor-arg ref="proceso" />
```

En "ref" indicamos el identificador del bean al que haremos referencia, en este caso es el bean "proceso" que creamos antes. La declaración del bean "servicioRemoto" queda de la siguiente forma:

```
<bean id="servicioRemoto" class="ejemplos.spring.ioc.beans.ServicioRemoto">
   <constructor-arg ref="proceso" />
</bean>
```

Si nuestro constructor tuviera más de un argumento, bastaría con agregar otro elemento "<constructor-arg>".
Ahora ejecutaremos nuestra aplicación de forma normal. Con el siguiente contenido en el método "main":

```
public static void main(String[] args)
{
   ApplicationContext   applicationContext =  new
ClassPathXmlApplicationContext("applicationContext.xml");
    ServicioRemoto   servicio =  applicationContext.getBean("servicioRemoto",
ServicioRemoto.class);
    System.out.println("El valor es " + servicio.consultaDato()); }
```

Que es el mismo código que ya teníamos. Al correr nuestro código obtendremos la siguiente salida:

```
run:
El valor es [1, 3, 5, 6, 9, 10]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Como vemos, hemos obtenido la salida esperada, sin tener que escribir ninguna línea de código para resolver las dependencias que usa la clase "ServicioRemoto", que en este caso es el proceso. Es más, "ServicioRemoto" ni siquiera sabe de qué tipo es el "Proceso" que está ejecutando, ni cómo encontrarlo, es el motor de **IoC** de **Spring**, que haciendo uso de **DI** se encarga de crear una instancia de la clase apropiada e inyectarla, en este caso mediante el constructor, a "ServicioRemoto".

Ahora veremos otro de los encantos de **Spring**. Sin tocar una sola línea de nuestro código (o más bien dicho, sin mover ninguna clase) haremos que "ServicioRemoto" obtenga una dependencia distinta. Para esto cambiaremos la línea:

```
<bean id="proceso" class="ejemplos.spring.ioc.beans.Ordenamiento" />
```

Por

```
<bean id="proceso" class="ejemplos.spring.ioc.beans.Concatenacion" />
```

Si volvemos a ejecutar nuestra aplicación ahora obtendremos la siguiente salida:

```
run:
El valor es Hola  mundo
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ahora se ha inyectado una instancia de "**Concatenacion**" y "**ServicioRemoto**" la usa para ejecutar su **Proceso**. ¿Por qué es útil esto? Pues imagínense el caso que tengamos nuestra aplicación ya compilada y en producción. Si quisiéramos hacer un cambio sería solo cuestión de modificar el archivo XML de configuración (que siempre está en texto plano) y reiniciar la aplicación para que haya un cambio. Esta es una de las ventajas que tiene la configuración en XML sobre las anotaciones.

Solo hemos visto un par de ejemplos muy simples, para entender el mecanismo, pero recuerden que usando esta misma lógica podemos extender estos ejemplos para armar aplicaciones grandes. También es importante recordar que es posible combinar ambas formas de configuración, anotando unos beans, y declarando otros en el archivo de configuración XML.

3.3 Implementando la arquitectura orientada a servicios con Spring

Como hemos dicho antes, la arquitectura orientada a servicios implementada con el framework **Spring** ha sido de inspiración para el desarrollo y aplicación en nuestra tesis.
El motivo fundamental fue el conocimiento práctico sobre nuestros propios trabajos. En el diseño y desarrollo de los trabajos orientados a servicios, tenemos capas bien definidas, donde cada una se encarga del manejo puntual e independiente de un conjunto de tareas en común que ayudan a resolver partes de un sistema con una tecnología en particular. Desglosando así el trabajo, permitiendo que cada capa sea independiente, sin conocimiento del desarrollo de las otras pero pudiendo comunicarse entre sí para resolver eficientemente los requerimientos de un producto particular.

En nuestra práctica cuando creamos una aplicación, típicamente definimos una capa de servicios, quienes procesan y resuelven peticiones externas, una capa de objetos del modelo de negocio, una capa de acceso a datos (o DAOs), una capa de infraestructura (con objetos como el "SessionFactory" de Hibernate) y una capa de presentación.
En el desarrollo de la tesis nos centraremos principalmente en las 4 primeras capas nombradas y dejaremos de lado la capa de presentación.

Para la capa de DAOs, acceso a datos, usaremos el framework **Hibernate,** quien desarrolla la tecnología de mapeo de los objetos al repositorio de datos en una base de datos relacional; con quien **Spring** es amigable y nos permite fácil manejo e incorporación del mismo.
Más adelante en la práctica explicaremos los aspectos básicos del mapeo de objetos y el lenguaje particular para definir funciones de recuperación de datos.

Como imaginarán son muchos objetos que albergan gran funcionalidad para las diferentes capas del sistema. Cada servicio tiene un conjunto de métodos común y otro conjunto de métodos que se determina por sus propiedades, lo mismo ocurre con los Daos que son intuitivos los métodos que se pueden definir teniendo la descripción de los objetos persistentes para la administración de sus datos, así como los filtros de búsqueda y también de los archivos de mapeo y de configuración.

Estudiando la tecnología Model Driven Architecture (**MDA**), mediante la cual el diseño de los sistemas se orienta a modelos. El desarrollo orientado a modelos permite una alta flexibilidad en la implementación, integración, mantenimiento, prueba y simulación de los sistemas.

Lo primero que se nos ocurrió fue ver si podíamos desarrollar una herramienta de modelado de la tecnología de Spring para agilizar el diseño de los sistemas, pudiendo desde el modelo de objetos **UML**, obtener una derivación a código automática, con el esqueleto definido de las clases y archivos xml que integran el desarrollo del sistema. Obteniendo así, una herramienta que sea funcional para los desarrolladores **JAVA** que utilizan la arquitectura **web Service** de **Spring**, agilizándoles el trabajo de implementación y codificación. Por otro lado, ayudar al desarrollo práctico del paradigma **MDD**, acentuando la capacidad que brindan las transformaciones de modelos y dejando constancia de su practicidad.

A continuación introduciremos a los conceptos de MDD para comprender la tecnología y luego mostraremos el trabajo práctico realizado y los resultados y conclusiones finales obtenidas.

ESTADO DE LA CUESTIÓN

3.4 Desarrollo de software dirigido por modelos

La aplicación de modelos al desarrollo de software es una propuesta que, aunque tiene ya una larga tradición en Ingeniería del Software, ha experimentado un aumento espectacular de popularidad desde la creación del lenguaje unificado de modelado (UML – *Unified Modeling Language*). Sin embargo, éste fue utilizado inicialmente como herramienta de documentación, al permitir expresar únicamente de forma intencional, no formal, la relación entre el modelo y la implementación. La principal desventaja que se deriva de este hecho se debe a que los sistemas de software no son estáticos, sino que son propensos a cambios, particularmente intensos durante las primeras fases del ciclo de vida. Como la documentación necesita estar meticulosamente sincronizada con la implementación para evitar inconsistencias, el mantenimiento de dichos modelos resulta en una serie de tareas complicadas que termina aumentando costos y esfuerzos, disminuyendo claramente la utilidad real de este tipo de propuestas para los desarrolladores de software.

Como contrapartida al carácter intencional de lenguajes tipo UML, el *desarrollo de software dirigido por modelos* (MDSD – *Model Driven Software Development*) conocido también de manera menos precisa como *desarrollo dirigido por modelos* (MDD – *Model Driven Development*) introduce un enfoque diferente: los modelos no constituyen únicamente la documentación, sino que adquieren un estatus análogo al del código, de tal manera que su implementación puede ser automatizada.

MDSD busca encontrar una abstracción del dominio específico y hacerlo accesible a través de modelado formal, lo cual proporciona el potencial necesario para automatizar la producción de software, de tal manera que se incremente la productividad. Así mismo, MDSD hace posible también que los modelos puedan ser entendidos por expertos del dominio y a su vez servir como medio de comunicación entre los participantes del proyecto de desarrollo de software, en el tiempo y en el espacio. Estas ventajas son esenciales en el contexto de los proyectos de software, debido a las características dinámicas de los mismos.

3.4.1 Introduciendo a Model Driven Architecture (MDA)

Uno de los principios básicos de la ingeniería de software es la abstracción, para separar lo esencial de lo no esencial.
En términos de negocio, lo esencial es la funcionalidad, y lo no esencial la plataforma tecnológica. Estas abstracciones nos las proveen los modelos. El modelado y la transformación de modelos, hasta el nivel de abstracción requerido, constituye el núcleo del desarrollo dirigido por modelos.

Los modelos se utilizan para capturar los requisitos, y automatizar total o parcialmente el desarrollo. Con los modelos podemos centrarnos en el diseño lógico de la aplicación, y liberarnos de los detalles de la implementación. El esfuerzo invertido en el modelado tiene una continuidad durante el desarrollo, estos modelos no son meramente parte de la documentación sino que dirigen de forma automatizada el desarrollo de código.

Model Driven Architecture (**MDA**) es un framework para el desarrollo de software definido por el Object Management Group (**OMG**).
MDA es una vía para organizar y gestionar las arquitecturas empresariales soportadas por herramientas automatizadas para definir los modelos y facilitar las transformaciones entre ellos.
Con MDA el proceso de desarrollo está dirigido por la actividad de modelar el software. Utiliza estándares abiertos de modelado, como Unified Modeling Language (**UML**).
Modelos independientes de la plataforma tecnológica pueden ser transformados en plataformas empresariales, libres o propietarias, incluyendo **J2EE**, **.Net**, **PHP…**

La idea clave que subyace a MDA es que si el desarrollo está guiado por los modelos del software, se obtendrán beneficios importantes en aspectos fundamentales como son la **productividad**, la **portabilidad**, la **interoperabilidad** y el **mantenimiento**.

Para conseguir estos beneficios, MDA plantea el siguiente proceso de desarrollo: de los requisitos se obtiene un **modelo independiente de la plataforma (PIM)**, luego este modelo es transformado con la ayuda de herramientas en uno o más modelos específicos de la plataforma **(PSM),** finalmente cada PSM es transformado en **código**. Por tanto, MDA incorpora la idea de transformaciones entre modelos (PIM a PSM, PSM a código), por lo que se necesitarán herramientas para automatizar esta tarea. Estas herramientas de transformación son, de hecho, uno de los elementos básicos de MDA.

MDA pretende separar, por un lado, la especificación de las operaciones y datos de un sistema, y por otro lado, los detalles de la plataforma en la que el sistema será construido. Para ello, MDA proporciona las bases para:

- Definir un sistema independiente de la plataforma sobre la que se construye.
- Definir plataformas sobre las que construir los sistemas.
- Elegir una plataforma particular para el sistema y
- Trasformar la especificación inicial del sistema a la plataforma elegida.

3.4.2 Modelos en MDA

Un **modelo** es una descripción de todo o parte de un sistema escrito en un lenguaje bien definido. El hecho de que un modelo esté escrito en un lenguaje bien definido tiene una gran importancia para MDA, ya que supone que el modelo tiene asociada una sintaxis y una semántica bien definida. Esto permite la interpretación automática por parte de transformadores o compiladores de modelos, fundamentales en MDA.

3.4.2.1 Unified Modeling Language: UML

UML (Unified Modeling Language) es una especificación del OMG que tiene como propósito fundamental facilitar para escribir planos de software. UML puede utilizarse para visualizar, especificar, construir y documentar los artefactos de un sistema que involucra una gran cantidad de software.
Además, esta especificación ha sido aceptada por ISO (International Organization for Standardization).
UML es apropiado para modelar desde sistemas de información en empresas hasta aplicaciones distribuidas basadas en la Web, e incluso para sistemas empotrados de tiempo real muy exigentes. Es un lenguaje muy expresivo, que cubre todas las vistas necesarias para desarrollar y luego desplegar tales sistemas.
UML es sólo un lenguaje y por tanto es tan sólo una parte de un método de desarrollo de software.

3.4.2.1.2 Diagramas en UML

Un diagrama es la representación gráfica de un conjunto de elementos.
Los diagramas se dibujan para visualizar un sistema desde diferentes perspectivas, de forma que un diagrama es una proyección de un sistema.
En teoría, un diagrama puede contener cualquier combinación de elementos y relaciones. En la práctica sólo surge un pequeño número de combinaciones, las cuales son consistentes con las cinco vistas más útiles que comprenden la arquitectura de un sistema con gran cantidad de software.

1. Diagrama de clases: Muestra un conjunto de clases, interfaces y colaboraciones, así como sus relaciones. Estos diagramas son los más comunes en el modelado de sistemas orientados a objetos y cubren la vista de diseño estática de un sistema.

2. Diagrama de objetos: Muestra un conjunto de objetos y sus relaciones. Los diagramas de objetos representan fotografías de instancias de los elementos encontrados en los diagramas de clases.
Estos diagramas cubren la vista de diseño estática o la vista de procesos estática de un sistema como lo hacen los diagramas de clases, pero desde la perspectiva de casos reales o prototipos.

3. Diagrama de casos de uso

Muestra el conjunto de casos de uso y actores (un tipo especial de clases) y sus relaciones. Los diagramas de casos de uso cubren la vista de casos de uso estática de un sistema. Estos diagramas son especialmente importantes en el modelado y organización del comportamiento de un sistema.

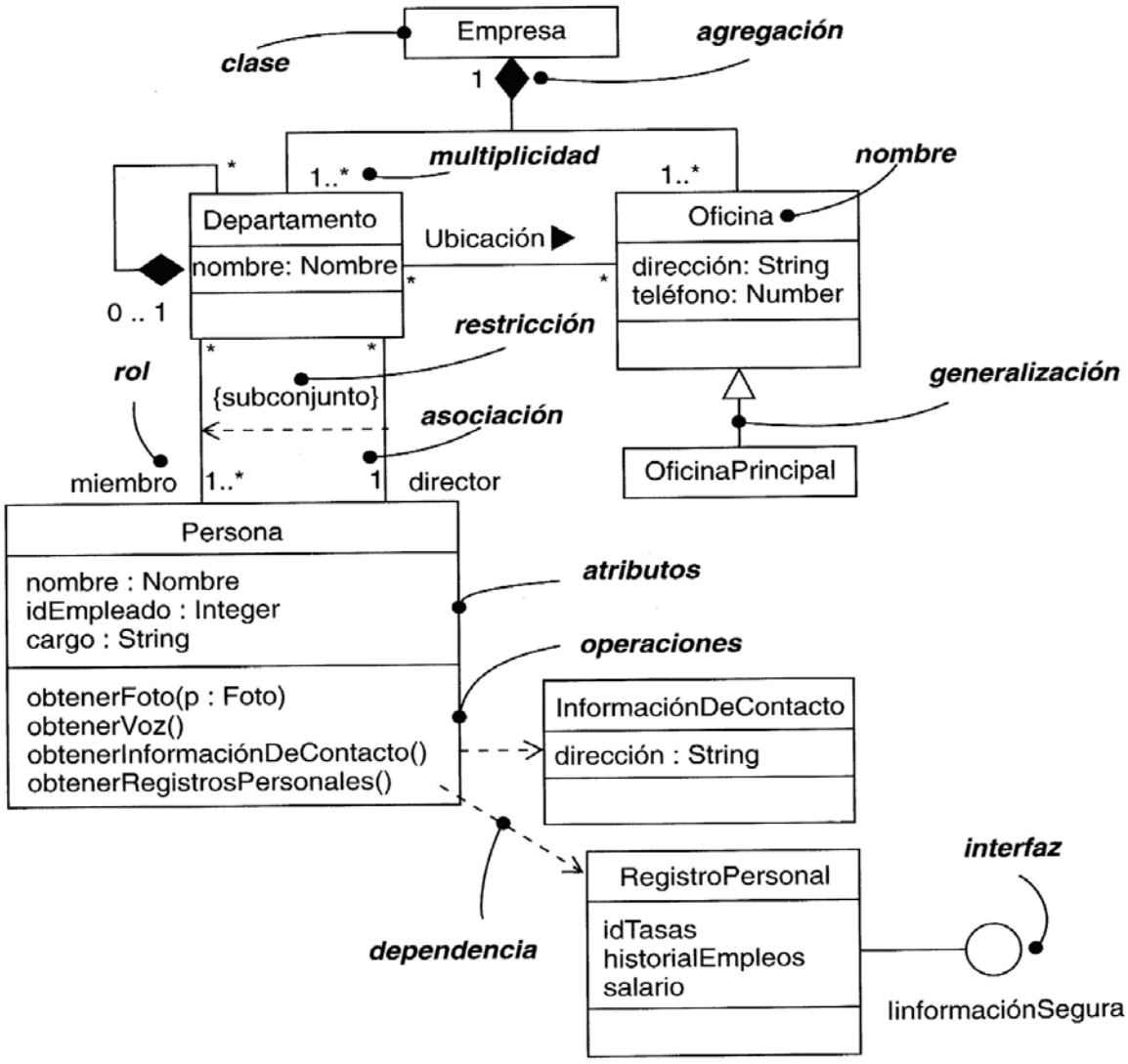


Figura 3.6.Diagrama de Clases UML.

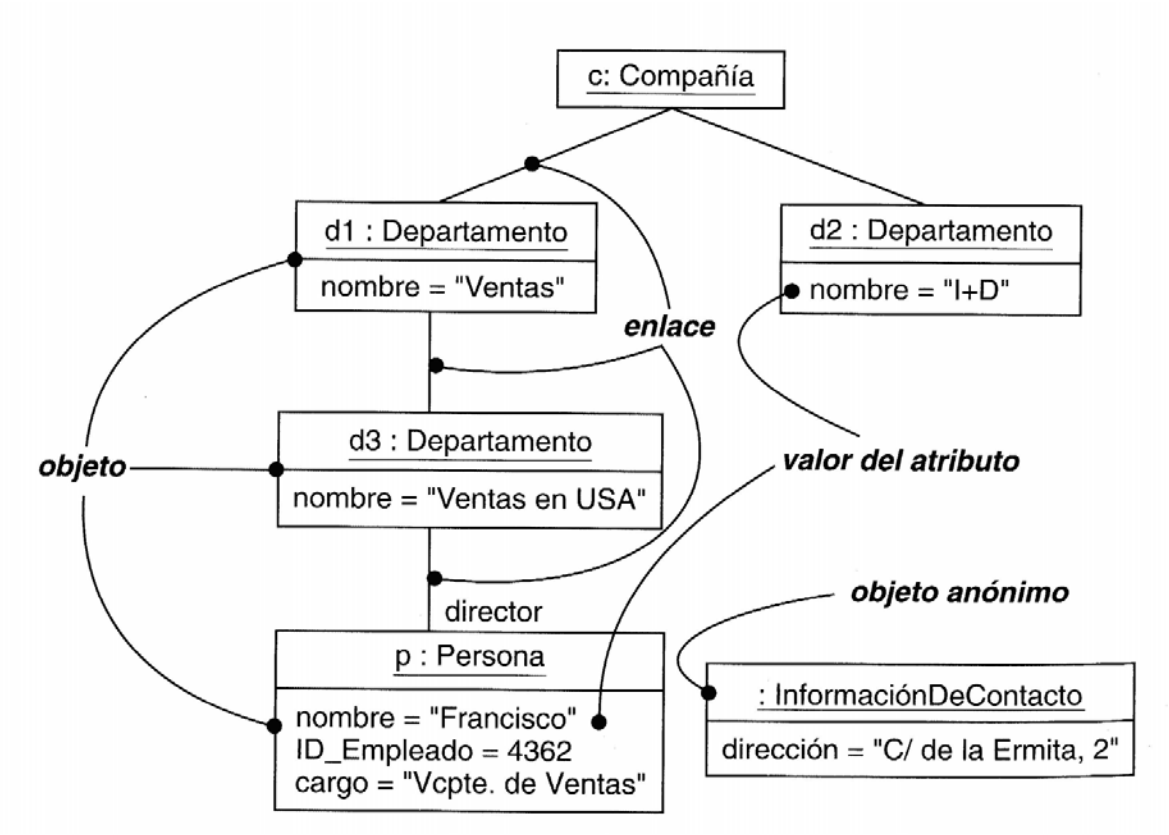


Figura 3.7.Diagrama de Objetos UML.

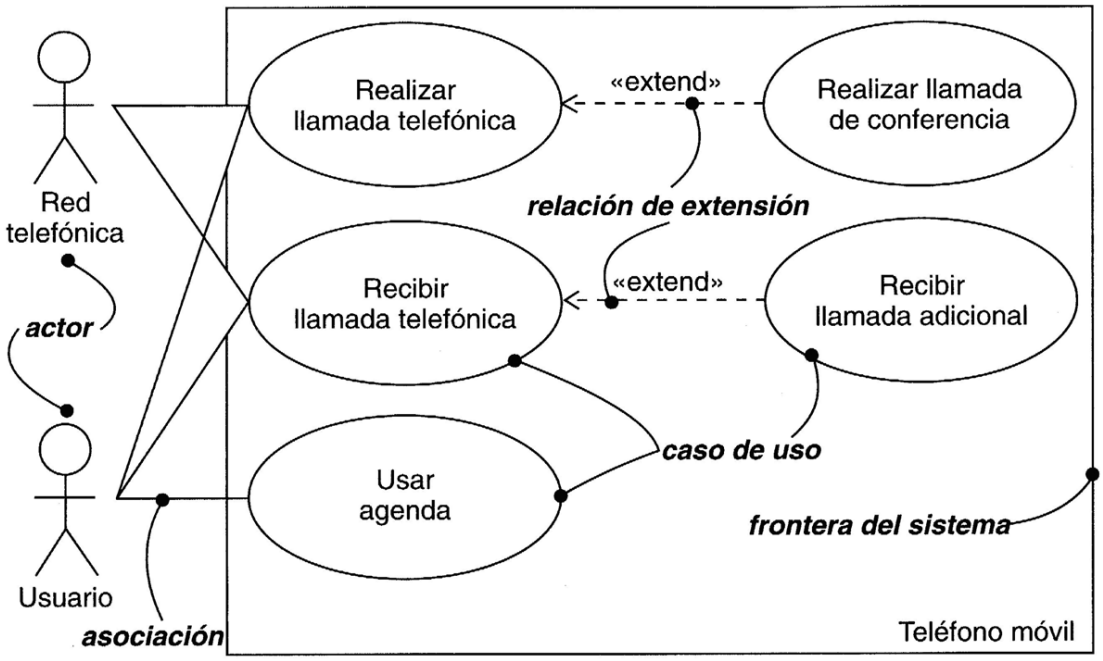


Figura 3.8.Diagrama de Casos de Uso UML.

4. Diagrama de interacción

Muestra una interacción, que consta de un conjunto de objetos y sus relaciones, incluyendo los mensajes que pueden ser enviados entre ellos. Estos diagramas cubren la vista dinámica de un sistema. Los diagramas de secuencia y de colaboración son de este tipo. Además estos diagramas son isomorfos, es decir, que se puede tomar uno y transformarlo en el otro.

• Diagrama de secuencia: es un diagrama de interacción que resalta la ordenación temporal de los mensajes.

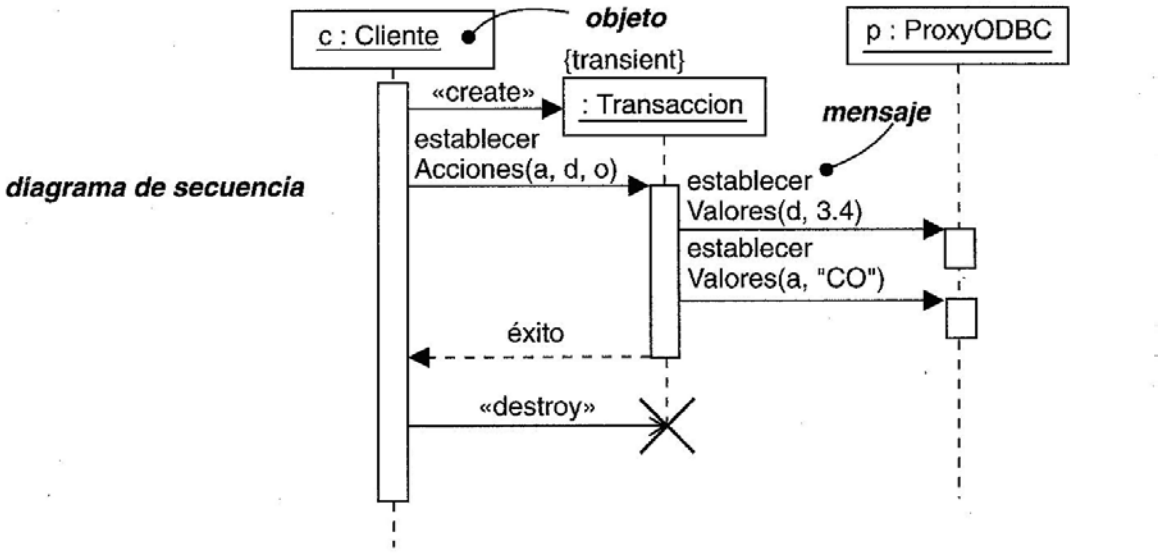


Figura 3.9.Diagrama de Secuencia.

• Diagrama de colaboración: es un diagrama de interacción que resalta la organización estructural de los objetos que envían y reciben mensajes.

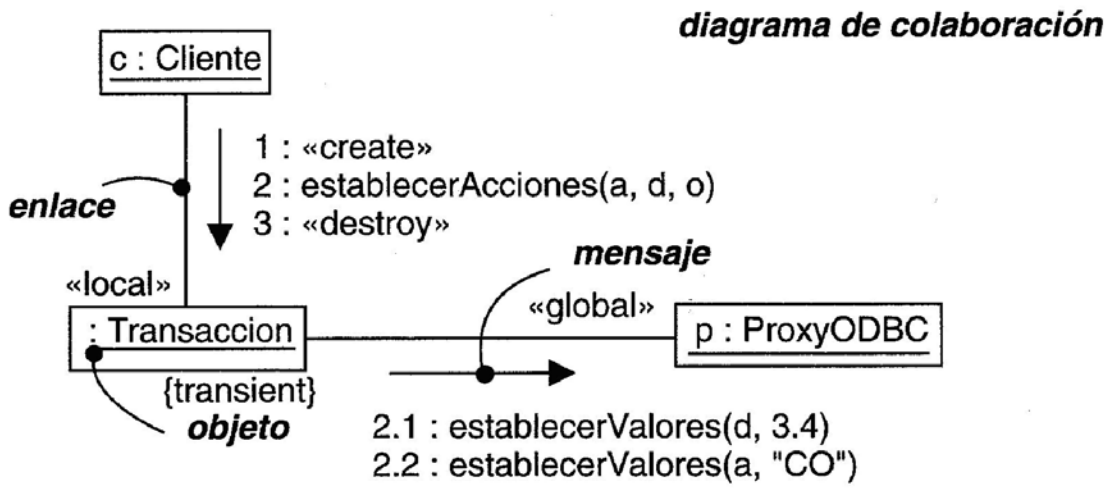


Figura 3.10.Diagrama de Colaboración.

5. Diagrama de estados

Muestra una máquina de estados, que consta de estados, transiciones, eventos y actividades. Los diagramas de estados cubren la vista dinámica de un sistema. Son especialmente importantes en el modelado del comportamiento de una interfaz, una clase o una colaboración y resaltan el comportamiento dirigido por eventos de un objeto.

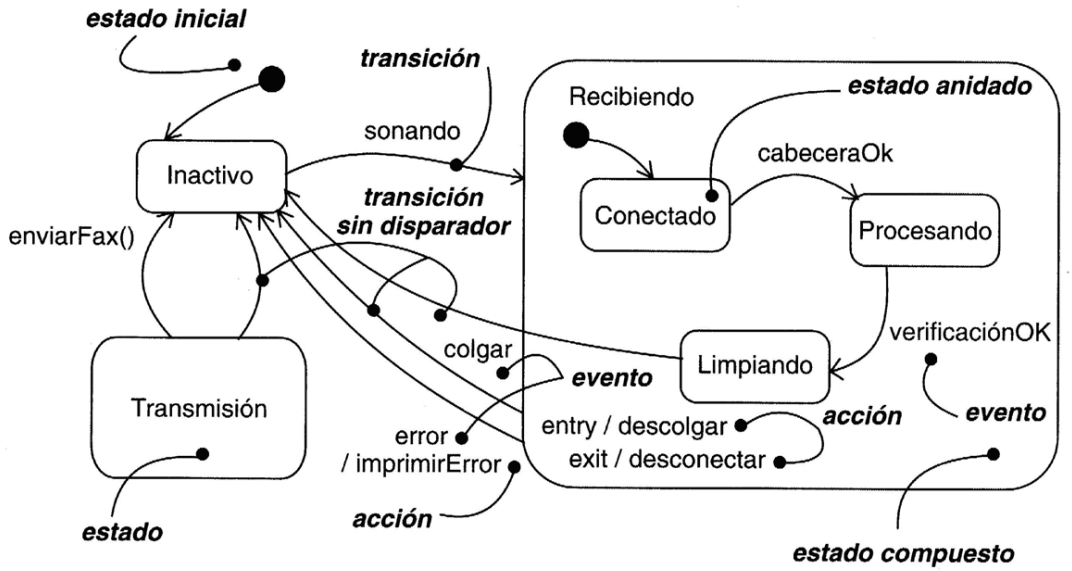


Figura 3.11.Diagrama de Estados.

3.4.2.2 Mecanismos de extensibilidad en UML

UML proporciona un lenguaje estándar para escribir planos de software, pero no es posible que un lenguaje cerrado sea siempre suficiente para expresar todos los matices posibles, de todos los modelos, en todos los dominios y en todos los momentos. Por esto, UML es abierto-cerrado, siendo posible extender el lenguaje de manera controlada.
Los mecanismos de extensión son:

- Estereotipos
- Valores etiquetados
- Restricciones

3.4.2.2.1 Técnicas de definir lenguajes de modelado

Debido a esta gran variedad de propuestas de transformación y a la ausencia de un estándar que guíe todo el proceso, han surgido multitud de arquitecturas y técnicas para definir lenguajes de modelado.
En MDA y hasta discusiones recientes, siempre se ha supuesto la existencia de tres niveles de modelado, pero siempre ha quedado abierta la cuestión de qué usar en cada nivel. Diferentes autores abogan por usar el meta-modelado para crear nuevos modelos que se ajusten más a la realidad de los problemas que se intenten reflejar, para otros bastaría con usar perfiles UML, otros abogan por el uso de perfiles con novedosos sistemas de marcas y así sucesivamente.
Todas estas técnicas requieren de un alto conocimiento de UML pues están ligadas a él de un modo u otro. UML es un lenguaje de modelado estándar de nivel M2, definido bajo MOF y es el de uso más extendido. Su estructura y la de MOF son muy parecidas con lo que resulta de gran ayuda conocer previamente UML antes de estudiar MOF.
Por otro lado, UML es el meta-modelo por excelencia para el diseño preliminar de los sistemas, es decir, para el diseño de los modelos PIMs, pues carece de estar orientado a ninguna plataforma o tecnología particular.

3.4.2.2.2 Estereotipos y Valores Etiquetados

Los estereotipos pueden extender cualquier tipo de elementos del metamodelo UML. Vienen definidos por un nombre y por el conjunto de elementos del meta-modelado sobre el que pueden asociarse. Gráficamente se definen dentro de clases haciendo uso de la sintaxis <<stereotype>> y pueden llevar asociados valores etiquetados.

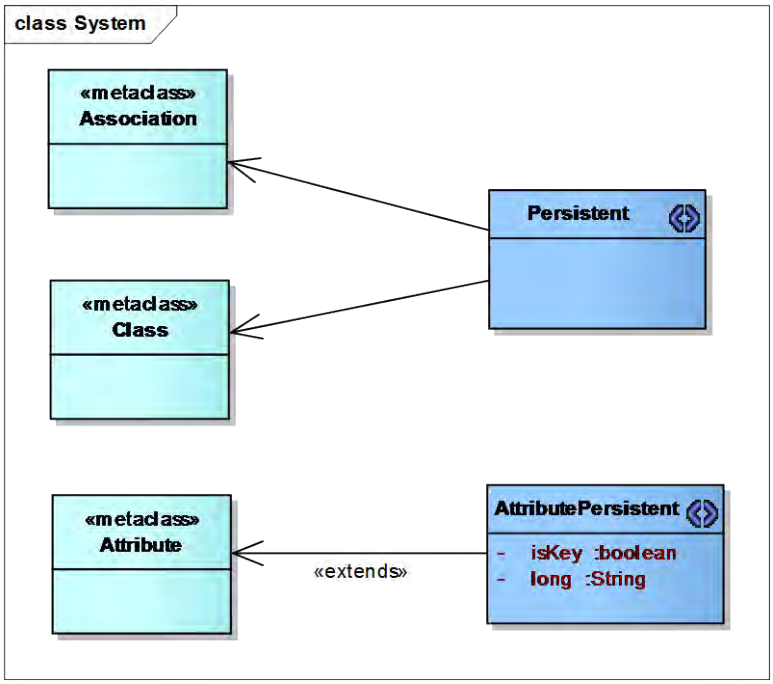


Figura 3.12. Ejemplo de estereotipo y valor etiquetado de un perfil UML.

Aunque en el ejemplo de la Figura 3.13 se ven los valores etiquetados asociados a un estereotipo, esto no tiene por qué ser siempre así. Se pueden definir valores etiquetados que no formen parte de ningún estereotipo y que extiendan a su vez propiedades de los elementos del meta-modelo UML. Por ejemplo, para el caso anterior, se podría definir el valor etiquetado isTx para indicar que el elemento Operation especifica una operación transaccional, de manera que, si no se ejecuta con éxito, se le aplica un roll back a todos los cambios realizados por la misma, como se ve en la siguiente figura.

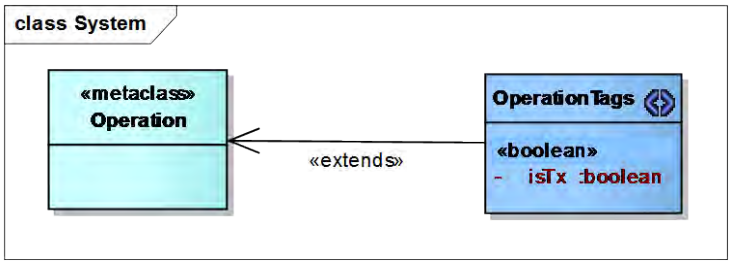


Figura 3.13. Valor etiquetado asociado a una metaclase.

3.4.2.2.3 El uso de marcas

MDA propone la conversión de un PIM sin detalles, que capte la funcionalidad y estructura del sistema, a otro PSM que mantenga toda la información del predecesor pero que, a su vez, contenga los detalles suficientes para transformarlo a código. Sin embargo, este mapeo no está exento de problemas.

En uno de los diagramas del PIM se pueden tener una relación como la de la Figura 3.14, en la cual se ve cómo un Alumno tendrá una colección de asignaturas, lo cual podría dar lugar a un mapeo en el que Alumno tendría un método getAsignaturas() para obtener dicho conjunto. Pero, a la hora de realizar esta transformación a una plataforma dada, por ejemplo J2EE, se presenta el problema de cómo representar ese conjunto, teniendo en cuenta que se pueden tener distintos tipos de colecciones en Java (interfaz Collection) dependiendo, por ejemplo, de si pueden albergar objetos duplicados o no.

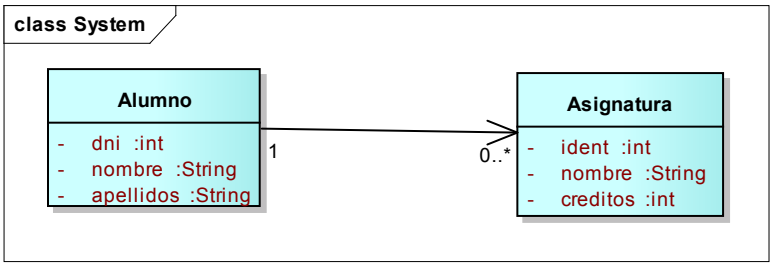


Figura 3.14. Relación Curso/Asignaturas.

Si se utiliza la aproximación de parametrizar la transformación para que todas las colecciones se transformen al mismo tipo concreto, sólo quedaría la alternativa de que una vez obtenido el PSM, un desarrollador curtido en la tecnología a implementar lo modificará para que cada colección tenga el tipo que se desee.
La solución podría pasar por parametrizar el modelo PIM añadiendo “marcas” o etiquetas a los elementos del modelo de manera que se pueda indicar qué colección usar. Claro está, de esta manera el PIM deja de ser un modelo independiente, con lo que lo ideal sería disponer de herramientas que permitan añadir diferentes tipos de etiquetado al modelo, pero que a la vez facilite conservarlo independiente de este etiquetado para seguir manteniendo la ventaja de la abstracción.
De este modo se puede disponer del PIM original y de un conjunto de PIM parametrizados según la alternativa elegida para cada plataforma.

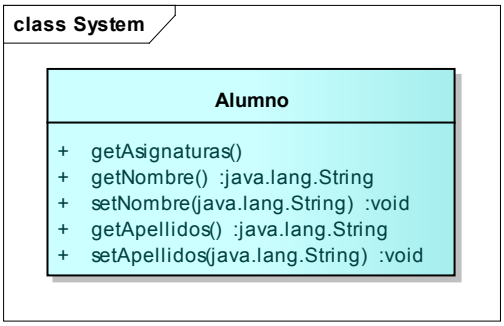


Figura 3.15 Implementación de Alumno.

Si se etiqueta un PIM se puede realizar su transformación a su correspondiente modelo en código. De manera que ya no se vería el PSM como un paso intermedio en el proceso de transformación sino que se podría conseguir el modelo implementado directamente. Así se tendrían dos niveles de modelos en el proceso de desarrollo, uno de ellos sí sería el PIM, con diferentes variantes etiquetadas.
De todas formas se podría seguir pensando en generar el PSM. Un PSM de solo lectura pero que proporcionara el beneficio de tener una ayuda visual de la Implementación en la plataforma elegida, que pueda ayudar a la depuración y fase de pruebas del modelo, por ejemplo, si se quisiera añadir alguna pre o post-condición. O si por ejemplo (y también lo más probable) la herramienta solo genera parte del código, el PSM puede servir a los programadores de la aplicación como una excelente guía-manual de lo que ya hay implementado. Así pues y aunque pueda que no sea estrictamente necesario, el nivel de PSM se justifica en este trabajo gracias a las ventajas que aporta.

3.4.2.2.4 Perfiles

Otra posibilidad planteada es extender el meta-modelo UML mediante su mecanismo de perfiles que presenta para tal tarea.
Los diseñadores de UML tomaron la decisión de no hacer UML como un lenguaje de modelado para cualquier cosa que la gente necesitara. Por ello prefirieron hacerlo lo más sencillo posible y a la vez dotarlo de un mecanismo de extensión. Este mecanismo de extensión permite, a través de una herramienta UML, definir constructores de modelado adicionales basándose en los ya definidos por UML.
Un conjunto de extensiones constituye esencialmente un dialecto de UML, lo cual es denominado un perfil. UML podría ser visto no como un lenguaje, sino como la base para una familia de lenguajes basados en él. MDA hace fuerte uso del mecanismo de perfiles debido a la necesidad de soportar diferentes aspectos del sistema y niveles de abstracción.
El mecanismo de extensión que UML proporciona se basa en estereotipos (Stereotypes) y valores etiquetados (tagged values), a los que se debe añadir las condiciones que deben cumplir los elementos del modelo para que esté bien formado mediante restricciones OCL. Un perfil UML es la definición de un conjunto de estereotipos y valores etiquetados que extienden los elementos del meta-modelo de UML.

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 .NET o J2EE) o de un dominio de aplicación (tiempo real, 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, sea el del propio UML, o el de otro Perfil:

- 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 del sistema terminología propia de EJB como “Home interface”, “entity bean”, “archive”, etc.).

- Definir una sintaxis para construcciones que no cuentan con una notación propia (como sucede con las acciones).

- Definir una nueva notación para símbolos ya existentes, más acorde con el dominio de la aplicación objetivo (poder usar, por ejemplo, una figura con un ordenador en lugar del símbolo para representar un nodo que por defecto ofrece UML para representar ordenadores en una red).

- Añadir cierta semántica que no aparece determinada de forma precisa en el metamodelo (por ejemplo, la incorporación de prioridades en la recepción de señales en una máquina de estados de UML).

- Añadir cierta semántica que no existe en el metamodelo (por ejemplo relojes, tiempo continuo, etc.).

- Añadir restricciones a las existentes en el metamodelo, restringiendo su forma de utilización (por ejemplo, impidiendo que ciertas acciones se ejecuten en paralelo dentro de una transición, o forzando la existencia de ciertas asociaciones entre las clases de un modelo).

- Añadir información que puede ser útil a la hora de transformar el modelo a otros modelos, o a código.

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). Para ilustrar estos conceptos utilizaremos un pequeño ejemplo de Perfil UML, que va a definir dos nuevos elementos que pueden ser añadidos a cualquier modelo UML: colores y pesos.

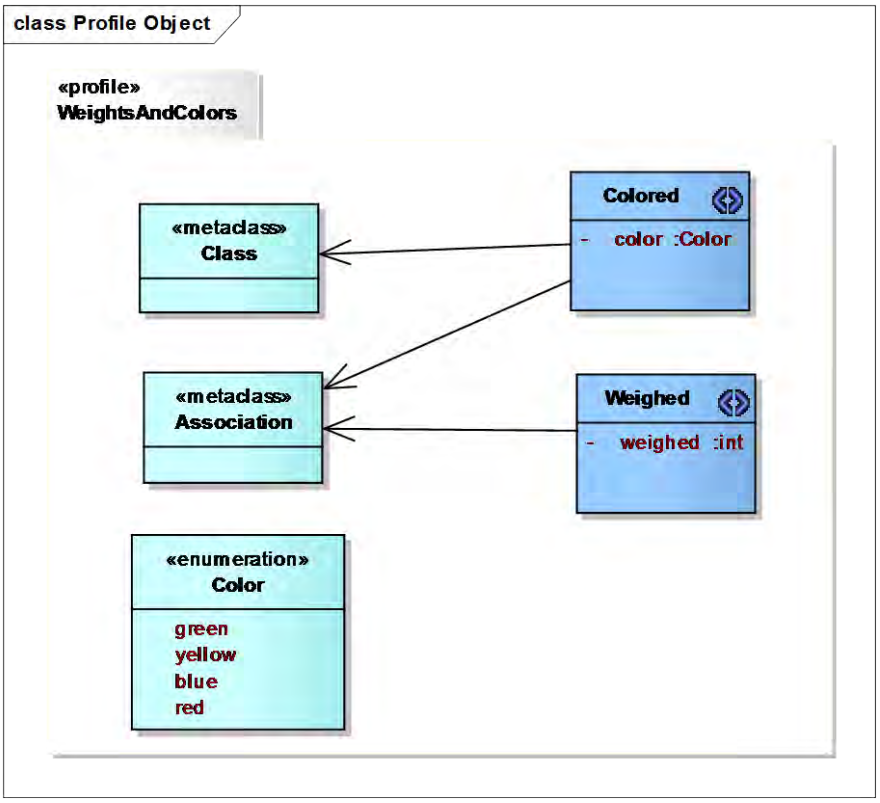


Figura 3.16. Ejemplo de Perfil UML

(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, estereotipadas «stereotype». En nuestro ejemplo, el Perfil UML WeightsAndColors define dos estereotipos, Colored y Weighed, que proporcionan color y peso a un elemento UML. Tal y como se indica en el Perfil, sólo las clases y las asociaciones de UML pueden colorearse, y sólo las asociaciones pueden tener asociado un peso. Obsérvese cómo el Perfil especifica los elementos del metamodelo de UML sobre los que se pueden asociar los estereotipos estereotipados «metaclass», mediante flechas continuas de punta triangular en negrita.

(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 si dos o más clases están unidas por una asociación coloreada, el color de las clases debe coincidir con el de la asociación.
Dicha restricción se traduce en la siguiente restricción del Perfil UML, en el lenguaje OCL

(Object Constraint Language).

context UML::InfrastructureLibrary::Core::Constructs::Association
inv : self.isStereotyped("Colored") **implies**
self.connection->**forAll**(isStereotyped("Colored") **implies** color=self.color)

Las restricciones pueden expresarse tanto en lenguaje natural como en OCL. OCL es un lenguaje para consultar y restringir los elementos de un modelo, definido por OMG y es parte integrante de UML.

(3) Finalmente, un valor etiquetado es un meta-atributo 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, el estereotipo «Weighed» puede contar con un valor etiquetado denominado "weight", de tipo Integer, y que indicará el peso de cada asociación que haya sido estereotipada como «Weighed». Los valores etiquetados se representan de forma gráfica como atributos de la clase que define el estereotipo.

Es importante señalar que estos tres mecanismos de extensión no son de primer nivel, es decir, no permiten modificar metamodelos existentes, sólo añadirles elementos y restricciones, pero respetando su sintaxis y semántica original. Sin embargo, sí que son muy adecuados para particularizar un metamodelo para uno o varios dominios o plataformas existentes. Cada una de estas particularizaciones o adaptaciones viene definida por un Perfil, que agrupa los estereotipos, restricciones, y valores etiquetados propios de tal adaptación.

3.4.3 OBJECT CONSTRAINT LANGUAGE: OCL 2.0

Un diagrama UML normalmente no está lo suficientemente definido para mostrar todos los elementos relevantes de una especificación. Entre otros aspectos, existe la necesidad de describir restricciones sobre elementos del modelo. Normalmente las restricciones se escriben en lenguaje natural y están llenas de ambigüedades. Para poder escribir restricciones no ambiguas, se debía definir un lenguaje formal que lo permitiera, pero la desventaja de utilizar un lenguaje tradicional es que resultaría demasiado complejo de aplicar a los sistemas de modelado.

OCL (Object Constraint Language) se desarrolló para resolver esta dificultad. Es un lenguaje formal fácil de escribir y de leer.

Se desarrolló como un lenguaje de modelado de negocios, que garantiza que no provoca efectos sobre los modelos. Cuando una expresión OCL se evalúa sólo devuelve el valor solicitado en la misma. No modifica nada del modelo y por lo tanto, no cambia el estado del sistema. Por esta razón, OCL no es un lenguaje de programación ya que no se pueden invocar procesos o actividades que no sean consultas.

OCL es un lenguaje tipado ya que cada expresión tiene asociado un tipo. Para ser un lenguaje bien formado cada expresión debe cumplir con las reglas del lenguaje, de manera que por ejemplo, no permita comparar un entero con una cadena de caracteres. Cada Classifier definido en un modelo UML representa a distintos tipos del lenguaje OCL. No obstante, OCL tiene su propio conjunto de tipos predefinidos.

El lenguaje OCL puede ser utilizado para diferentes propósitos:

- Como lenguaje de consulta.
- Dentro del modelo de clase para expresar invariantes sobre clases y tipos.
- Para especificar tipos invariantes para Estereotipos.
- Para describir pre y postcondiciones sobre Operaciones y Métodos.
- Para describir controles.
- Para especificar objetivos para mensajes y acciones.
- Para especificar restricciones sobre operaciones.
- Para especificar reglas de derivación de atributos para una expresión sobre el modelo UML.

3.4.3.1 Fundamentos De OCL

3.4.3.1.1 Expresiones, tipos y valores en OCL

En OCL, cada valor, ya sea un objeto, una instancia de un componente o un valor de datos, tiene un tipo que define qué operaciones pueden ser aplicadas al objeto. Los tipos se dividen dentro de dos grupos: tipos predefinidos en la biblioteca estándar y tipos definidos por el usuario.
Dentro de los tipos predefinidos se pueden distinguir los básicos y de colección. Los tipos básicos son Integer, Real, String y Boolean, cuyas definiciones son similares a otros lenguajes. Dentro de las colecciones están Collection, Set, Bag, OrdenedSet y Sequence. Las colecciones se utilizan para especificar exactamente los resultados de navegación a través de las asociaciones en el diagrama de clases.
Los tipos definidos por el usuario se definen en los diagramas UML. Cada elemento instanciable del modelo, es decir, cada clase, interfaz, componente o tipo de datos en un diagrama UML es automáticamente un tipo de OCL.

Las expresiones OCL representan un valor, por tanto tiene un tipo, ya sea definido por el usuario o predefinido. Además, cada expresión tiene un resultado que será el valor que resulta de evaluar la expresión. El tipo del resultado es igual al tipo de la expresión.
OCL distingue entre tipos de valores y tipos de objetos. Ambos son tipos, ambos especifican instancias, pero hay una diferencia importante: los tipos de valor definen instancias que no cambian. Por ejemplo si tenemos un Integer con valor 1 siempre tendrá ese valor. En cambio los tipos de objetos o “Classifiers” representan tipos que definen instancias que pueden cambiar su valor o valores. Por ejemplo: una instancia de la clase Persona puede cambiar el valor del atributo Edad pero seguirá siendo la misma instancia.

Cada uno de los tipos básicos tiene una serie de operaciones. Además, dentro de las operaciones existen unas reglas de precedencia. Dentro de los operadores también se puede utilizar los operadores infijos.

3.4.3.1.2 Tipos definidos por el usuario

Cuando se define un tipo en un diagrama UML, a un tipo de usuario se le otorgan una serie de características. Cada una de estas características puede ser utilizada en una expresión OCL. Las características de un tipo definido por el usuario incluye:

1. Atributos.
2. Operaciones.
3. Atributos de Clase.
4. Operaciones de Clase.
5. Extremos de asociaciones que derivan de las asociaciones y las agregaciones.

Los atributos de un tipo definido por el usuario pueden ser utilizados en expresiones escribiendo un punto seguido del nombre del atributo. De forma similar, las operaciones de este tipo también se pueden utilizar en las expresiones. Existe una restricción, al ser OCL un lenguaje libre de efectos laterales, no está permitido el cambio de un objeto. Solamente se pueden realizar operaciones de consulta, que retornan un valor pero no cambian nada. De acuerdo con la especificación UML, cada operación tiene una etiqueta denominada "isQuery", cuyo valor verdadero indica que la operación no tiene efectos laterales y puede ser utilizada en expresiones.
La visibilidad de los atributos y las operaciones se ignora por defecto en OCL.
Opcionalmente OCL puede utilizar reglas dadas en la especificación UML. En este caso un atributo privado no puede utilizarse en una expresión OCL, dado que no es visible en la instancia contextual.

Las operaciones de clase y los atributos de clase también pueden utilizarse en expresiones. La sintaxis para referirse a los atributos de clase u operaciones será el nombre de la clase seguido por dos puntos y el nombre del atributo u operación y sus parámetros.
Otra de las características de los tipos definidos por el usuario se deriva de las asociaciones del modelo de clases. Cada asociación tiene un número de extremos.

Existe un tipo definido por el usuario especial, el tipo enumeración (enumeration). Este tipo se utiliza habitualmente como un tipo para los atributos. Se define dentro de un diagrama de clase de UML usando el estereotipo de enumeración.
El valor definido en la enumeración puede ser utilizado dentro de una expresión OCL.

3.4.3.2 SINTAXIS DE UN SUBCONJUNTO DE EXPRESIONES OCL

Antes de dar la gramática veremos que significa cada una de las palabras claves de la sintaxis OCL, adjuntando ejemplos para familiarizarse con las expresiones escritas en OCL.

Invariantes

Una expresión OCL puede ser parte de un invariante. Una expresión OCL es un invariante de tipo y debe ser verdadero para todas las instancias de ese tipo en cualquier momento.

Por ejemplo:
El siguiente invariante específica que el número de empleados deber ser siempre mayor que 72, en el contexto de tipo Banco. Este invariante vale para toda instancia de tipo Banco.

```
context Banco inv cantEmpleados:
    self.nroEmpleados > 72
```

El nombre cantEmpleados, podría omitirse, se utiliza para poder referenciar al invariante en otra expresión. En la mayoría de los casos, se utiliza la palabra **Self** ya que cada expresión OCL es escrita en el contexto de una instancia de un tipo específico y la palabra reservada self se usa para referirse a la instancia contextual. Otra opción equivalente sería:

```
context b:Banco inv cantEmpleados:
    b.nroEmpleados > 72
```

Precondiciones y Potscondiciones

Una expresión OCL puede ser parte de una Precondición o Poscondición, asociada a un Método u Operación. El nombre self puede ser usado en la expresión refiriéndose al objeto receptor de la operación. La palabra reservada result denota el resultado de la operación, si es que hay uno. Los nombres de los parámetros pueden ser usados en la expresión OCL.

```
context nombreDeTipo::NombreDeOperacion(param1 : Tipo1, ... ): TipoDeRetorno
    pre : alguna-expresion-ocl
    post: result = alguna-expresion-ocl
```

Por ejemplo:

```
context Persona::edad(fechaActual): Integer
post: result = fechaActual – self.fechaDeNacimiento
```

Valores previos en Postcondiciones

En una postcondición, la expresión puede referirse a dos conjuntos de valores:

- El valor de una propiedad al comenzar la operación o método.
- El valor de la propiedad luego de completar la operación o método.

Para referirnos al valor de una propiedad al comenzar la operación o método, se usa la palabra "@pre"

Ejemplo:

```
context Cuenta::depositar(n)
        post: saldo =saldo@pre+n
```

Paquete

Las fórmulas OCL pueden agruparse dentro de paquetes para facilitar su comprensión y manejo. Las declaraciones de paquetes tienen la siguiente sintaxis:

```
package Package::SubPackage
        …una lista de fórmulas ocl…
Endpackage
```

En un archivo OCL puede haber varias definiciones de paquetes, permitiendo a todos los invariantes, precondiciones, y poscondiciones estar escritos y almacenados en el mismo archivo.

Expresiones Let

La expresión let permite definir un atributo derivado o una operación para ser usada luego en otras expresiones ocl.
Por ejemplo: Se define un atributo indicando si el cliente tiene al menos una cuenta, para luego ser usado.

```
context Cliente inv:
        let tieneCuenta : Boolean = self.cuentas->
                notEmpty() inself.tieneCuenta implies self.edad()<21
```

Una expresión let puede ser incluida en un invariante, precondición o poscondición. Para poder reutilizar las operaciones y/o variables del let, se escribe la palabra def. Todas las variables y operaciones definidas en el alcance def son conocidas en el mismo contexto en el que cualquier propiedad del Classifier puede ser usada.

Veamos un ejemplo:

```
context Cliente def:
        let tieneCuenta : Boolean = self.cuentas-> notEmpty()
```

Operaciones de Colecciones

OCL define varias operaciones sobre los tipos Colecciones. Veamos algunas de ellas

Select y Reject

La operación select especifica un subconjunto de una colección y su sintaxis es la siguiente:
colección->select(c:Tipo | expresión-lógica-con-c)
El resultado de la operación select, son todos los elemento de la colección, para los cuales la expresión lógica-con-c resultó verdadera. La variable c es llamada iterador. Cuando el select es evaluado, c itera sobre la colección y la expresión-lógica-con-c es evaluada con cada c. El tipo de la variable iterador es opcional, con lo que nos quedaría otra forma equivalente:
colección->select(c | expresión-lógica-con-c)

Y se puede abreviar mediante: colección->select(expresión-lógica)

Por ejemplo, las tres formas de escribir un select son las siguientes (el invariante especifica que en el banco debe existir al menos una cuenta cuyo saldo sea mayor que cero):

```
context Banco inv:
        self.cuentas->select(c: Cuenta| c.saldo > 0) ->notEmpty()

context Banco inv:
        self.cuentas->select(c| c.saldo > 0) ->notEmpty()

context Banco inv:
        self.cuentas->select(saldo > 0) ->notEmpty()
```

El atributo self.cuentas es de tipo Set(Cuenta). El select toma cada cuenta cuyo saldo sea mayor a 0.

La operación reject puede expresarse como un select con la expresión booleana negada. Es decir, que las dos siguientes expresiones son idénticas:

```
colección-> reject ( c:Tipo | expresión-lógica-con-c )
colección-> select( c:Tipo | not expresión-lógica-con-c )
```

Otras operaciones que provee OCL para las colecciones

- Size
- Includes
- Excludes
- IncludesAll
- ExcludesAll
- IsEmpty
- NotEmpty
- Any

OCL está perfectamente integrado con EMF, por lo que lo hemos utilizado para realizar comprobaciones sobre el diagrama, como por ejemplo: restricciones para la correcta creación estructural del diagrama, estereotipado correcto de los componentes, relaciones y pertenencias de los componentes.

3.4.4 Metadata Interchange (XMI)

XMI es un estándar del OMG para el intercambio de información vía XML. Puede ser utilizado para todos los metadatos de los metamodelos que estén expresados en MOF. El uso más común de XMI es como formato de intercambio de modelos UML, es decir, se puede utilizar para la serialización de metamodelos.
Desde la perspectiva de modelado de OMG, los datos se dividieron en modelos abstractos y modelos concretos. Los modelos abstractos representan la información semántica, mientras que los modelos concretos representan diagramas visuales. Los modelos abstractos son instancias de lenguaje de modelado basado en MOF como puede ser UML. Para los diagramas se utiliza el XMI.

El propósito de XMI es facilitar el intercambio de los metadatos entre las herramientas de modelado UML y los repositorios de metadatos MOF en entornos heterogéneos.
XMI integra cuatro estándares: XML, UML, MOF y la correspondencia entre MOF y XMI La integración de estos cuatro estándares dentro de XMI permite herramientas de desarrollo para sistemas distribuidos de modelos de objetos y otros metadatos.

3.4.4.1 Tipos de modelos en función del nivel de abstracción

3.4.4.1.1

● **CIM** (Computation Independent Model): Modelado de negocio y requerimientos.
CIM debe su nombre a este foco en el negocio por sobre la tecnología, que en español se traduce como: “Modelo Independiente de la Computación”. El **CIM** se centra en los requerimientos y representa el nivel más alto del modelo de negocios. Usa un lenguaje para modelar procesos de negocios que no es **UML**, aunque este lenguaje puede ser derivado perfectamente utilizando **MOF** (meta-object facility). El CIM transciende a los sistemas; cada proceso de negocio interactúa con trabajadores humanos y/o componentes de máquinas. El CIM describe solamente aquellas interacciones que tienen lugar entre los procesos y las responsabilidades de cada trabajador, sea o no humano. Un objetivo fundamental del CIM, es que cualquiera que pueda entender el negocio y los procesos del mismo puede comprenderlo, ya que éste evita todo tipo de conocimiento especializado o de sistemas.

Figura 3.17. Modelado CIM

3.4.4.1.2

● **PIM** (Platform Independent Model)

El **PIM**, que se traduce al castellano como "Modelo Independiente de la Plataforma", representa el modelo de procesos de negocio a ser implementado. Comúnmente se usa **UML** o un derivado de UML para describir el PIM. El PIM modela los procesos y estructuras del sistema, sin hacer ninguna referencia a la plataforma en la/s que será desplegada la aplicación. A su vez, ignora los sistemas operativos, los lenguajes de programación, el hardware y la topología de red. Suele ser el punto de entrada de todas las herramientas para **MDA** e incluso de muchos artículos que hablan de MDA, dejando de lado el **CIM**.

Facilita la creación de diferentes implementaciones del sistema en diferentes plataformas, dejando intacta su estructura y su funcionalidad básica.

- Análisis y diseño independiente de la plataforma tecnológica.
- El analista realiza el PIM que describe el sistema.
- Un PIM se transforma en uno o más PSM.

La siguiente figura muestra un ejemplo de un **PIM** sencillo con tres clases interrelacionadas.

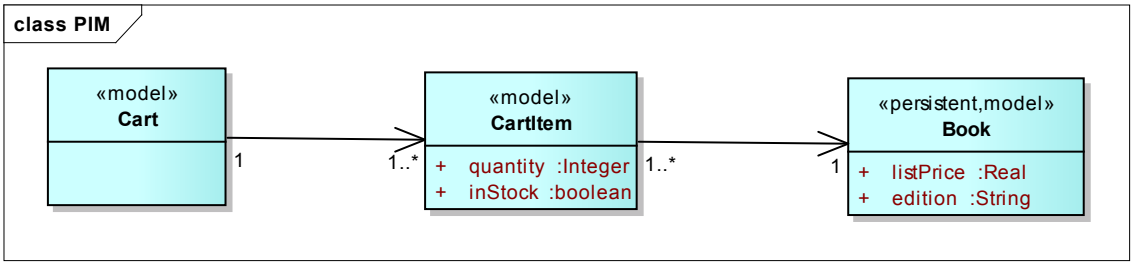


Figura 3.18. Ejemplo de PIM. Se modela mediante diagrama de clases UML.

3.4.4.1.3

● **PSM** (Platform Dependent Model)

El **PSM**, que se traduce al castellano como "Modelo Específico de la Plataforma", representa la proyección de los PIMs en una plataforma específica. Un PIM puede generar múltiples PSMs, cada uno para una tecnología distinta. Generalmente, los PSMs deben colaborar entre sí para una solución completa y consistente. Normalmente, esto se realiza en **UML**, creando distintos profiles (perfiles) que definen un PSM para cada tecnología requerida. Los PSMs tienen que lidiar explícitamente con los sistemas operativos, los lenguajes de programación, las plataformas (**CORBA**, **.Net**, **J2EE**, etc), etc.

El PSM es, un modelo del sistema de más bajo nivel, mucho más cercano a la vista del código que el PIM. Puede incluir más o menos detalle de acuerdo a su propósito:

- Diseño dependiente de la plataforma tecnológica
- El arquitecto adapta el modelo para una implementación tecnológica específica
- Cada PSM se puede transformar en código.

La figura 3.19, muestra un PSM construido a partir del PIM de la figura 3.18, representado también mediante un diagrama de clases UML. Este sencillo ejemplo muestra una porción del modelo de libros donde se representa la plataforma del modelo de negocio. En el paso del PIM al PSM se han producido varias transformaciones:

- Se han modificado la visibilidad de los atributos del PIM al PSM de públicos a privados.
- Se han añadido tipos a las variables de acuerdo al lenguaje Java en este caso.
- Se han agregado métodos públicos de lectura y modificación (get y set) para cada atributo.
- Se han agregado las clases DAOs y Service según las clases estereotipadas del PIM (esta parte se mostrará más adelante en el capítulo práctico).

Para la construcción de los PSMs se usan los perfiles **UML Profiles**, que son extensiones de UML que permiten añadir información semántica a los modelos para expresar detalles específicos de la plataforma.

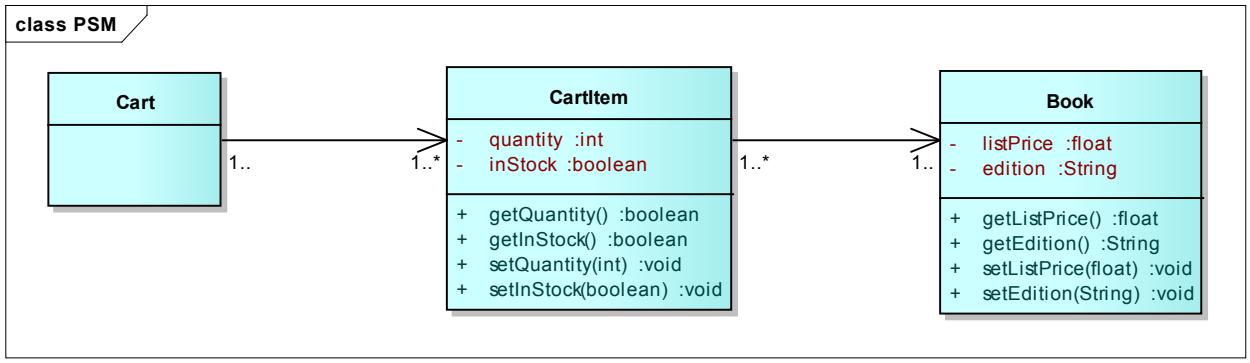


Figura 3.19 Modelo PSM

Hay que destacar que a partir de un PIM pueden generarse varios PSMs, cada uno describiendo el sistema desde una perspectiva diferente.

La transformación del PIM al PSM puede llevarse a cabo de varias formas:

- Construyendo manualmente el **PSM** a partir del **PIM**.
- De forma semiautomática generando un **PSM** esqueleto que es completado a mano.
- De forma totalmente automática generando un **PSM** a partir del **PIM**.

Para las transformaciones automáticas se usan herramientas especializadas. Estas herramientas tienen implementado distintos algoritmos de transformación para pasar de un modelo a otro, y suponen uno de los pilares de MDA.

Un PSM también puede refinarse, transformándose sucesivamente en PSMs de más bajo nivel, hasta llegar al punto en que pueda ser transformado de manera directa a código.

A partir del PSM, y gracias nuevamente a una herramienta de transformación, se obtiene gran parte del código que implementa el sistema para la plataforma elegida. El desarrollador tan solo tendrá que añadir aquella funcionalidad que no puede representarse mediante el PIM o el PSM.

3.4.5 Code Model

El modelo de código representa el código desplegable (deployable), normalmente en un lenguaje de programación de alto nivel, como **Java**, **C#**, **C++**, **VB**, **JSP**, etc. Idealmente, el modelo de código está listo para compilar y no debería requerir la intervención humana; el despliegue de la aplicación podría ser automatizado. Según los puristas y algunos fanáticos de **MDA**, en un ambiente **MDA** maduro no se debería pensar en el código más que como simples archivos, o como un mero objeto intermedio para generar el ejecutable final. Pero debido a que **MDA** no está maduro, y difícilmente se llegue alguna vez a la utopía de no tener que tocar ningún código, los desarrolladores seguirán necesitando conocer la tecnología para complementar la generación de código, debuguear la aplicación y, lidiar con errores inesperados, extraños y divertidos.

3.4.6 Beneficios de la separación entre modelos PIM y PSM

_ Facilita validar **correctitud** del sistema en el modelo PIM al estar desligado de aspectos de plataforma tecnológica
_ Facilita **producir** implementaciones del mismo sistema en diferentes plataformas tecnológicas, cumpliendo los objetivos de negocio planteados en el modelo PIM
_ La **integración** e **interoperabilidad** entre varios sistemas se puede definir de manera más clara en los modelos PIM, y luego se puede transformar a mecanismos específicos de plataforma (modelos PSM)

3.4.7 Desarrollo tradicional vs. Desarrollo con MDA

Problemas del desarrollo tradicional

El desarrollo de sistemas software siempre ha sido una labor intensa, pero a medida que ha ido evolucionando la tecnología, esta labor se ha complicado cada vez más. La evolución de los lenguajes, entornos y técnicas de programación provoca que haya que desarrollar los mismos sistemas una y otra vez, que éstos utilicen e integren diferentes tecnologías o que exista una necesidad de comunicación entre sistemas dispares.

Un proceso típico del desarrollo de software incluye las siguientes fases:

1_ Recogida de Requisitos
2_ Análisis
3_ Diseño
4_ Codificación
5_ Prueba
6_ Despliegue

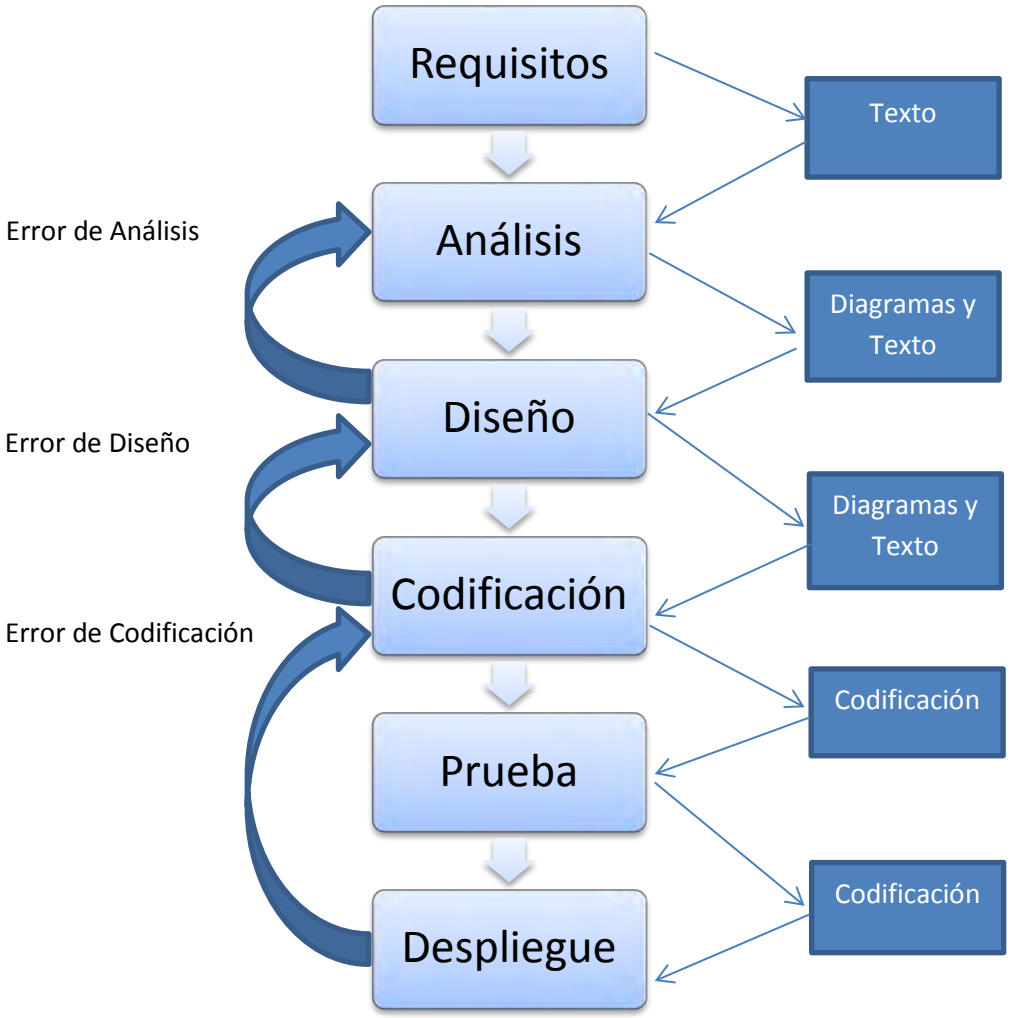


Figura 3.20. Proceso de desarrollo de Software Tradicional.

Durante los últimos años se han hecho muchos progresos en el desarrollo de software, que han permitido construir sistemas más grandes y complejos. Aun así la construcción de software tradicional sigue teniendo múltiples problemas:

- **Productividad:** El proceso tradicional produce una gran cantidad de documentos y diagramas para especificar requisitos, clases, colaboraciones, etc. La mayoría de este material pierde su valor en cuanto comienza la fase de codificación, y gradualmente se va perdiendo la relación entre los diagramas. Y más aún cuando el sistema cambia a lo largo del tiempo: realizar los cambios en todas las fases (requisitos, análisis, diseño...) se hace inmanejable, así que generalmente se hacen las modificaciones en el código. Por lo tanto, cabría preguntarse si vale la pena perder un tiempo precioso en una definición a alto nivel de la especificación del sistema, una vez iniciada la codificación. Evidentemente, la respuesta negativa sería impensable, ya que incumpliría todos los principios de modelado de la ingeniería del software, pero en los casos en los que no se actúe conforme a las metodologías definidas, se podría llegar a una conclusión de que se dedica mucho tiempo en mantener y que por lo tanto ver como más productivo la codificación.
  A pesar de que exista la tentación de plantearse esta pregunta, lo que no se puede negar es que, en proyectos de desarrollo, ambas tareas son necesarias y tienen su importancia, por lo que hay que se plantea la necesidad de encontrar un mecanismo que facilite la generación y el mantenimiento de la documentación de los sistemas software. Un soporte para que, un cambio en cualquiera de las fases se traslade fácilmente al resto.

- **Portabilidad**: En la industria del software, cada año nacen nuevas tecnologías y las empresas necesitan adaptarse a ellas, bien porque la demanda de esa tecnología es alta o bien porque realmente resuelve problemas importantes. Como consecuencia, el software existente debe adaptarse o migrar a la nueva tecnología. Esta migración no es ni mucho menos trivial, y obliga a las empresas a realizar un importante desembolso.

- **Interoperabilidad**: La mayoría de sistemas necesitan comunicarse con otros, probablemente ya construidos. Incluso si los sistemas que van a interoperar se construyen desde cero, frecuentemente usan tecnologías diferentes. Por ejemplo, un sistema que use *Enterprise JavaBeans,* necesita también bases de datos relacionales como mecanismo del almacenamiento de datos. Necesitamos que la interoperabilidad entre sistemas, nuevos o ya existentes, se consiga de manera sencilla y uniforme.

- **Mantenimiento y Documentación**: Documentar un proyecto de Software es una tarea lenta que consume mucho tiempo, y que en realidad no interesa tanto a los que desarrollan el software, si no a aquellos que lo modificarán o lo usarán más adelante. Esto hace que se ponga poco empeño en la documentación y que generalmente no tenga buena calidad. La solución a este problema a nivel de código es que la documentación se genere directamente del código fuente, asegurándonos que esté siempre actualizada. No

obstante, la documentación de alto nivel (diagramas y texto) todavía debe ser mantenido a mano.

Estos problemas se solucionan en MDA como veremos a continuación:

3.4.8 Beneficios del MDA:

Problema: <u>Productividad</u>

- **Solución**: En MDA el foco del desarrollo recae sobre el PIM. Los PSMs se generan automáticamente (al menos en gran parte) a partir del PIM. Por supuesto alguien tiene que definir las transformaciones exactas, lo cual es una tarea especializada y difícil. Pero una vez implementada la transformación, puede usarse en muchos desarrollos. Y lo mismo ocurre con la generación de código a partir de los PSMs. Este enfoque centrado en el PIM aísla los problemas específicos de cada plataforma y encaja mucho mejor con las necesidades de los usuarios finales, puesto que se puede añadir funcionalidad con menos esfuerzo. El trabajo "sucio" recae sobre las herramientas de transformación, no sobre los desarrolladores.

Problema: <u>Portabilidad</u>

- **Solución**: En MDA la portabilidad se logra también enfocando el desarrollo sobre el PIM. Al ser un modelo independiente de cualquier tecnología, todo lo definido en él será totalmente portable. Otra vez el peso recae sobre las herramientas de transformación, que realizarán automáticamente el paso del PIM al PSM de la plataforma deseada.

Problema: <u>Interoperabilidad</u>

- **Solución**: Los PSMs generados a través de un mismo PIM generalmente tendrán relaciones, que es lo que en MDA se llaman **"puentes"**. Normalmente los distintos PSMs no podrán comunicarse entre ellos directamente, ya que pueden pertenecer a distintas tecnologías. Este problema lo soluciona MDA generando no solo los PSMs, sino también los **puentes** entre ellos. Como es lógico estos puentes serán construidos por las herramientas de transformación, que como vemos son uno de los pilares de MDA.

Problema: <u>Mantenimiento y Documentación</u>

- **Solución**: Como ya hemos dicho, a partir del PIM se generan los PSMs y a partir de los PSMs se genera el código. Básicamente, el PIM desempeña el papel de la documentación de alto nivel que se necesita para cualquier sistema de software. Pero la gran diferencia es que el PIM no se abandona tras la codificación. Los cambios realizados en el sistema se reflejarán en todos los niveles, mediante la regeneración de los PSMs y el código. Aun así, seguiremos necesitando documentación adicional que no puede expresarse con el PIM, por ejemplo, para justificar las elecciones hechas para construir el PIM.

3.4.9 El nuevo proceso de desarrollo

Si comparamos el proceso MDA con el proceso tradicional de desarrollo de software, vemos que las fases recogidas de requisitos, de prueba y de despliegue seguirán igual.
Lo que cambia son, las fases de **análisis**, **diseño** y **desarrollo** de la siguiente manera:

- **Análisis:** un grupo de personas desarrollarán el PIM guiado por las necesidades del negocio y la funcionalidad que debe incorporar el sistema.
- **Diseño:** otro grupo de personas se encargarán de las transformaciones del PIM a uno o más PSMs. Estas personas tendrán conocimientos suficientes para poder elegir la plataforma y arquitectura que mejor se adapte a los requisitos del sistema y establecer los parámetros de las distintas transformaciones. Los creadores de los PSMs tendrán comunicación constante con los desarrolladores de los PIM para tener más información sobre el sistema. (p.e para conocer los requisitos no funcionales)
- **Codificación:** esta parte se reduce a generar el código del sistema mediante herramientas especializadas. Los programadores únicamente tendrán que añadir la funcionalidad que no puede reflejarse en los modelos y, si es necesario, retocar el código generado.

Pero en este nuevo proceso de desarrollo aún falta un tercer grupo de personas, aquellas que escriben definiciones de transformaciones. Estas transformaciones son fundamentales para construir software con MDA, pues permiten a las herramientas pasar de PIM a PSM y de PSM a código.
La figura 3.21. Muestra un esquema simple que ilustra este nuevo proceso de desarrollo, y gracias a las herramientas de transformación, estos cambios se trasladan rápidamente al resto de las fases.

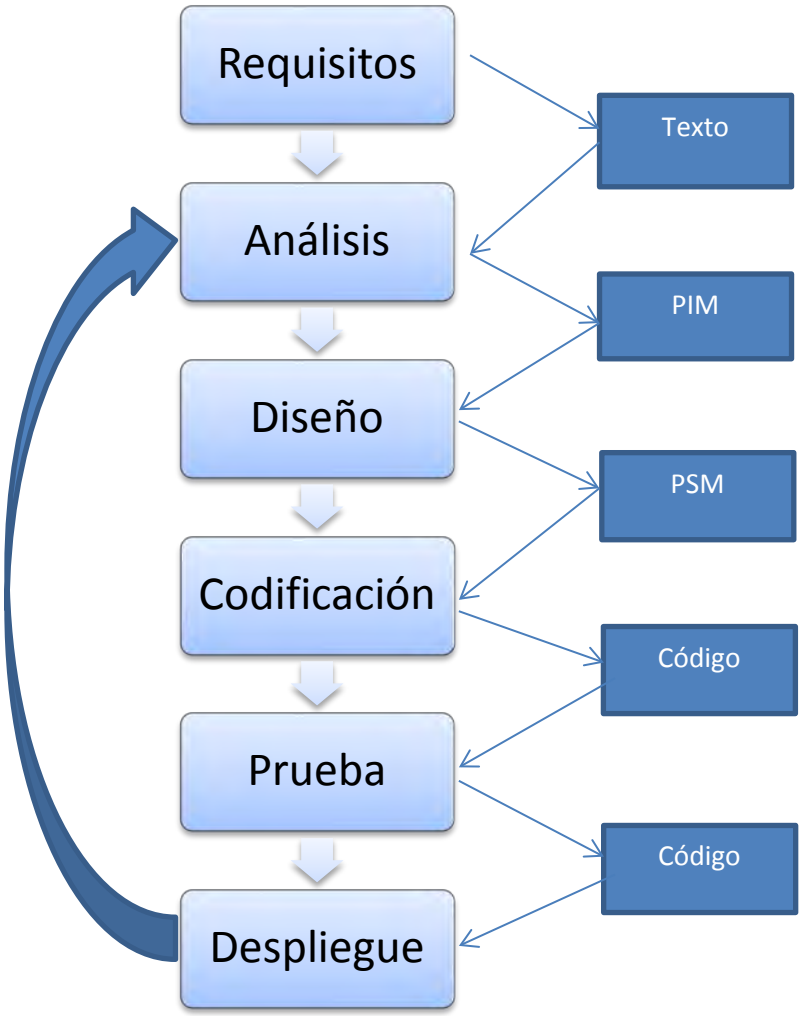


Figura 3.21. Proceso de desarrollo con MDA.

3.4.10 Visión de MDA

Dada la complejidad del desarrollo de una solución software, se hace necesario la implementación de estrategias para alcanzar beneficios fundamentales como son la productividad, la interoperabilidad, la portabilidad y la facilidad de mantenimiento, es por esto que propuestas como la de MDA se fundamentan en principios básicos como los expuestos en el manifiesto MDA planteado por IBM:

- **Representación directa** para enfocarse en el dominio del problema más que en la tecnología.

- **Automatización** de las tareas mecánicas que no requieren intervención humana. MDA incrementa la velocidad de desarrollo y reduce errores usando herramientas automatizadas para transformar modelos específicos de plataforma en código de implementación

- **Estándares abiertos**, los estándares de la industria no solo ayudan a eliminar la diversidad gratuita, sino que también alientan a los vendedores a producir herramientas tanto para propósitos generales como para propósitos especializados. El desarrollo de código abierto asegura que los estándares se implementan consistentemente y animan la adopción de estándares abiertos por parte de los vendedores. Posibilitan la interoperabilidad de las herramientas y plataformas.

Tal como se observa en la Figura **3.22**, estos tres elementos evidencian la importancia que tienen las herramientas en el proceso de desarrollo basado en MDA, para posibilitar la construcción de modelos, apoyar su transformación y permitir la interdisciplinariedad de los participantes, a través de la integración de las diversas plataformas y herramientas que estos utilizan.

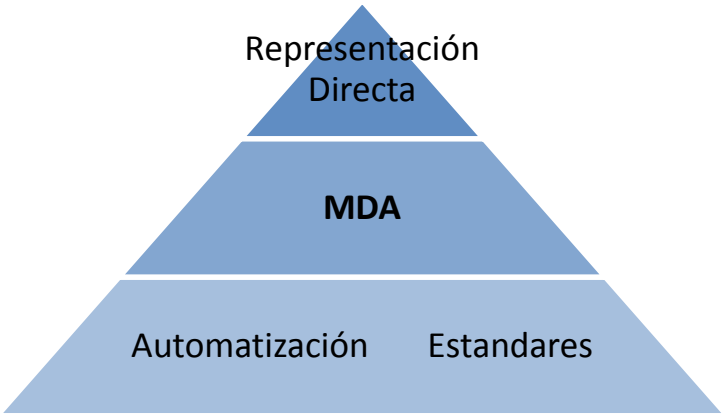


Figura 3.22.Visión alternativa de MDA.

3.4.11 Construyendo Metamodelos

3.4.11.1 ¿Qué es un metamodelo y para qué sirve?

Un metamodelo es un modelo de un lenguaje de modelado.

- Especifica los conceptos del lenguaje de modelado
- Especifica exactamente qué significa un modelo
- Simplifica la comunicación entre los interesados en un modelo

  Ejemplo: en lugar de explicar “En el modelo un Cliente es algo que tiene atributos y operaciones y que persiste tanto como el sistema viva”, se puede decir “En el modelo un Cliente es una Entidad” siempre y cuando el concepto Entidad esté definido en el Metamodelo.

- En base a conceptos de metamodelos se puede establecer de forma concreta las funciones de transformación de un modelo a otro.

  Ejemplo: cada Class de un modelo 1 se transformará a un Entidad del modelo 2.

3.4.11.2 Ejemplo de un metamodelo

Subconjunto del metamodelo del lenguaje UML:

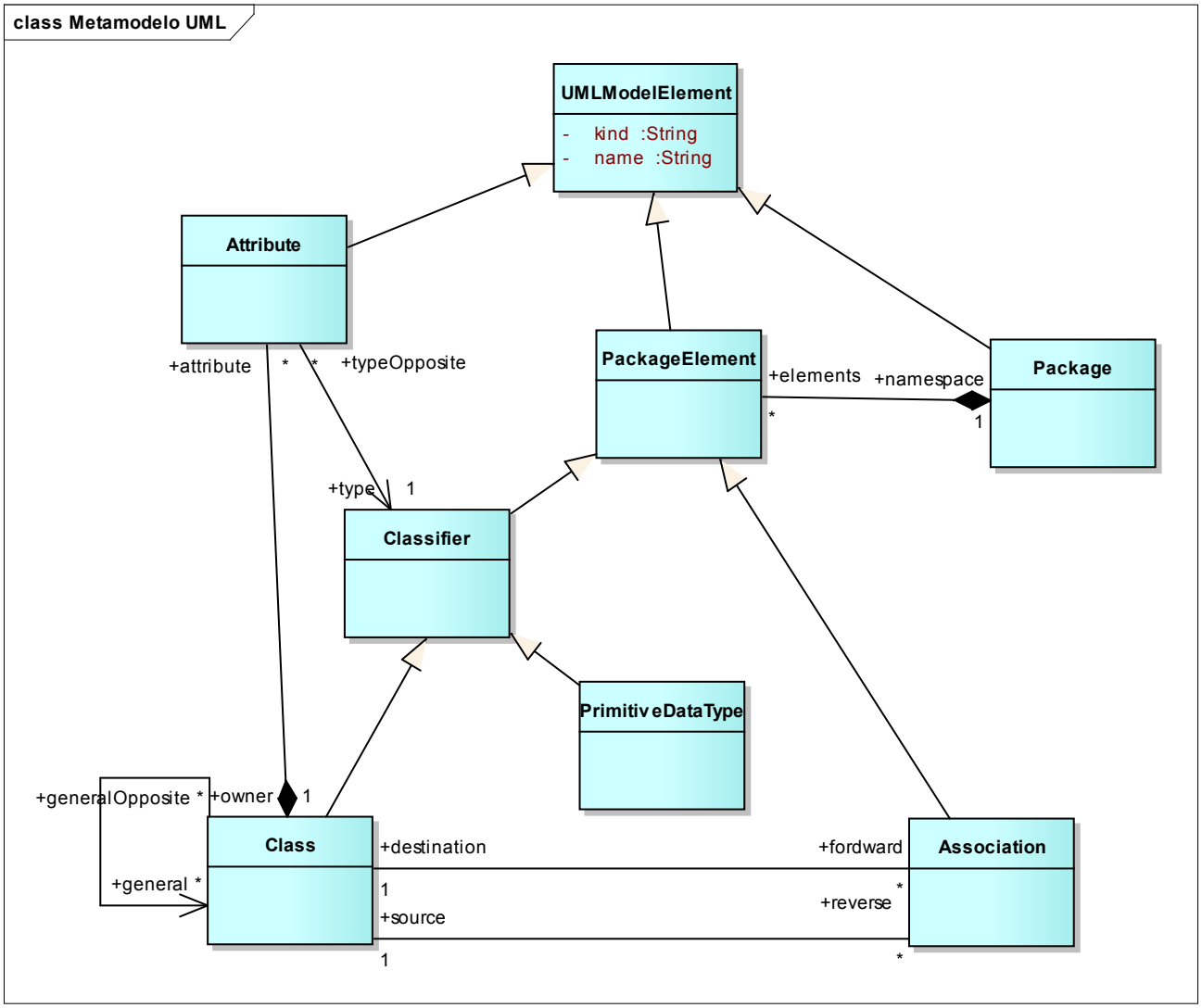


Figura 3.23. Fragmento del metamodelo de UML.

Los elementos de un modelo son instancias de los conceptos del metamodelo.

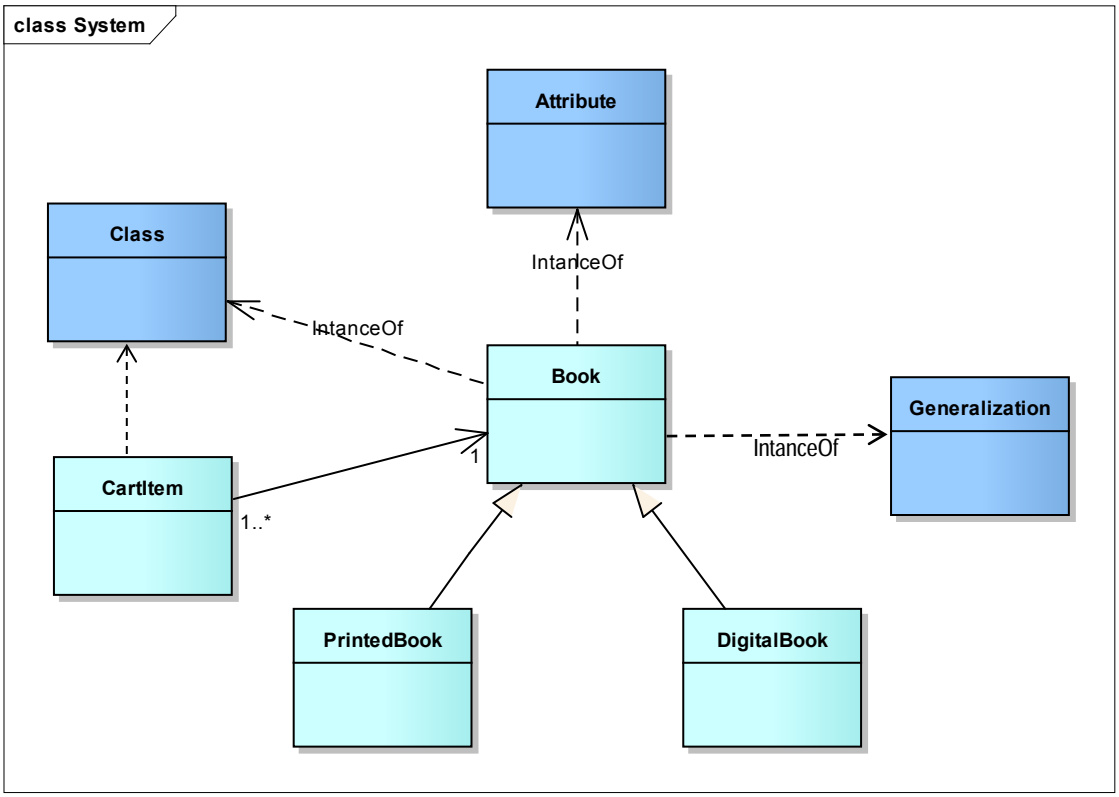


Figura 3.24. Ejemplo de clases instancias del metamodelo UML.

3.4.12 Fundamentos de MDA

3.4.13 Meta-modelos y MOF

El lenguaje de modelado UML es el principal mecanismo en la que se apoyan hoy en día las herramientas de diseño de aplicaciones. Sin embargo, cada vez más se exigen técnicas de diseño más especializadas, enfocadas a ciertos ámbitos de negocio sobre los que se quiere desarrollar nuevas aplicaciones. Y justo ahí es donde el UML clásico empieza a presentar sus grietas.

Este escenario impulsa la necesidad de disponer de nuevos lenguajes específicos del dominio que se adapten mejor al ámbito de negocio de las aplicaciones que se quieran desarrollar. Nuevos lenguajes de modelado que puedan ser usados en el ámbito MDA, como PIMs o PSMs.
Para ello, el OMG pone a disposición MOF (MetaObject Facility), un lenguaje para describir lenguajes de modelado. Un lenguaje de modelado puede usar MOF para definir formalmente la sintaxis abstracta de su conjunto de constructores de modelos. En otras palabras, con MOF se pueden definir tales constructores formalmente. Pero además, un **meta-modelo** también especifica semántica informal por medio del lenguaje natural; y es esta combinación de definiciones formales e informales a lo que se puede llamar un meta-modelo MOF, o si se prefiere simplemente un modelo MOF.

MOF es un estándar hermano de UML y es una de los elementos en los que se asienta MDA incluso mucho más que UML y sus perfiles, ya que hasta éstos son definidos vía MOF. Pero la relación con UML no acaba ahí. MOF toma prestados de UML los constructores del modelo de clases orientado a objetos y los presenta como la norma para describir la sintaxis abstracta de los constructores de modelado para definir la sintaxis abstracta de un meta-modelo. Por lo tanto, los meta-modelos MOF son como los diagramas de clases UML y en consecuencia, se pueden usar herramientas de diseño UML para crearlos. Con MOF se modela un constructor de modelado como una clase y las propiedades del constructor como los atributos de la clase. Modelando además las relaciones entre constructores como asociaciones.

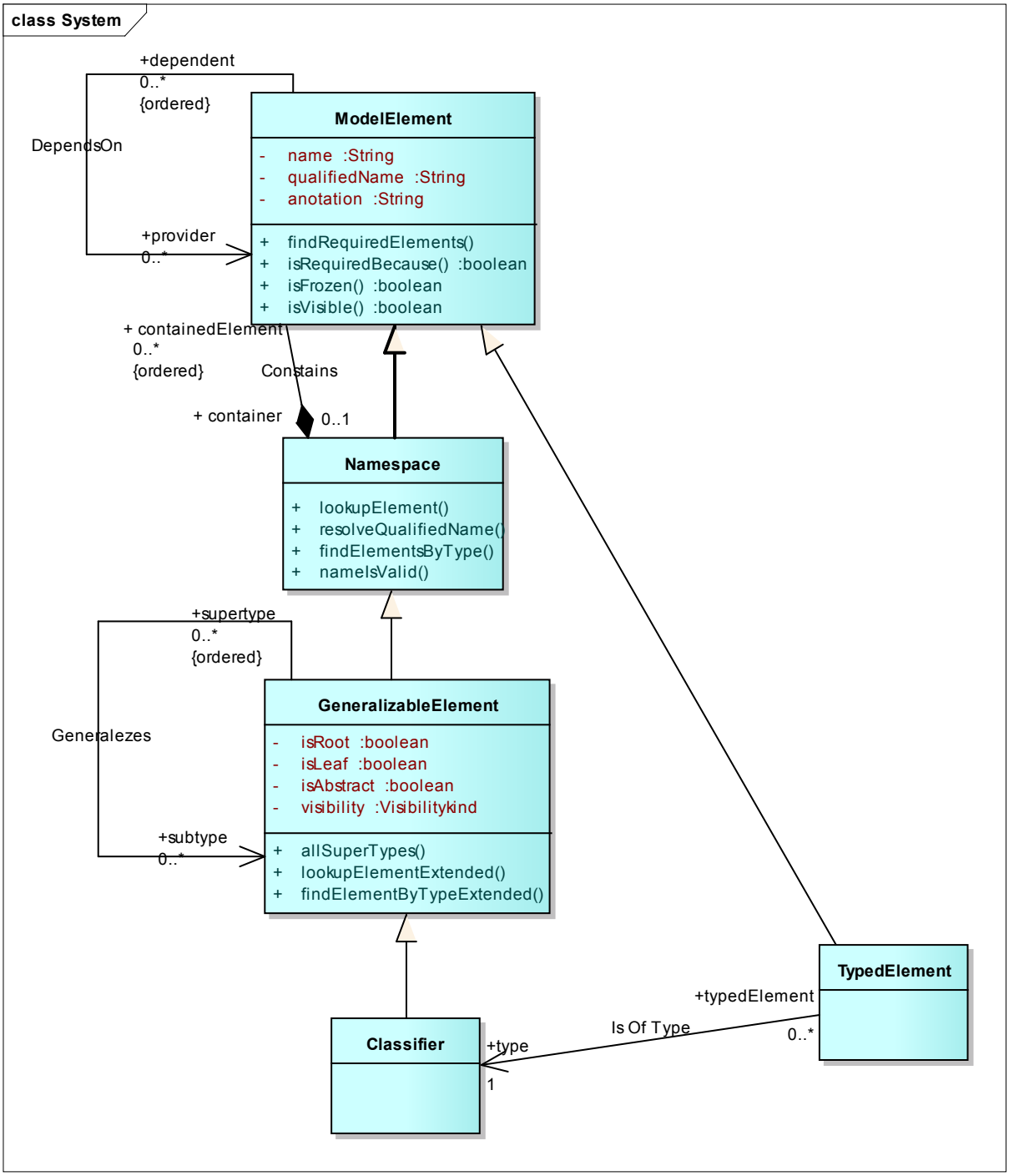


Figura 3.25. Abstracciones principales de MOF.

Para poner un poco de orden al lenguaje de meta-modelos MOF y todo lo que se puede derivar del mismo, el OMG ha establecido una arquitectura de cuatro niveles de capas de modelado. Estos niveles son conocidos como M3, M2, M1 y M0.

- **Nivel M3:** Está formado por un lenguaje capaz de definir nuevos metamodelos (MOF).No se necesitan más niveles por encima, puesto que el OMG estableció que MOF pudiera ser definido por sí mismo, siendo en cierto modo también un meta-modelo. Esto supone que todos los metamodelos de la capa M2, por ejemplo, el metamodelo de UML, son instancias de MOF. O lo que es lo mismo **UML se define usando MOF.**

M3: metamodelo del metamodelo

Ejemplo: MOF como metamodelo del metamodelo de UML indica propiedades del metamodelo M2, que son requeridas por las herramientas de intercambio.
Ejemplo: La figura 3.26 muestra un ejemplo de entidades de la capa M3.

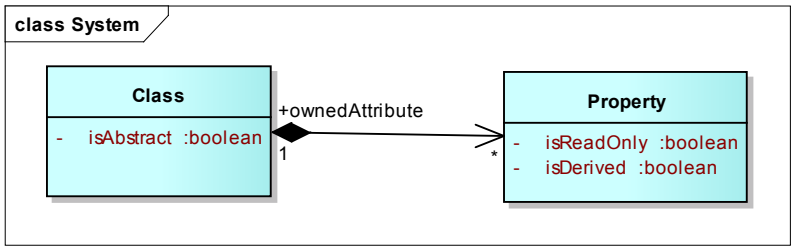


Figura 3.26. Entidades de MOF (Capa M3)

- **Nivel M2:** Lo forman todos aquellos meta-modelos que son instancias de MOF, es decir, que son definidos bajo éste. Hay un gran número de estos meta-modelos, incluyendo estándares tales como UML, CWM y CCM que son construidos usando elementos tales como clases, atributos y asociaciones MOF.

M2: metamodelo asociado al modelo de la aplicación.

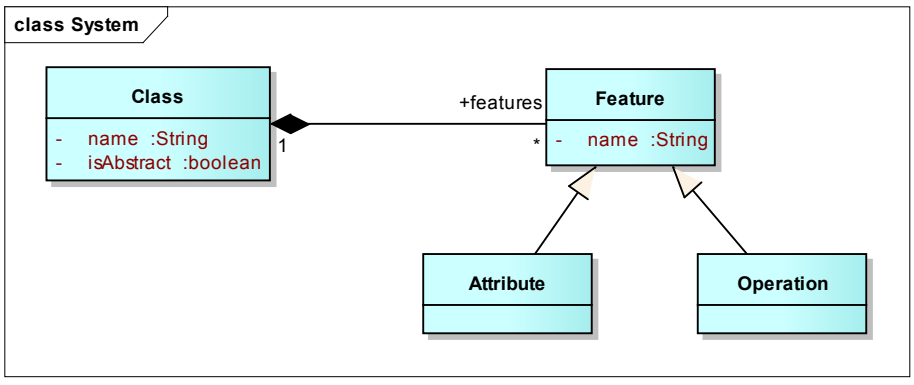


Figura 3.27. Ejemplo: metamodelo de UML

La clase de Class, **UML Class** es una instancia de la clase de **M3 MOF Class.**

- **Nivel M1:** Este nivel lo acaparan todos los modelos que son considerados instancias de M2.

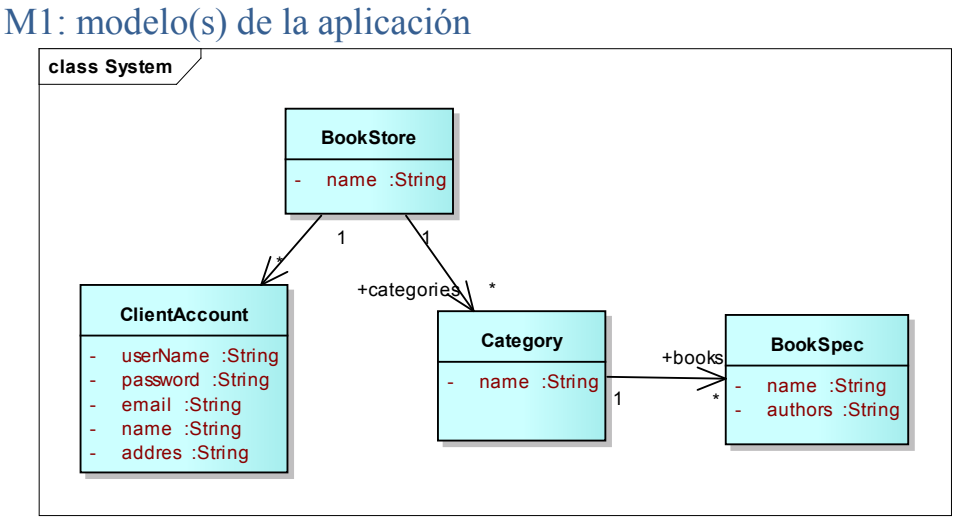


Figura 3.28. ej: modelo de clases UML (modelo de clases, modelo de objetos)

El cliente de nombre Juan García puede verse como una instancia de la entidad Cliente (ClientAccount) y 'Cien Años de Soledad' como una instancia de Libro (BookSpec). En la siguiente figura se muestra la relación entre el nivel M0 y el nivel M1.

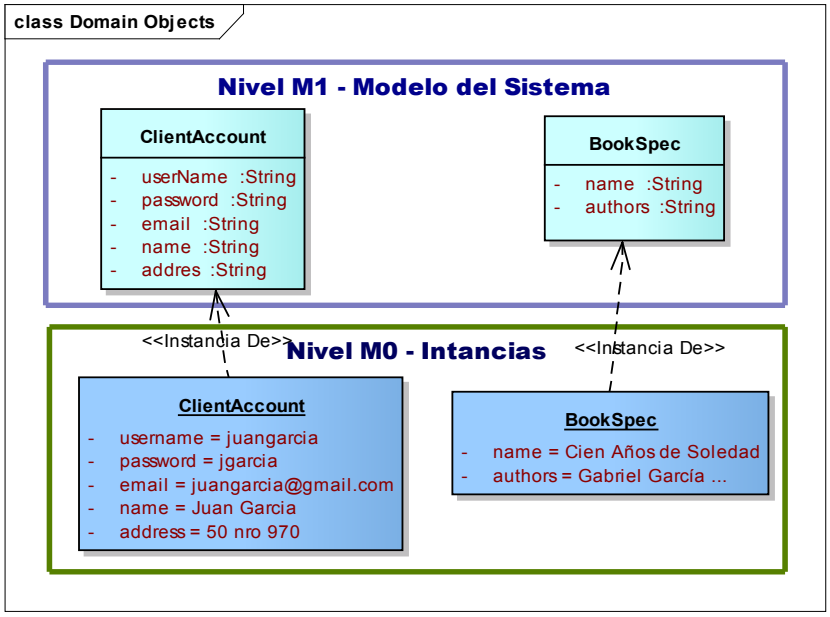


Figura 3.29. Relación entre el nivel M0 y el nivel M1

- **Nivel M0:** En este nivel están todos los objetos y datos que son instancias de elementos del nivel M1. Aquí ya no se trata con clases ni atributos, sino solo entidades físicas.

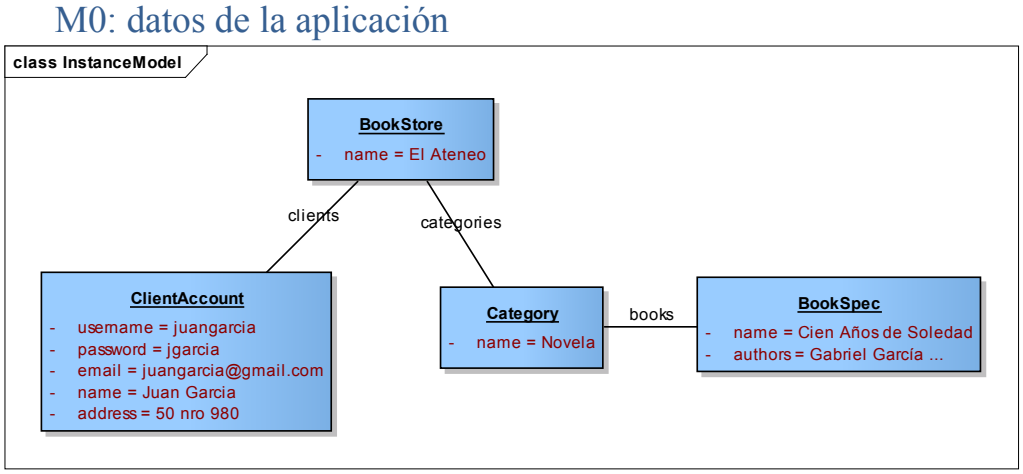
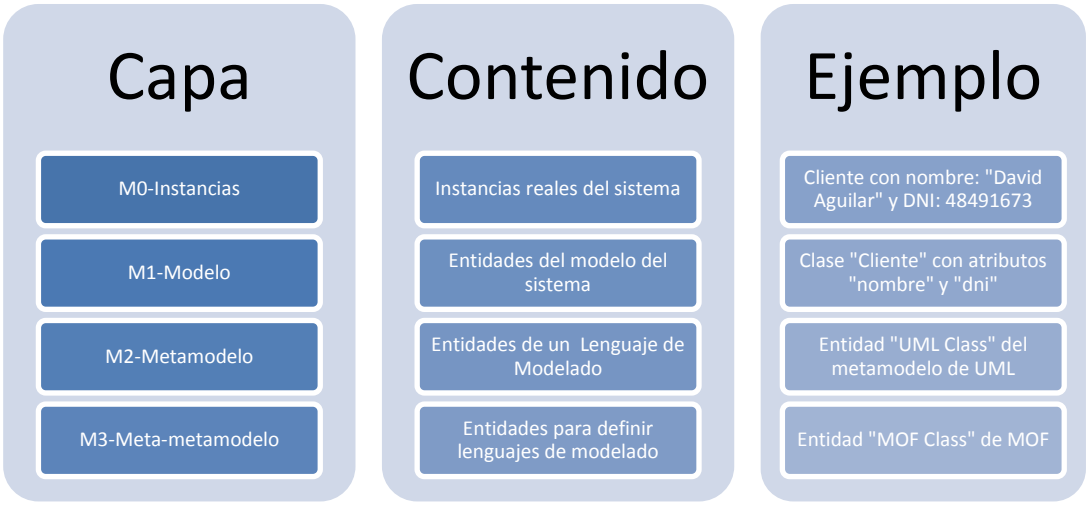


Figura 3.30. Entidades de la capa M0 del modelo de cuatro capas

Una Visión Completa de las Capas

Se podrían añadir más niveles a los ya descritos pero no sería muy útil. En lugar de definir una capa M4, el OMG estableció que, todos los elementos de M3, se puede definir con instancias de conceptos de M3, **lo que significa que MOF se define así mismo.**

Mostramos una tabla resumiendo las distintas capas del modelado:



La siguiente figura muestra un ejemplo completo donde se aprecia la relación existente entre las capas de modelado definidas por el **OMG**.

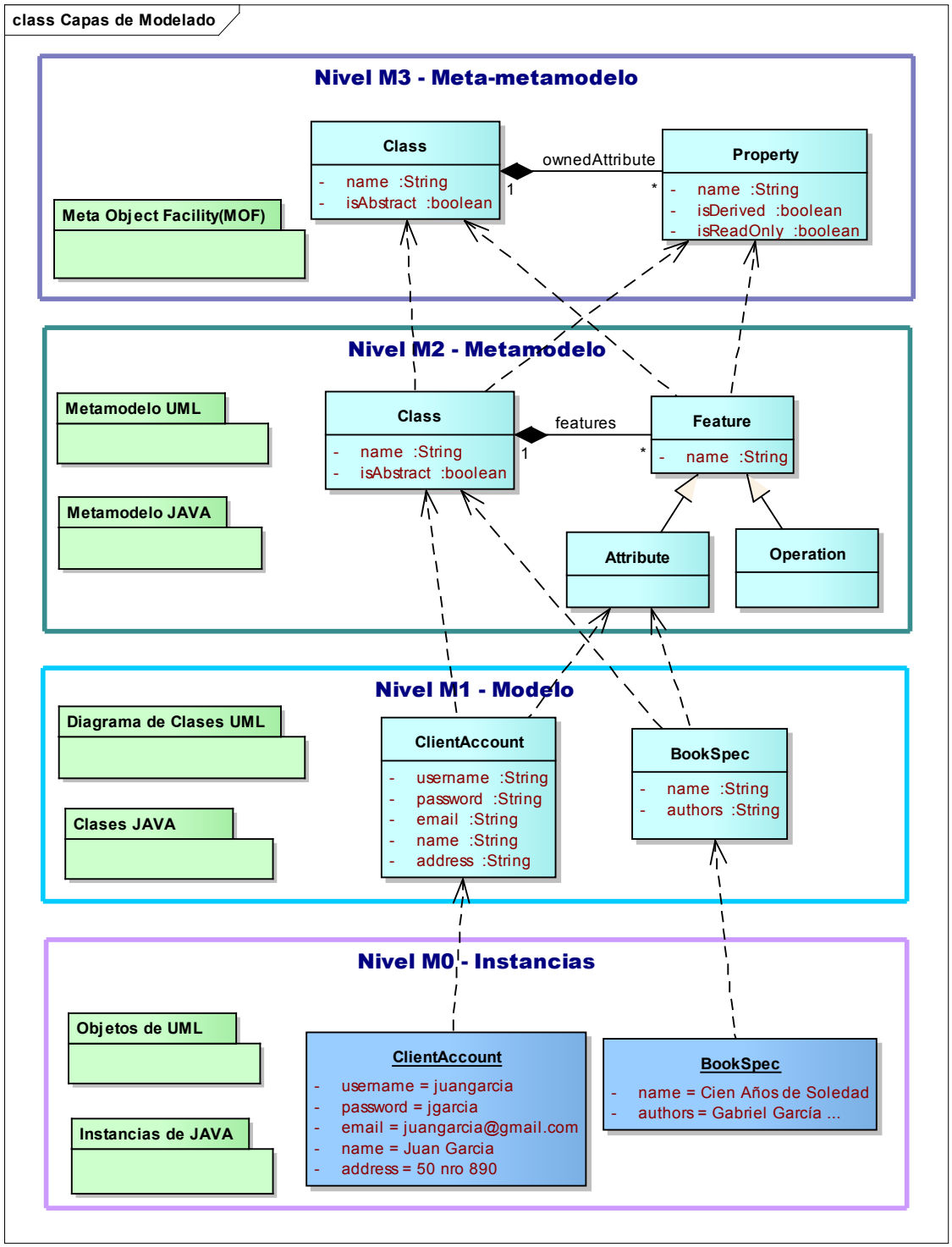


Figura 3.31. Relación entre las distintas capas de modelado.

3.4.13.1 Essential MOF (eMOF) y Complete MOF (cMOF)

MOF ha servido como base a diversos marcos de metamodelado, los cuáles, en la práctica, se limitan a implementar un subconjunto relevante de MOF. Actualmente el más influyente de estos marcos es Eclipse Modeling Framework (EMF) y su meta-metamodelo eCore. La implementación de EMF, a su vez, ha influido en el proceso de estandarización de MOF 2.0. Como consecuencia de dicha influencia, OMG identificó un subconjunto de MOF llamado essential MOF (EMOF), durante la estandarización de MOF 2.0 en 2003, como subconjunto suficiente para la mayoría de las realizaciones del estándar. Como consecuencia, el meta-metamodelo eCore de EMF es ahora conforme con el estándar EMOF del OMG.

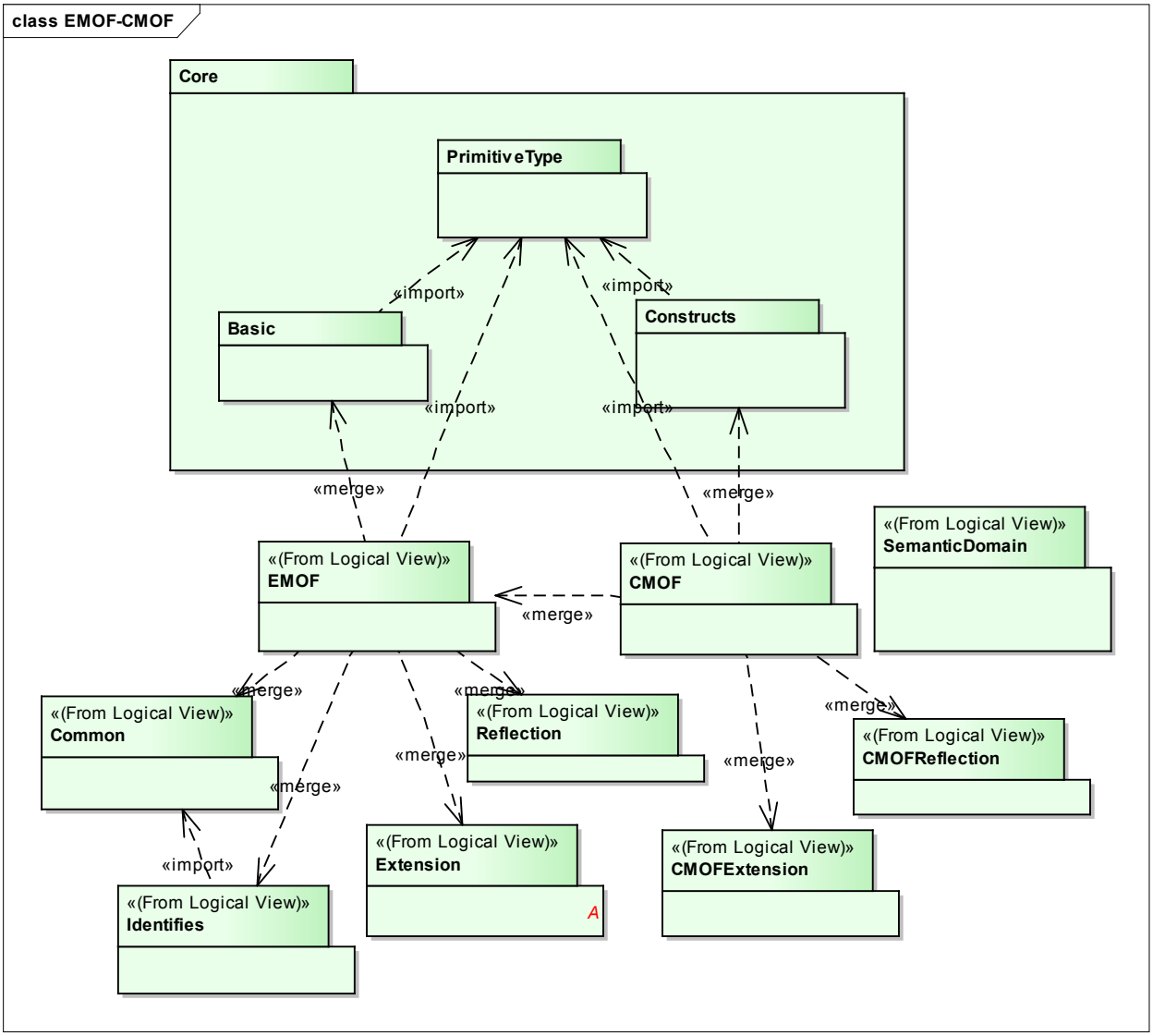


Figura 3.32. Paquetes de EMOF y CMOF.

El OMG también ha definido CMOF (Complete MOF), el cual se utiliza para la definición de metamodelos más complejos, como UML, y que además posee características introspectivas más poderosas, debido a que está construido a partir de EMOF, agregando nuevas funcionalidades, entre ellos paquetes de reflexión y los constructores del núcleo (core) de la InfrastructureLibrary (UML 2).

3.4.13.2 Importancia del Metamodelado en MDA

En primer lugar el metamodelado es importante en MDA porque actúa como mecanismo para **definir lenguajes de modelado**, de forma que su definición no sea ambigua. Esta no ambigüedad, permite que una herramienta pueda leer, escribir y entender modelos como los construidos con UML.

Por otro lado, las **reglas de transformación**, usadas para transformar un modelo de lenguaje A en otro modelo de lenguaje B, usan los metamodelos de lenguajes A y B para definir las transformaciones.

3.4.14 Transformaciones en MDA

- La transformación de modelos es una parte clave de **MDA.**
- La herramienta de transformación toma un **PIM** y lo transforma en uno o más **PSM**.
- Una segunda (o la misma herramienta) transforma el PSM a código.
- Poder adaptar las transformaciones a nuestras necesidades es uno de los puntos fuertes de **MDA.**

Una transformación (**mapping**) entre modelos está definida por una función de transformación compuesta de muchas reglas de transformación.

Objetivo: transformación automática de un modelo (más abstracto) a otro (más concreto)

- mayor portabilidad
- mayor eficiencia de desarrollo
- mayor calidad del software
- menor tiempo de desarrollo
- menor costo

Los detalles de transformación se separan de los modelos fuente y destino

- la transformación debería ser repetible para regenerar el modelo destino cuando cambia el modelo fuente.
- el modelo destino por definición está sincronizado con el modelo fuente.

3.4.14.1 Revisión de lenguajes de transformación de modelos

Como se ha visto en el apartado anterior, la transformación de modelos es una tecnología fundamental en los diversos enfoques del desarrollo de software dirigido por modelos. Esta tecnología permite caracterizar artefactos capaces de traducir automáticamente modelos que se ajustan a un determinado metamodelo origen, en modelos que se ajustan a un determinado metamodelo objetivo (transformaciones modelo a modelo). Así mismo, estas tecnologías permiten también realizar transformaciones entre modelos y otros formatos de representación (p.ej., entre modelos y formatos textuales, dando lugar a las denominadas transformaciones de modelo a texto y de texto a modelo).
En los últimos años se han definido una gran cantidad de lenguajes de transformación, cada uno con diferentes características. Sin embargo, aún es necesaria una mejor comprensión de la naturaleza de las transformaciones de modelos, así como seguir investigando para encontrar las propiedades deseables de un lenguaje de transformación de modelos.
Entre las propuestas existentes se encuentran los lenguajes más populares: QVT el estándar presentado por el OMG, ATL por ser una implementación muy cercana al estándar QVT que realiza transformaciones de Modelo a Modelo, MT model transformation language, el cual está implementado como un lenguaje especifico de dominio (DSL) embebido o interno, MOFScript que es un lenguaje de transformación de modelos a texto, y UMLX que es un lenguaje gráfico.

Analizaremos particularmente ATL y MOFScript que son los lenguajes que hemos utilizado para la implementación de nuestro trabajo; para la transformación de modelo a modelo y de modelo a texto respectivamente.

3.4.14.2 ATL(Atlas Transformation Language)

ATL es un lenguaje de transformación de modelos híbrido que permite, en su definición de transformaciones, especificar construcciones declarativas e imperativas.
ATL es compatible con los estándares de la OMG: es posible describir transformaciones modelo a modelo, modelos que deben conformar metamodelos definidos con MOF. Además, el lenguaje se basa en QVT. También permite definir pre y postcondiciones en OCL, resultando sumamente expresivo y de fácil escritura para sus usuarios.

En cuanto a las reglas de transformación, ATL define, tal y como se ha sugerido, un dominio (metamodelo) para el origen (*source*) y otro dominio para el destino (*target*), siendo ambos instancias de MOF y con direcciones *in* y *out* respectivamente. Los metamodelos *source* y *target* pueden tener iguales o diferentes dominios, pero, aun siendo iguales, ambos deben ser claramente identificados. Si bien las transformaciones son unidireccionales, ATL permite la definición de transformaciones bidireccionales mediante la implementación de dos transformaciones, una para cada dirección.
Como características particulares del lenguaje, podemos mencionar las estructuras que define este lenguaje. En primer lugar, la definición de transformaciones forman módulos (*modules*) que contienen las declaraciones iniciales y un número de *helpers* y reglas de transformación.

Los *helpers* son una estructura intermedia dentro de las transformaciones que permiten definir operaciones y tuplas OCL, componentes que facilitan la navegación, la modularización y la reutilización (vendrían a ser una especie de funciones auxiliares). Existe también una construcción llamada *called rule* que permite representar procedimientos que pueden contener argumentos y ser invocados por nombre.
La aplicación de las reglas se realiza de forma no determinista, por equiparación de patrones, no habiéndose provisto ninguna construcción o cláusula que permita aplicar en forma condicional las reglas. Cabe mencionar que la invocación de *called rules* es determinista. Esta invocación y la utilización de parámetros permiten soportar recursión. ATL además provee modularización basada en procedimientos (las ya citadas *called rules*), así como mecanismos de reutilización, ya que las reglas se pueden heredar y sus módulos

pueden incluir a otros módulos.
En cuanto a la trazabilidad, este lenguaje crea un enlace de trazabilidad (*traceability link*) con la ejecución de cada regla, que se guarda en el motor de la transformación. Este enlace relaciona tres elementos: la regla, los elementos originales (*source*) y los elementos creados (*target*).

3.4.14.2 .1 Justificación de la elección de ATL

Lo cierto es que no hay varios lenguajes para trabajar en el ámbito de las transformaciones M2M (model-to-model).
Al día de hoy se podría decir que el más utilizado y mejor posicionado es, sin lugar a dudas, ATL. El hecho que mejor lo demuestra es que es la solución estándar incorporada por el proyecto Eclipse M2M que es el que contiene todos los plugins necesarios para trabajar en temas de modelización conceptual sobre esa plataforma. ATL, contiene múltiples características que lo convierten en la mejor opción: realiza las conversiones a partir de metamodelos, permitiendo utilizar metamodelos propios diseñados, utiliza una sintaxis próxima a OCL que permite construir consultas complejas, es un proyecto activo, simple de entender y se puede afirmar que posee una documentación muy elaborada.

3.4.14.2 .2 Estructura general de la definición de Transformación ATL

- Diferencias entre modelos origen y destino
  - Los modelos origen solo se pueden leer
  - Los modelos destino solo pueden ser escritos

Tipos de Ficheros

- **ATL module:** Conjunto de reglas de transformación
- **Library:** Conjunto de funciones de ayuda que puede ser importado
- **Query:** Consultas que obtienen un valor simple de un modelo

Reglas de Transformación

- Una regla declarativa especifica:
  - Un patrón origen que se buscará en los modelos fuente,
  - Un patrón destino que se creará en el destino por cada regla que haga matching.
- Una regla imperativa es un procedimiento:
  - Se llama explícitamente (Called Rules)
  - Puede tomar argumentos
  - Puede contener:
    - Un patrón declarativo,
    - Un action block (secuencia de sentencias),
    - Ambos.

Patrón Origen

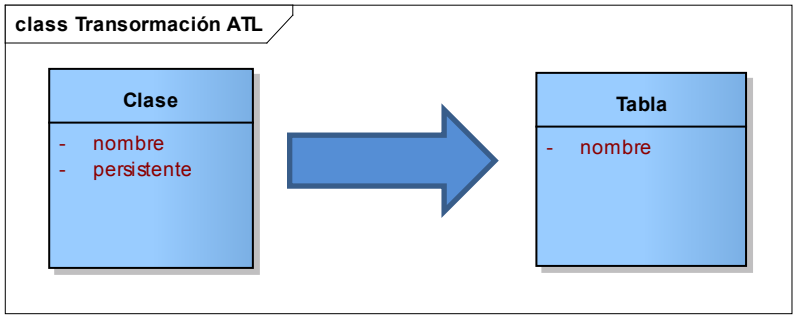
- El patrón origen está formado por:
  - Un conjunto etiquetado de elementos de los metamodelos origen
  - Una guarda
    - Expresión booleana a modo de filtro
- Se producirá un matching si se encuentran elementos en los modelos origen que:
  - Sean de los tipos especificados en el patrón,
  - Satisfagan la guarda.

Patrón Destino

- El patrón destino está compuesto de:
  - Elementos etiquetados del metamodelo destino
  - Para cada elemento una inicialización de sus propiedades
- Para cada coincidencia se aplica el patrón destino:
  - Se crean los elementos destino
  - Se inicializan sus propiedades
  - Primero, evaluando el valor de las expresiones
  - Después, asignando dicho valor a la propiedad indicada

Ejemplo de Regla

"Las Clases marcadas como persistentes, se transformarán en Tablas cuyo nombre será el nombre de la clase original"



3.33. Regla simple Transformación ATL.

- Metamodelos
  - Origen: Diagrama de Clases
  - Destino: Modelo Relacional
- Patrón origen
  - Elementos
    - c: Clase
  - Guarda
    - c.persistente
- Patrón destino
  - Elementos
    - t:Tabla
  - Inicialización
    - t.nombre ← c.nombre

Orden de Aplicación de las reglas

- Con las reglas declarativas permiten al desarrollador despreocuparse del orden de ejecución:
  - No se especifica el orden de
    - Aplicación de reglas
    - Inicialización de propiedades

- Sin embargo, se mantiene el determinismo
  - La ejecución de una regla no puede cambiar el modelo origen
    - → No puede alterar las coincidencias
  - El modelo destino no es navegable
    - → La creación de un elemento no puede afectar el valor de otro

Un Vistazo a la Sintaxis

Cabecera

module module_name;
create output_models [from|refines] input_models;

- Se especifica:
  - Nombre del modulo
- Modelos y Metamodelos implicados: OUT1: UML, OUT2: TraceModel…
- Tipo de transformación
  - Modelo nuevo
  - Refinamiento:En versiones recientes la palabra clave es refining

Librerías

uses lib_name;

- Permiten la modularización y reutilización de helpers

  Uses string;

Helpers

helper [**context** *context_type*] **def** : *helper_name*(*parameters*) : *return_type* = *exp*;

- Permiten extender el metamodelo con funciones y atributos derivados
  - Son atributos si no se usan parámetros
  - Las expresiones se definen en OCL
  - El contexto es opcional
- El modulo ATL es el contexto por defecto

- Ej: **helper context** Liga!Partido **def**: nombre : String = self.local.nombre +' -'+self.visitante.nombre;

  Sobre el objeto de tipo Partido define la función nombre que retorna un string con el nombre completo compuesto por el nombre de la variable 'local' y la variable 'visitante'.

Reglas

```
rule rule_name {
 from in_var : in_type [(condition)]
      [using {var1 : var_type1 = init_exp1;  ... varn : var_typen = init_expn;}]
 To
   out_var1 : out_type1 ( bindings1 )  ...
      out_varn : out_typen ( bindingsn  )
     [ do { statements } ] ….        }
```

Extras

- Depuración
  - La perspectiva de depuración permite ejecutar paso a paso las reglas
- Queries
  - Permiten calcular valores primitivos a partir de elementos del modelo.
  - Ejemplo
    - query caps = Book!Chapter.allInstances()->size().toString();
    - El método writeTo(fichero:String) permite escribir la cadena a un fichero.

3.4.14.3 MOFScript

MOFScript es un lenguaje de transformación de modelo a texto. Este lenguaje presta particular atención a la manipulación de texto y al control e impresión de salida de archivos. Es especialmente útil para realizar implementaciones dependientes de la plataforma, como, por ejemplo: la creación de código fuente en lenguajes como Java o C# a partir de modelos UML, o la generación automática del código de creación de una base de datos mediante SQL a partir de un Modelo de base de datos relacional. También puede ser utilizado para crear documentación o transformaciones de modelos a lenguajes textuales como XML o HTML.
MOFScript se basa en otros estándares de la OMG: es compatible con QVT y MOF para el modelo de entrada (el modelo objetivo –*target*- siempre es texto). Para las reglas de transformación se define un metamodelo de entrada (*source*) sobre el cual operarán las reglas. El target es generalmente un archivo de texto (o salida de texto por pantalla). Existe una implementación como un plugin para eclipse: MOFScript tools.
Las transformaciones son siempre unidireccionales, y no es posible definir pre y post condiciones para ellas. La separación sintáctica resulta clara por la misma definición de reglas, lo que hace al lenguaje legible. No provee estructuras intermedias, pero sí la parametrización necesaria para la invocación de reglas.

En cuanto a la aplicación de reglas, éstas se aplican en forma determinista y en orden secuencial. Se provee aplicación condicional e iteración de reglas. Las condiciones de aplicación se expresan en cláusulas when, como definición de guardas. La iteración se realiza mediante los bucles for each y while.
MOFScript no organiza las reglas en módulos propiamente dichos. Ahora bien, en la definición de una regla se pueden invocar a otras reglas, utilizando incluso parámetros, con lo cual las reglas en sí pueden entenderse como mecanismos básicos de modularización. También es posible definir jerarquías de transformaciones.
Finalmente, para trazabilidad, MOFScript define (pero no implementa aún) un conjunto de conceptos para relacionar elementos fuente con sus ubicaciones en los archivos de texto generados en el target.

Facilitando el desarrollo del trabajo

3.4.15 Eclipse Modeling Framework (Framework de modelado Eclipse, EMF)

Eclipse Modeling Framework (EMF) es un framework de modelado y facilidad de generación de código para construir herramientas y otras aplicaciones basadas en un modelo de datos estructurado. Desde una especificación del modelo descripta en XMI, EMF suministra herramientas y soporte runtime para producir un conjunto de clases Java para el modelo, un conjunto de clases Adapter que permiten visualización y edición basándose en comandos del modelo, y un editor básico. Los Modelos pueden ser especificados usando Anotación Java, documentos XML, o herramientas de modelado como Rational Rose, y después ser importados a EMF. Lo más importante de todo, EMF suministra las bases para la interoperabilidad con otras herramientas y aplicaciones basadas en EMF.

3.4.15.1 Eclipse

Figura 3.34. Logo Eclipse

La plataforma Eclipse consiste en un Entorno de Desarrollo Integrado (IDE, Integrated Development Environment) abierto y extensible. Un IDE es un programa compuesto por un conjunto de herramientas útiles para un desarrollador de software. Como elementos básicos, un IDE cuenta con un editor de código, un compilador/intérprete y un depurador. Eclipse sirve como IDE Java y cuenta con numerosas herramientas de desarrollo de software. También da soporte a otros lenguajes de programación, como son C/C++, Cobol, Fortran, PHP o Python. A la plataforma base de Eclipse se le pueden añadir extensiones (plugins) para extender la funcionalidad.

El término Eclipse además identifica a la comunidad de software libre para el desarrollo de la plataforma Eclipse. Este trabajo se divide en proyectos que tienen el objetivo de proporcionar una plataforma robusta, escalable y de calidad para el desarrollo de software con el IDE Eclipse.

3.4.15.2 Eclipse EMF

Eclipse EMF se puede utilizar para modelar el modelo de dominio. EMF tiene una distinción entre el meta-modelo y el modelo real. El meta-modelo describe la estructura del modelo. Un modelo es entonces la instancia de este meta-modelo. EMF proporciona un framework para almacenar y manipular la información del modelo, el valor predeterminado utiliza XMI (XML Metadata Interchange) que persiste la definición del modelo.

EMF permite crear meta-modelo a través de diferentes medios, por ejemplo XMI, anotaciones Java, UML o un esquema XML. En la implementación de nuestro trabajo utilizaremos las herramientas EMF, TOPCASE en este caso, directamente para crear un modelo de EMF.
Una vez especificado el EMF meta-modelo puede generar las clases de implementaciones Java correspondientes de este modelo. EMF proporciona la posibilidad de que el código generado se pueda extender manualmente de forma segura.

3.4.15.3 Meta Models - Ecore and Genmodel

Hemos dicho anteriormente que EMF tiene un meta-modelo. En realidad EMF se basa en dos meta-modelos, el Ecore y el modelo Genmodel. El metamodelo Ecore contiene la información sobre las clases definidas. El Genmodel contiene información adicional para el codegeneration, por ejemplo, la ruta y la información del archivo. El genmodel contiene también el parámetro de control para generar la codificación.

El modelo Ecore permite definir los diferentes elementos:

- **EClass**: representa una clase, con cero o más atributos y cero o más referencias.
- **EAttribute**: representa un atributo que tiene un nombre y un tipo.
- **Ereferencia**: representa un extremo de una asociación entre dos clases. Tiene bandera para indicar si representa una contención y una clase de referencia al que se apunta.
- **EDataType**: representa el tipo de un atributo, por ejemplo, int, float o java.util.Date.

El modelo Ecore muestra un objeto raíz que representa todo el modelo. Este modelo tiene hijos que representa los paquetes, cuyos hijos representa las clases, mientras que los hijos de las clases representan los atributos de estas clases.
Con EMF hacer el modelo de dominio, ayuda a proporcionar una visibilidad clara del modelado. También proporciona funcionalidad de notificación de cambio en el modelo en caso que exista.

Otra ventaja es que se puede generar el código Java a partir del modelo en cualquier punto en el tiempo.

Capítulo 4

Implementación de la Solución

4.1 Objetivo

El objetivo de este trabajo es la realización de una herramienta de software que permita definir modelos de manera gráfica utilizando **UML 2 y UML 2 Service,** donde este último es un nuevo perfil de UML 2 que se desarrollará para modelar la arquitectura de servicios del Framework **Spring**, y derivar código fuente a partir de los mismos.

4.2 Caso de Estudio

Nos basaremos en el ejemplo sencillo y clásico de tienda de libros virtual, para ejemplificar y sobre él iremos explicando el diseño de los módulos, interfaces y clases esperadas, sus relaciones y los archivos necesario de configuración.

Mostraremos cómo quedarán finalmente definidos los objetos, archivos y la estructura de carpetas, todos para luego explicar cómo hemos agilizado el diseño y parte del desarrollo con la ayuda de la tecnología **MDE**, **"Ingeniería de Software Dirigida por Modelos"**, que hemos investigado como base de este trabajo, así como también promoviendo el "buen" diseño y uso que nos ofrecen las arquitecturas elegidas. Tenerlas al alcance de la mano, no nos asegura aprovechar su potencial si no hacemos el uso correcto.

4.2.1 Modelo estructural

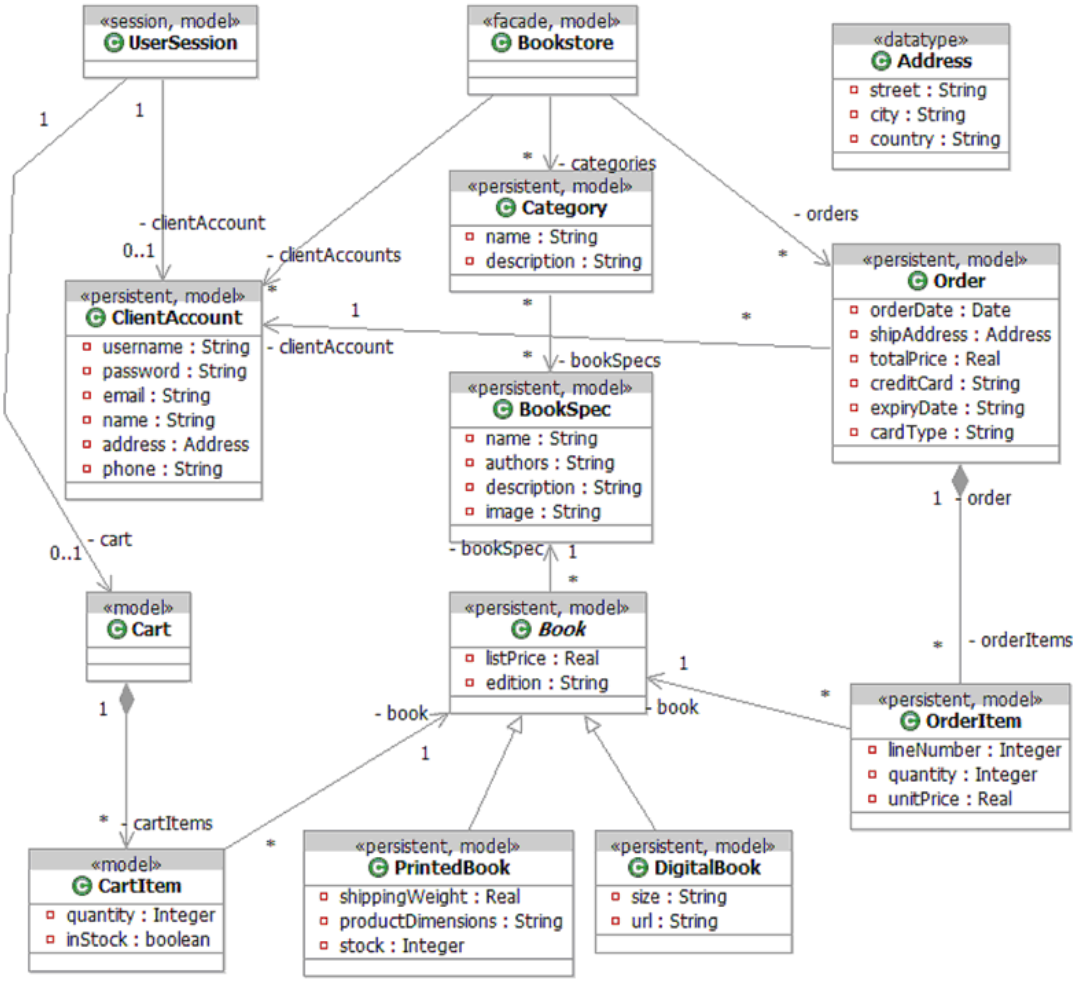


Figura 4.1. Diagrama de Clases Bookstore.

La figura 4.1 exhibe el diagrama de clases del sistema bookstore. Como puede verse en la figura, un bookstore tiene información acerca de sus clientes (clientAccount) y de los libros que tiene en stock. Cada libro tiene una especificación de sus características (BookSpec) y puede presentarse en dos versiones: digital (DigitalBook) o impreso (PrintedBook). Los libros se organizan en categorías (Category). Cada orden de compra (Order) está formada por varios ítems (OrderItem), donde se registra el ítem comprado, la cantidad y el precio pagado. Un usuario puede ir acumulando ítems en un carrito de compras (Cart) y posteriormente efectivizar la compra de estos ítems.

Como vemos, este es nuestro modelo de objetos, en el cual debemos identificar cuáles serán las clases persistentes, las mismas identifican los objetos cuya información es necesaria persistir en un repositorio de base de datos y para ellas definir los **DAOs** que se encargarán de la administración de los datos, del proceso de recuperación, modificación y guardado de los mismos. Utilizando para ello la implementación de consultas en el lenguaje de consultas, **Hibernate Query Language (HQL),** propuesto por la arquitectura de **Hibernate**.

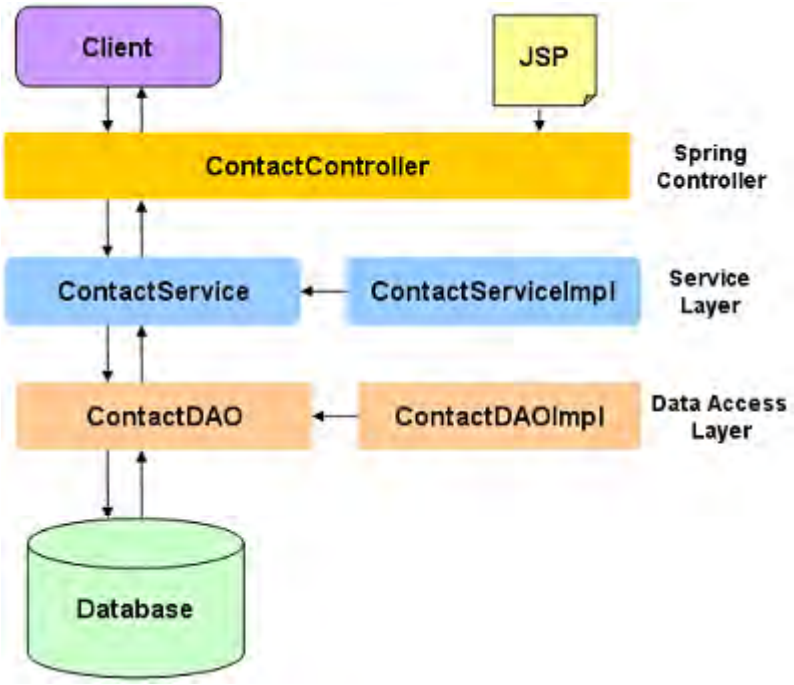
Básicamente todos los elementos del modelo son objetos del modelo de negocio, salvo "address" que está definido como un tipo de datos, pero no todos van a ser objetos persistentes y explicaremos porqué. Los datos de "libros" que ofrece la librería deben estar persistidos para tener el estado almacenado y el conocimiento de los mismos en tiempo y forma, sus "categorías", los "clientes" registrados en el sistema, junto con sus "ordenes de compras" realizadas y el "detalle" de las mismas. No obstante para el usuario, "userSession" que se encuentra en este momento usando el sistema, el cual se mantiene una sesión volátil (si se desconecta del sistema finaliza la misma.), se mantienen datos monitoreando su actividad, no es un dato persistente, también así el "carrito de compras" con sus "CartItems" no son datos persistentes, se mantiene en la sesión y se va llenando hasta que se hace eficiente la compra.

Identificamos los objetos del modelo con "etiquetado" <<model>> y los persistentes con el <<persistent>>, que luego va a definir cada uno un "perfil" para MDA, que explicaremos más adelante cuál es su significado y utilidad.

Ahora para el manejo de los objetos persistentes vamos a definir los DAOs correspondientes, y dar un ejemplo de los archivos "xml" que usa **hibernate** para el mapeo de los mismos. Daremos un ejemplo sencillo ya que no vamos a profundizar en cada tecnología, porque son varias y cada una requiere un análisis intenso, explicaremos los conocimientos básicos como para entender la idea general y poder mostrar en detalle el potencial que nos ofrece la arquitectura **MDE** que es el objetivo de este trabajo.

4.2.2 Arquitectura de la aplicación

Para nuestra aplicación web vamos a tener una arquitectura por capas. Accederemos a la base de datos a través de una capa (DAO, esta es la capa que utiliza Hibernate) y la llamaremos mediante una capa de servicios.



Definiendo Servicios, DAOs, Filtros y archivos de mapeo.

4.2.2.1 La capa de Negocio

Toda petición de cliente se convierte a una petición de servicio de negocio y se le pasa al servicio de negocio apropiado. Cada objeto de servicio de negocio realiza la lógica de negocio necesaria y hace uso del **DAO** apropiado para acceder al almacén de datos.

En este ejemplo vamos a definir los siguientes Servicios:

- **Services**

    - **ClientAccountService**: para la administración lógica de datos de los clientes representados por los objetos “ClientAccount”.

    - **BookSpecService**: para la administración lógica de datos de las características de libros representados por los objetos “BookSpec”.

    - **CategoryService**: para la administración lógica de datos de las categorías de libros representados por los objetos “Category”.

    - **BookService**: para la administración lógica de datos de los libros representados por los objetos “Book”.

    - **OrderService**: para la administración lógica de datos de las ordenes de compras y sus ítems representadas por los objetos “Order” y “OrderItem”, siendo OrderItems objetos que componen al objeto Order.

En nuestros proyectos orientados a servicios definimos una clase abstracta **AbstractService**, donde se definen métodos genéricos de los servicios, de la cual todos los servicios van a extender. Además para cada servicio particular, se define una interface, quien determina y publica los métodos que el servicio puede resolver. Por lo pronto cada servicio implementa la interface correspondiente.

Por una cuestión de claridad, la nomenclación de los mismos sigue la siguiente regla:

> **La clase del servicio determinado por el nombre “name” se llamará “Name_ServiceImpl” y la interface que implementa “IService_Name”**

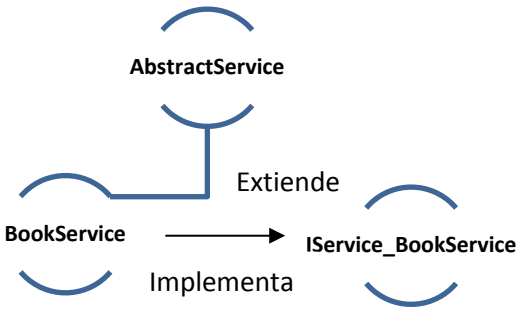
Ejemplificamos sobre la clase BookService:

La clase **BookService** encapsula los métodos para operar sobre objetos **Book** administrando la información de los mismos en el sistema incluyendo almacenamiento, actualización, borrado y recuperación de ejemplares de **Book**. Definimos para la clase “BookService”, la interface “IService_BookService” correspondiente.

- Aquí puede ver la interface **IService_BookService:**

```
public interface IService_BookService
{

	public Book get_Book (int id_Book);
	public Book get_Book (Book_ParametroFilter parameterFilterBook);
	public List getAll_Book ();
	public List getAll_Book (Book_ParametroFilter parameterFilterBook);
	public List getAll_BookByBookSpec (BookSpec_ParametroFilter
parameterFilterBookSpec);

}
```

Como explicamos, la clase **BookService_ServiceImpl** implementa la interface **IService_BookService**, y extiende la clase **AbstractService**, por cual implementa todos los métodos determinados por la interface y la clase **AbstractService**:



- Aquí puede ver la clase **BookService_ServiceImpl**:

```
public class BookService_ServiceImpl extends AbstractService implements
IService_BookService {
	private BookDAO_DaoImpl bookDAO_DaoImpl;
	private BookSpecService_ServiceImpl bookSpecService_ServiceImpl;

	public Book get_Book (int id_Book){
		return null;	}

	public Book get_Book (Book_ParametroFilter parameterFilterBook){
		return null;	}

	public List<Object> getAll_Book (){
		return null;	}
```

```
	public List<Object> getAll_Book (Book_ParametroFilter parameterFilterBook){
		return null;	}

	public List<Object> getAll_BookByBookSpec (BookSpec_ParametroFilter
parameterFilterBookSpec){
		return null;	}

	private void setBookDAO_DaoImpl (BookDAO_DaoImpl BookDAO_DaoImpl){}

	private BookDAO_DaoImpl getBookDAO_DaoImpl (){
		return null;	}

	private void setBookSpecService_ServiceImpl (BookSpecService_ServiceImpl
BookSpecService_ServiceImpl){ }

	private BookSpecService_ServiceImpl getBookSpecService_ServiceImpl (){
		return null;	}


}
```

Como ven la clase **BookService_ServiceImpl** utiliza el DAO **bookDAO_DaoImpl** el cual administra el repositorio de datos de las clases **Book**, **DigitalBook y PrintedBook** que representan los libros.

Aclaración: Cabe destacar que en la capa de servicios, solo definimos los métodos básicos para cada servicio, para la administración de los objetos determinados por sus variables. Siendo nuestro objetivo obtener un esqueleto de las clases e interfaces involucradas en la arquitectura de servicios para un modelo de objetos determinado, para lograr una optimización en el diseño y desarrollo. Pero los servicios pueden resolver métodos de negocio aún más complejos que serán agregados según los requerimientos y complejidad de cada sistema.

4.2.2.1 La Capa de Datos

Toda petición de cliente se pasa al servicio de negocio apropiado para su procesamiento. Un servicio de negocio realiza la lógica de negocio necesaria y hace uso del **DAO** apropiado para acceder al almacén de datos. Cada DAO realiza las interacciones necesarias con Hibernate para poder actuar sobre un almacén de datos dado. La interface **Iname_DAO** define los métodos que cada clase DAO, **name_DaoImpl** debe implementar, siendo name en cada caso el nombre definido para la clase del dao correspondiente:

En este ejemplo nosotras vamos a definir los siguientes DAOs para el sistema de venta de libros:

- **DAOs**
  - **ClientAccountDAO**: para la administración de datos de los clientes representados por los objetos "ClientAccount".
  - **CategotyDAO**: para la administración de datos de los objetos "Category"
  - **BookSpecDAO**: para la administración de las características de los libros representados por los objetos "Book".
  - **BookDAO**: para la administración de datos de los libros representados por los objetos "Book".
  - **OrderDAO**: para la administración de datos de las órdenes de compras y sus ítems representadas por los objetos "Order" y "OrderItem".

- Siguiendo con el ejemplo, mostramos la interface **IBookDAO_Dao**

```
public interface IBookDAO_DAO
{
      public Book get_Book (int id_Book);
      public Book get_Book (Book_ParametroFilter parameterFilterBook);
      public List<Object> getAll_Book ();
      public List<Object> getAll_Book (Book_ParametroFilter parameterFilterBook);
      public List<Object> getAll_BookByBookSpec (BookSpec_ParametroFilter
parameterFilterBookSpec);}
```

Y a continuación la clase **BookDAO_DaoImpl** que implementa la interface **BookDAO_Dao**

```
public class BookDAO_DaoImpl extends AbstractDAOImpl implements IBookDAO_DAO {

	public Book get_Book (int id_Book){
		return null; }

	public Book get_Book (Book_ParametroFilter parameterFilterBook){
		return null; }

	public List getAll_Book (){
		return null; }

	public List getAll_Book (Book_ParametroFilter parameterFilterBook){
		return null; }

	public List getAll_BookByBookSpec (BookSpec_ParametroFilter
parameterFilterBookSpec){
		return null; }

	public void save (Object entity){   }

	public void update (Object entity){ }

	public void delete (Object entity){ }

}
```

El DAO **BookDAO_DaoImpl** administra los datos almacenados de las clases **Book, DigitalBook, PrintedBook**.

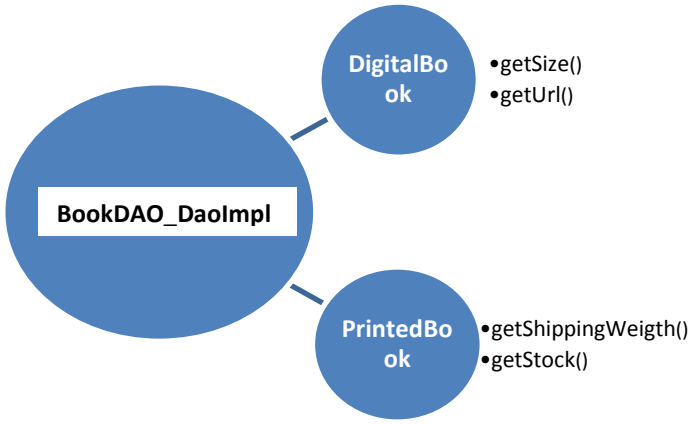


Figura 4.2. Relación DAOs- Objetos modelo.

- La clase del modelo Book:

```
public class Book implements AbstractPersistentObject {
      private int listPrice;
      private String edition;
      public BookSpec bookSpec;

      public int getListPrice (){
            return 0;   }

      public String getEdition (){
            return null;      }

      public void setListPrice (int listPrice){ }

      public void setEdition (String edition){}

      public BookSpec getBookSpec (){
            return null;      }

      public void setBookSpec (BookSpec bookSpec){}

      public int getId (){
            return 0;   }

      public void setId (int id){}

}
```

La clase **Book** representa un libro dado y se configura para Hibernate y su mapeo correspondiente en el fichero **Book.hbm.xml**:

```
<?xml version="1.0"?>
<hibernate-mapping
package="Uml_ventaLibrosProject.src.Model.uml_ventaLibros.Pk_adminVentaLibros.Pk_adminV
entaLibros_server.hibernate.model.Pk_model">
   <class name="Book" table="Book">
       <id name="id" type="long" unsaved-value="0">
          <column name="idBook" sql-type="BIGINT" not-null="true"/>
          <generator class="native"/>
       </id>
       <property name="listPrice" column="LISTPRICE" not-null="false" />
       <property name="edition" column="EDITION" not-null="false" />
       <property name="bookSpec" column="BOOKSPEC" not-null="false" />
       <joined-subclass name="PrintedBook" table="PrintedBook">
          <key column="idBook"/>
          <property name="shippingWeigth" column="SHIPPINGWEIGTH" not-null="false" />
          <property name="productDimensions" column="PRODUCTDIMENSIONS" not-
null="false" />
          <property name="stock" column="STOCK" not-null="false" /> </joined-subclass>
        <joined-subclass name="DigitalBook" table="DigitalBook">
          <key column="idBook"/>
          <property name="size" column="SIZE" not-null="false" />
          <property name="url" column="URL" not-null="false" />
          </joined-subclass>  </class>   </hibernate-mapping>
```

4.2.2.2.1 Una simple descripción del archivo de mapeo:

La línea `<class name="Book" table="Book">` indica que la clase “Book” se mapea a la tabla con el nombre “Book”.
La línea
```
<id name="id" type="long" unsaved-value="0">
        <column name="idBook" sql-type="BIGINT" not-null="true"/>
        <generator class="native"/>
    </id>
```
Determina que la tabla tendrá un campo identificador de tipo long llamado idBook.

La línea `<property name="name" column="NAME" not-null="false" />` indica que la propiedad “name” del objeto se mapeara a la columna "NAME" de la tabla Book, y asi se indica correspondencia a la tabla para todas las propiedades del objeto.

La línea
```
<joined-subclass name="DigitalBook" table="DigitalBook">
    <key column="idBook"/>
    <property name="size" column="SIZE" not-null="false" />
    <property name="url" column="URL" not-null="false" />
</joined-subclass>
```
indica que se mapeará la subclase de Book "DigitalBook" a la tabla correspondiente "DigitalBook". De la misma manera se mapea la subclase "PrintedBook".

Como podemos observar algunos métodos en la clase **BookDAO_DaoImpl** reciben como parámetros objetos de la clase Filter, como por ejemplo el siguiente método:

```
public Book get_Book (Book_ParametroFilter parameterFilterBook)
```

Explicamos ahora cómo se utilizan los filtros en las consultas y cómo aportan flexibilidad a las búsquedas.

4.2.2.2.2 El Lenguaje de Consultas de Hibernate

Hibernate ofrece un lenguaje de consultas que agrupa un potente y flexible mecanismo de consulta, almacenamiento, actualización y recuperación de objetos desde una base de datos. Este lenguaje, el **Hibernate Query Language (HQL),** es una extensión orientada a objetos de SQL. HQL permite acceder a los datos de varias formas, incluyendo consultas orientadas a objetos, como en el método find() del siguiente ejemplo:

```
List users = session.find("from Book as u where u.name = ?",
                "Cien Años de Soledad",
                Hibernate.STRING );
```

Se pueden construir consultas dinámicas utilizando la API criteria de Hibernate:

```
Criteria criteria = session.createCriteria(Book.class);
criteria.add(Expression.eq("name", " Cien Años de Soledad "));
criteria.setMaxResults(20);
List users = criteria.list();
```

Si se prefiere, se puede utilizar SQL o expresar una consulta SQL, utilizando createSQLQuery():

```
List books=  session.createSQLQuery("SELECT { book.*} FROM Book AS { book}",
       "book",Book.class).list();
```

Cuando se devuelven muchos objetos desde una consulta, los objetos serán cargados según se necesite utilizando uno de los métodos iterate(). Los métodos iterate() ofrecen un mejor rendimiento porque cargan los objetos bajo demanda:

```
Iterator iter =    session.iterate("from book as b where b.name= Literatura
       Romántica ");
while (iter.hasNext()) {
   book bookS = (book)iter.next();
   // process the book object here
}
```

Los objetos Filtros determinan una forma dinámica y encapsulada de crear criterios. Un ejemplo en este caso del método get_Book:

```
  public Book get_BookSpec (Book_ParametroFilter parameterFilterBook)
   {
      Book_Filter filtro = new Book_Filter (parameterFilterBook);
            List books= (List)
                    this.getHibernateTemplate().findByCriteria(filtro.crearCriteria());
            if (books.size() == 0){
                  return null;
            }else {
                  return (Book) books.get(0);
            }

   }
```

- La clase **Book_Filter:**

```
public class Book_Filter extends AbstractFilter{

	public Book_Filter (Book_ParametroFilter parametro){
		super(Book.class, parametro);
	}
	protected Book_ParametroFilter getParametro() {
		return (Book_ParametroFilter) this.parametro;
	}
	protected Book_Filter (Class claseRetorno) {
		super(claseRetorno);
	}
	public DetachedCriteria crearCriteria() {
		this.createDetachedCriteria();
		this.createFilterName();
		return this.getCriteria();                              }

	public void createFilter_listPrice (){
	}

	public void createFilter_edition (){
		if(this.getParametro().getEdition() != null &&
		!this.getParametro().getEdition ().equals("")){
                                if (this.getParametro().getIgualdad())

                        this.getCriteria().add(Restrictions.like("edition",
                    this.getParametro().getEdition (),MatchMode.EXACT));
                                else

                        this.getCriteria().add(Restrictions.like("edition",
                    this.getParametro().getEdition (),MatchMode.ANYWHERE));     }}      }
```

La clase **Book_ParametroFilter** tiene seteadas variables que determinan el filtro de búsqueda, en este caso el método **createFilter_edition()** realiza la búsqueda de **Book** por el parámetro **edition** que dependiendo si el parámetro está seteado como igualdad para la búsqueda, busca el nombre exacto, en caso contrario busca la palabra y el resultado puede devolver más de un objeto.

Estos filtros se pueden realizar por cualquier propiedad del objeto.
Bien, hemos definido las clases de **servicios**, **daos**, sus **interfaces**, **filtros** y **archivos de mapeo** de los objetos persistentes de nuestro sistema ejemplificando sobre un servicio particular.

Nos estaría faltando para finalizar, nuestros archivos de configuración definiendo al motor de Spring las clases que van a ser beans administrables en el entorno.
Aquí un bosquejo simple Project-context.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
      <bean id="project.app" lazy-init="false"
class="org.springframework.context.support.ClassPathXmlApplicationContext">
            <constructor-arg>
                  <list>
                        <value>classpath:spring-hibernate-mysql.xml</value>
                        <value>classpath:spring-daos.xml</value>
                        <value>classpath:spring-servicios.xml</value>
                  </list>
            </constructor-arg>  </bean> </beans>
```

Indicamos los archivos xml que se van a usar, en este caso:

classpath:spring-hibernate-mysql.xml: Determina la conexión a la base de datos e indica el listado de archivos ".hbm.xml" indicando la ruta del mismo para acceder a la descripción de mapeo de cada objeto persistente.

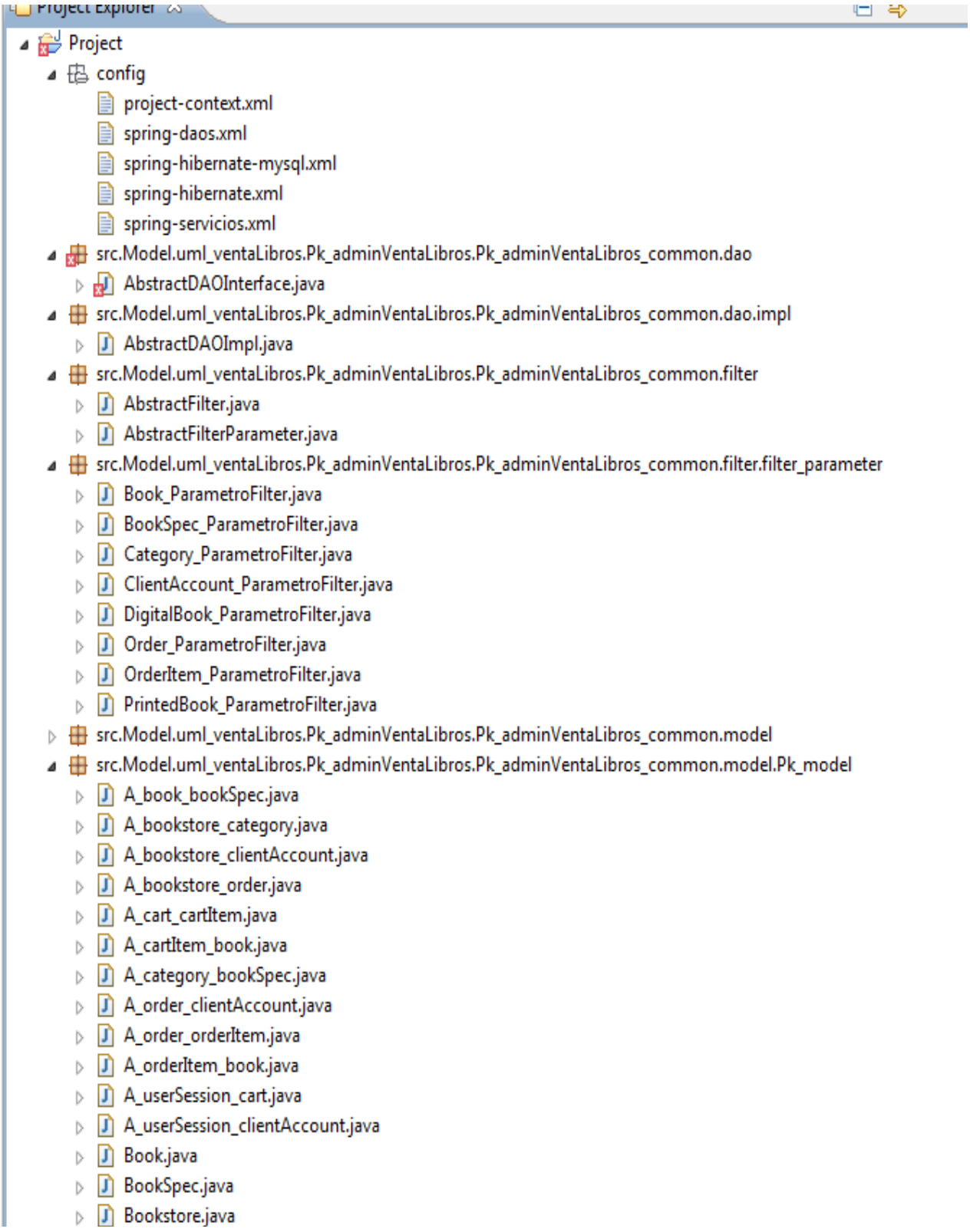
classpath:spring-daos.xml : Indicando las clases Daos del sistema y su ubicación

classpath:spring-servicios.xml: Indicando las clases Services del sistema y su ubicación.

No detallaremos cada archivo aquí, ya que son fáciles de entender y bien se pueden observar en el proyecto práctico.

Ahora sí, estamos en condiciones de mostrar el árbol de archivos del sistema de venta de libros, el cual quedaría de la siguiente manera:

4.2.3 Esqueleto estructural, árbol de archivos del sistema de venta de libros



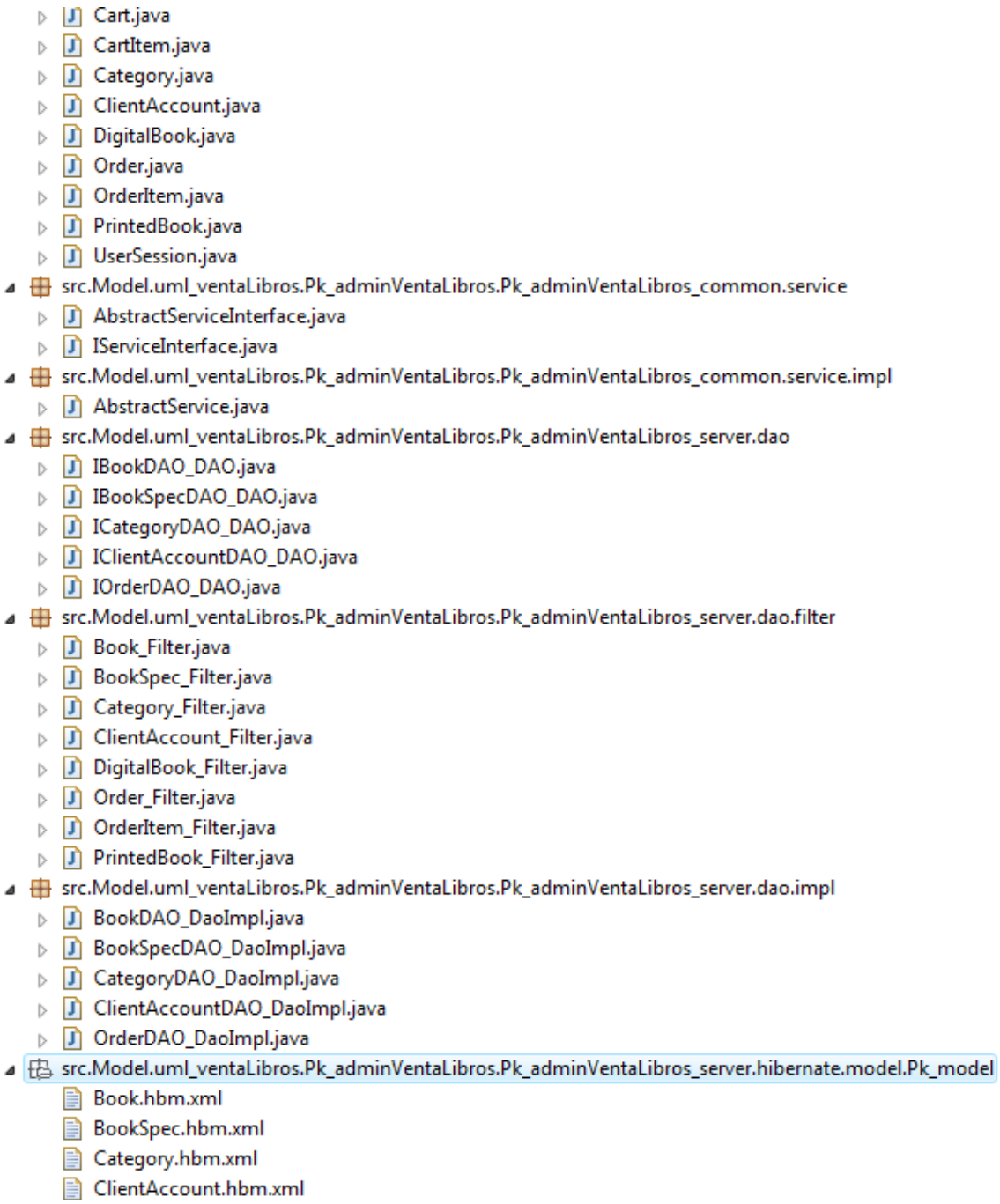


Figura 4.3. Esqueleto del árbol

Como vemos son muchos los archivos que se derivan del modelo de objetos completando el sistema orientado a servicios. Vamos a explicar ahora paso a paso cómo se diseñó el framework para agilizar el desarrollo y codificación de estos sistemas orientados a servicios con Spring enfatizando la correctitud y el buen diseño de los mismos.

4.3 Manos a la obra

- ### Definición de pasos a seguir

  - **4.3.1 Desarrollo del Plugin Profile UService**
    - **4.3.1.1 Definiendo el modelo de objetos enriquecido por el perfil UService**
    - **4.3.1.2 Creamos el Plugin Profile UService**
    - **4.3.1.3 Definición gráfica del modelo de objetos enrriquecido por el perfil UService**

  - **4.3.2 Desarrollo de reglas OCL para la validación de los modelos**
    - **4.3.2.1 Definiendo las reglas**
    - **4.3.2.2 Validando el modelo de objetos**

  - **4.3.3 Desarrollo de transformación del PIM al PSM con ATL**
    - **4.3.3.1 Definiendo metamodelos de dominio Source y Target**
      - **4.1.3.1.1 Creamos el metamodelo JAVA.ecore**

    - **4.3.3.2 Implementando la transformación del PIM al PSM con ATL**
      - **4.3.3.2.1 Definimos objetivo y comportamiento de la transformación**
      - **4.3.3.2.2 Detalles de implementación de la transformación ATL**
      - **4.3.3.2.3 Probamos la transformación y mostramos el resultado obtenido**

  - **4.3.4 Desarrollo de transformación de modelo a texto, derivando a código Java**
    - **4.3.4.1 Probamos la transformación a código verificando el esqueleto del sistema obtenido.**

  - **4.3.5 Desarrollo de plugin de ejecución completa de las dos transformaciones**
    - **4.3.5.1 Probamos la automatización completa del sistema, de manera transparente al usuario se ejecutan las dos transformaciones y observamos los resultados finales.**

4.3.1 Desarrollo del Plugin Profile UService para Eclipse

4.3.1.1 Definiendo el modelo de objetos enriquecido por el perfil UService

En primer lugar vamos a definir el perfil **UServices** que nos servirá para poder enriquecer nuestro modelo de objetos con la descripción necesaria para identificar las clases claves, que determinarán un comportamiento particular en nuestra transformación ATL.

Para ello vamos a pensar los siguientes estereotipos importantes:

- **<<PackagePrincipal>>:**
  - **Se aplica a la metaclase "package":** identifica al paquete principal de nuestro sistema.

- **<<PackageRoot>>:**
  - **Se aplica a la metaclase "package":** identifica al paquete root que determinara el namespace principal de nuestro sistema.

- **<<PackageModel>>:**
  - **Se aplica a la metaclase "package":** identifica al paquete donde se encuentran las clases del modelo de negocio.
  - **Constraints:**

    ✓ Los PackageModel tienen que estar dentro de un PackagePrincipal o de otro PackageModel
    ✓ Todos los elementos de un PackageModel tienen que ser BussinessModelElementcontext Package inv elementosBussinessModelElement:

- **<< BussinesModelElement >>:**
  - **Se aplica a la metaclase "Class":** identifica que un objeto responde al modelo de negocio.
  - **Constraints:**

    ✓ Los BussinessModelElement tienen que estar dentro de un PackageModel

- **<< PersistentClass >>:**
  - **Se aplica a la metaclase "Class":** identifica que un objeto deberá persistir en el repositorio de datos.
  - **Constraints:**

  ✓ Los PersistentClass deben ser BussinessModelElement

- **<< Service >>:**
  - **Se aplica a la metaclase "Class":** identifica que un objeto es un servicio y obtendrá el comportamiento correspondiente a los mismos.
  - **Constraints:**

  ✓ Los Services deben estar relacionados únicamente con objetos DAOs o Services
  ✓ Las relaciones de los Services deben ser de tipo dependencia "Usage"
  ✓ Los servicios deben estar relacionados al menos con un objeto

- **<< DAO >>:**
  - **Se aplica a la metaclase "Class":** identifica que un objeto es un DAO obtendrá el comportamiento correspondiente a los mismos.
  - **Constraints:**

  ✓ Los Daos deben estar relacionados únicamente con objetos PersistentClass
  ✓ Las relaciones de los Daos deben ser de tipo dependencia "Usage"
  ✓ Los Daos deben estar relacionados al menos con un objeto

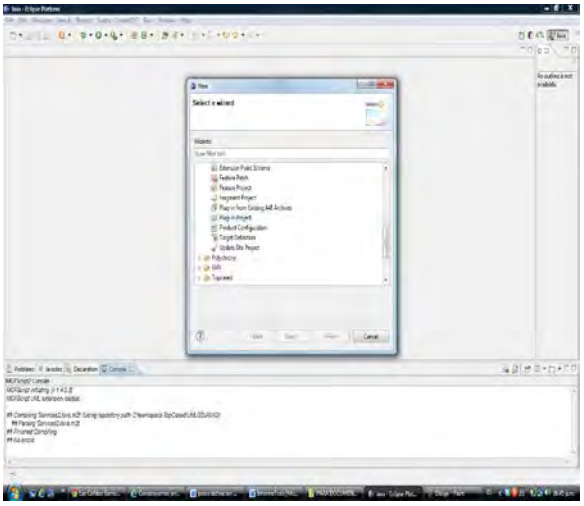
4.3.1.2 Creamos el Plugin Profile UService

Ahora vamos a crear el plugin para definir nuestro perfil UService, diseñado con estos estereotipos para luego poder extender el modelo UML.

Primeros pasos:

1_ Crear plugin:

- Parados sobre la plataforma Eclipse, botón derecho -> new -> others -> plug-in Development -> plug-in Project.



2_ Creamos el uml que define nuestros estereotipos y las metaclases asociadas. Lo hacemos con la ayuda de la herramienta TopCased:

- Para crear el uml : new -> others -> Topcased -> UML Model whit TOPCASED
- Setear en “From template model” -> Template: Profile Template.
- Crear estereotipos:
  - Crear una mataclase, y seleccionar en el panel de propiedades el tipo de la metaclase, ej CLASS.
  - Crear un stereotype, luego indicar con la relación “Extension” a qué tipo de metaclase se permite aplicarlo luego en los modelos.

- El uml generado:

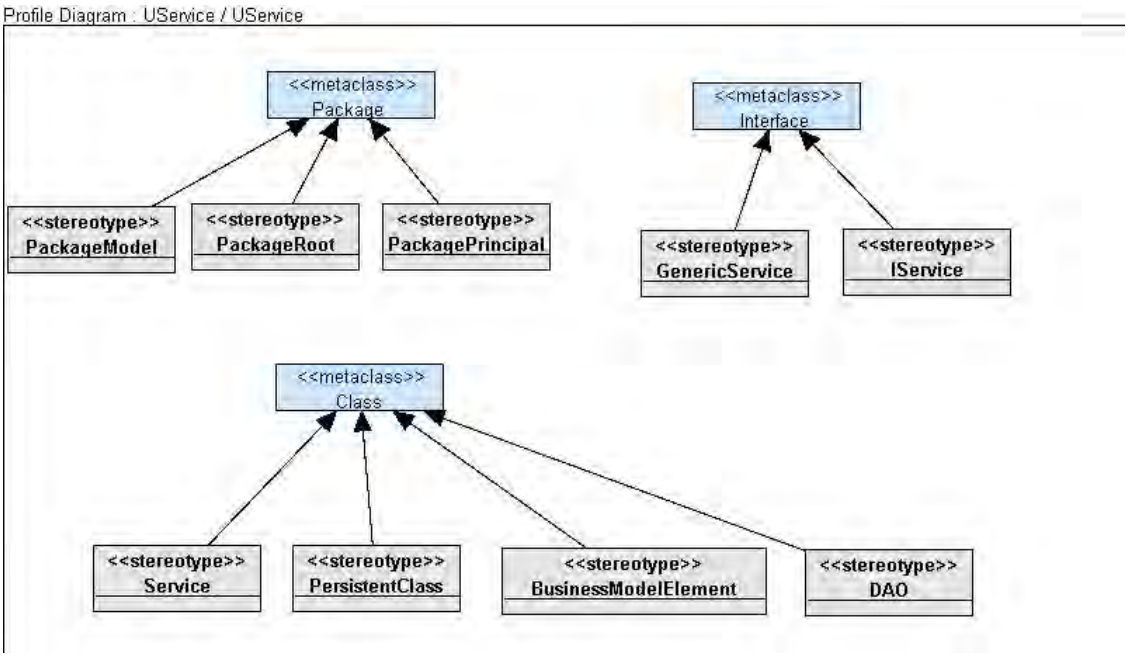


Figura 4.4. Modelo de Estereotipos. Perfil UService

2_ Crear archivo xml:

- Para crear el plugin.xml: new -> File y en file name: plugin.xml

- Plugin.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
 <extension
    point="org.eclipse.emf.ecore.uri_mapping">
  <mapping
    source="pathmap://UML2_PROFILES/"
    target="platform:/build/PlugginProjectNuevo/">
  </mapping>
 </extension>
```

- ### 4.3.1.3 Definición grafica del modelo de objetos enriquecido por el perfil UService.

El modelo independiente de plataforma (PIM)

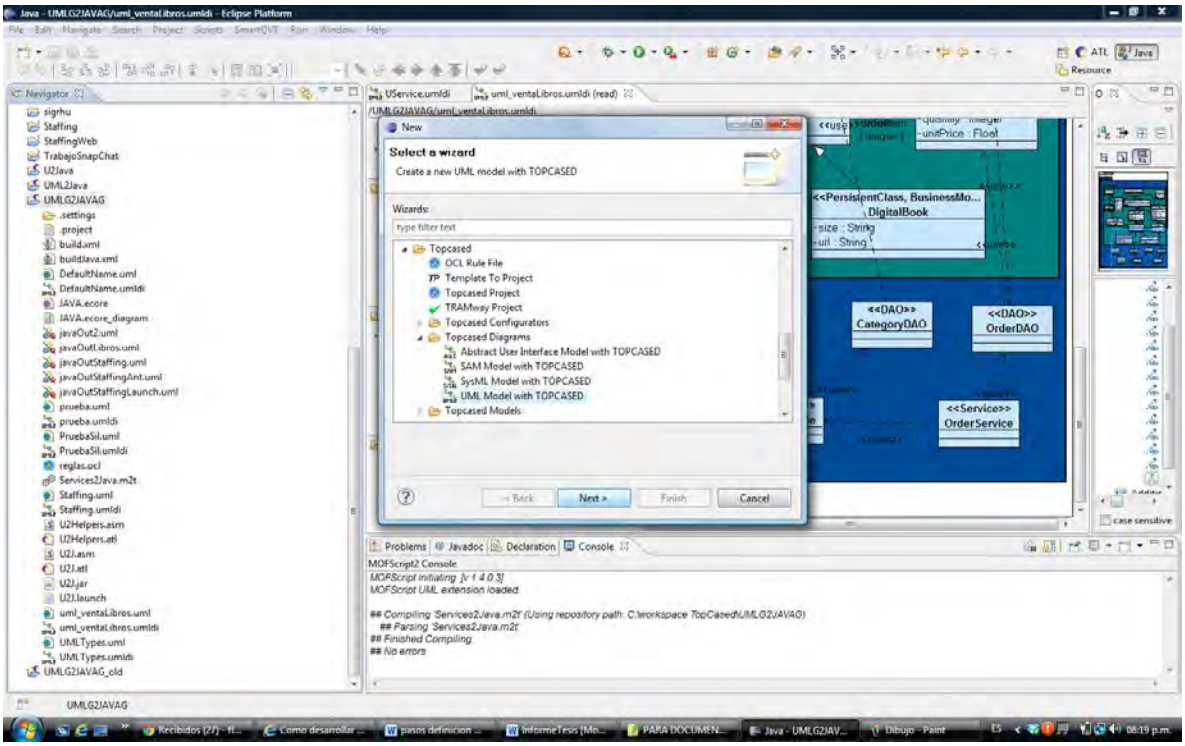
Ahora crearemos nuestro modelo de objetos para el sistema de venta de libros, que representa nuestro modelo PIM e indicaremos además con la ayuda de nuestro plugin, el paquete principal <<PackagePrincipal>>, el paquete de modelo <<PackageModel>>, los objetos de negocio <<BusinessModelElement>>, aquellos que serán persistentes <<PersistentClass>>, definiremos DAOS <<DAOS>> relacionados con las clases que van a administrar y servicios <<Service>> relacionados con los DAOS que van a utilizar.

Utilizaremos nuevamente la herramienta TopCased para diseñar el modelo uml incorporando nuestro Profile UService de la siguiente manera:

2_ Creamos el modelo uml_ventaLibros.uml:

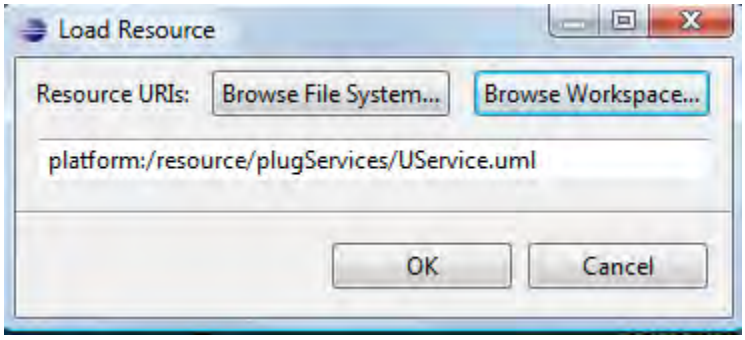
- Para crear el uml : new -> others -> Topcased -> TopcasedDiagram -> UML Model whit TOPCASED
- Se define el nombre del archivo y se deja seteado todo por defecto.

- Crear uml:



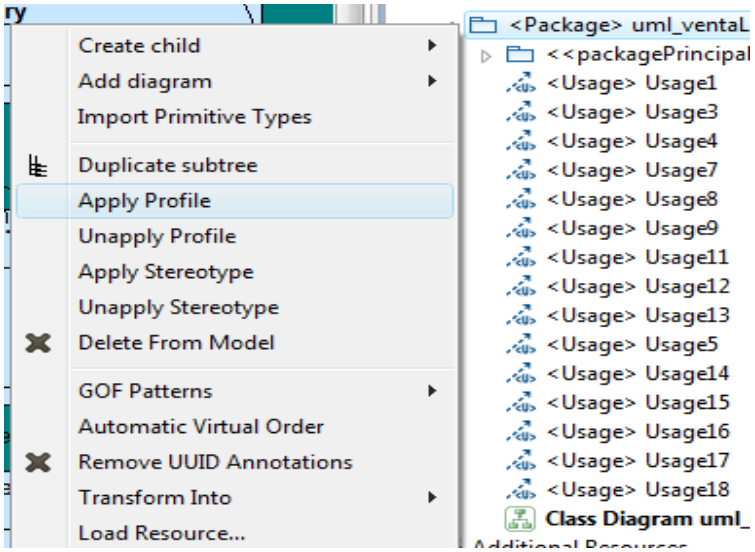
- **Importar el Profile**

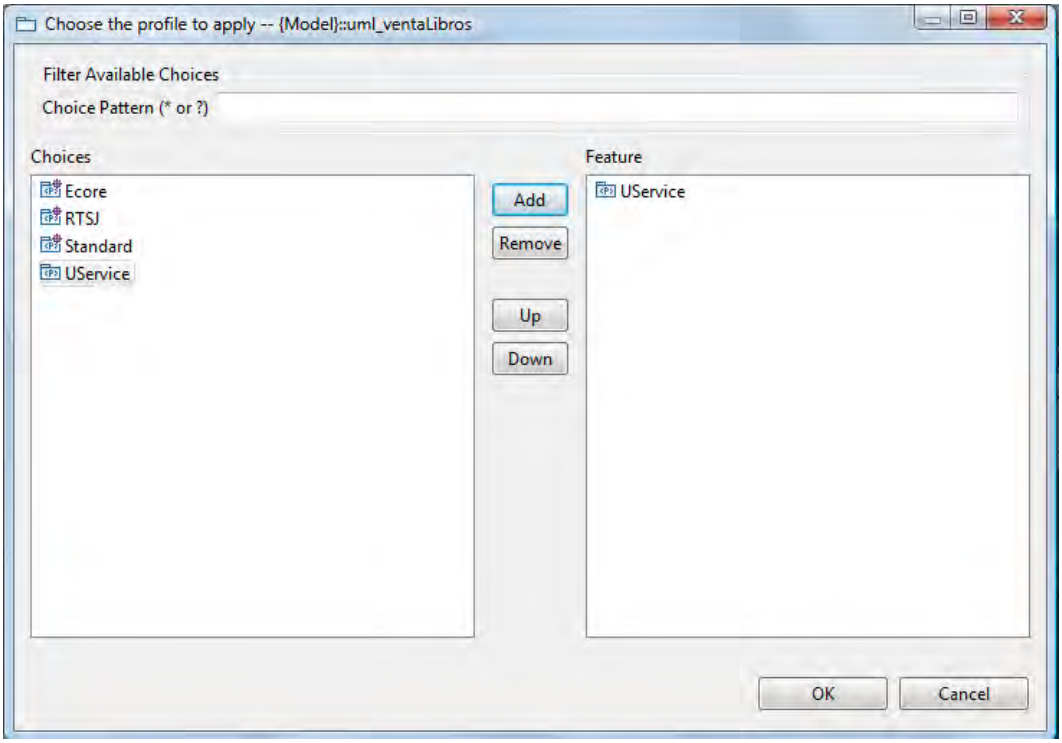
  Es necesario especificar los Perfiles que van a usarse en el modelo UML: en el Outline, **botón derecho -> Load resource**... En el cuadro de diálogo, seleccionar el fichero UML que contiene el Profile deseado y aceptar.



El fichero debe aparecer en el Outline, bajo la sección **Additional Resources** del árbol: este elemento contiene todos los recursos referenciados (Ficheros de Modelo o Profile).

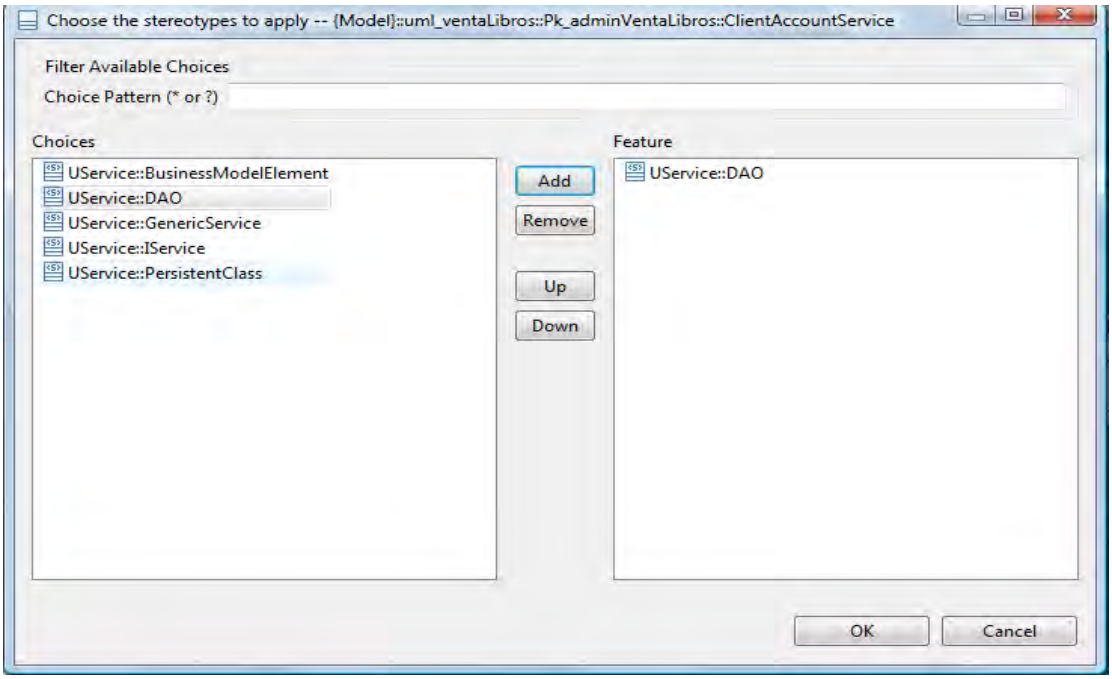
Posteriormente, es necesario aplicar un estereotipo a algún objeto del modelo. Seleccionar dicho objeto, pulsar **botón derecho -> Apply Profile** y elegir el Profile previamente cargado. Aparece un nuevo elemento (Profile Application) en el modelo, como hijo del paquete: se usa para guardar la versión del Profile que se ha aplicado.
(Nota: No es necesario referenciar primero el fichero del Profile antes de aplicar un estereotipo preexistente ("Standard" o "Ecore"). De hecho, al seleccionar un paquete y pulsar la opción Apply Profile, el Profile se añadirá automáticamente a los Recursos referenciados.)





- **Usar el Profile: Aplicar/Eliminar un estereotipo**

  A partir de este momento, es posible aplicar estereotipos a aquellos elementos que puedan recibirlos. Siempre desde el Outline, seleccionar un elemento UML. Botón derecho **-> Apply Stereotype**: se mostrarán los estereotipos disponibles. Elegir uno y aceptar. La etiqueta del elemento se actualizará en el Outline, y si el elemento está presente en el diagrama, también se mostrará el nuevo estereotipo.



Es posible eliminar un estereotipo aplicado a un elemento UML seleccionando dicho elemento y pulsando **botón derecho -> Unapply Stereotype**: Entonces se muestran todos los estereotipos aplicados al elemento para que podamos seleccionar cuál de ellos deseamos borrar.

- Ya incorporamos el profile UServices, ahora podemos diseñar fácilmente nuestro modelo de objetos UML definido en la figura 4.1, página 95 y agregarle comportamiento con el profile: UML + USERVICE.

- **Resultado del modelo:** Como resultado, tenemos el modelo de objetos de nuestro caso de estudio, venta de libros. Donde logramos plasmar los objetos definidos previamente, como característica distintiva con los estereotipos de UServices de la siguiente forma:

  - ✓ El paquete principal que contiene a todos los objetos del sistema, llamado "**Pk_adminVentaLibros**", identificado con el estereotipo <<PackagePrincipal>>, el mismo debe ser único para el proyecto y cumplir con las restricciones definidas en el capítulo OCL.

  - ✓ El paquete que contiene y representa a los objetos de negocio, llamado "**PK_model**", identificado con el estereotipo <<PackageModel>>, el mismo debe cumplir con las restricciones definidas en el capítulo OCL.

  - ✓ Los elementos de negocio identificados con el estereotipo <<BusinessModelElement>> y además aquellos que deben persistir en la base de datos con el estereotipo <<PersistentClass>>

  - ✓ Agregamos los DAOs indicados en el capítulo anterior: "**ClientAccountDAO**", "**BookSpecDAO**", "**CategoryDAO**", "**BookDAO**" y "**OrderDAO**", identificando las clases relacionadas como se describió previamente y distinguidos por el estereotipo <<DAO>>

  - ✓ De la misma forma los servicios indicados: "**ClientAccountService**", "**BookSpecService**", "**CategoryService**", "**BookService**" y "**OrderService**", identificando los DAOs y servicios relacionados.

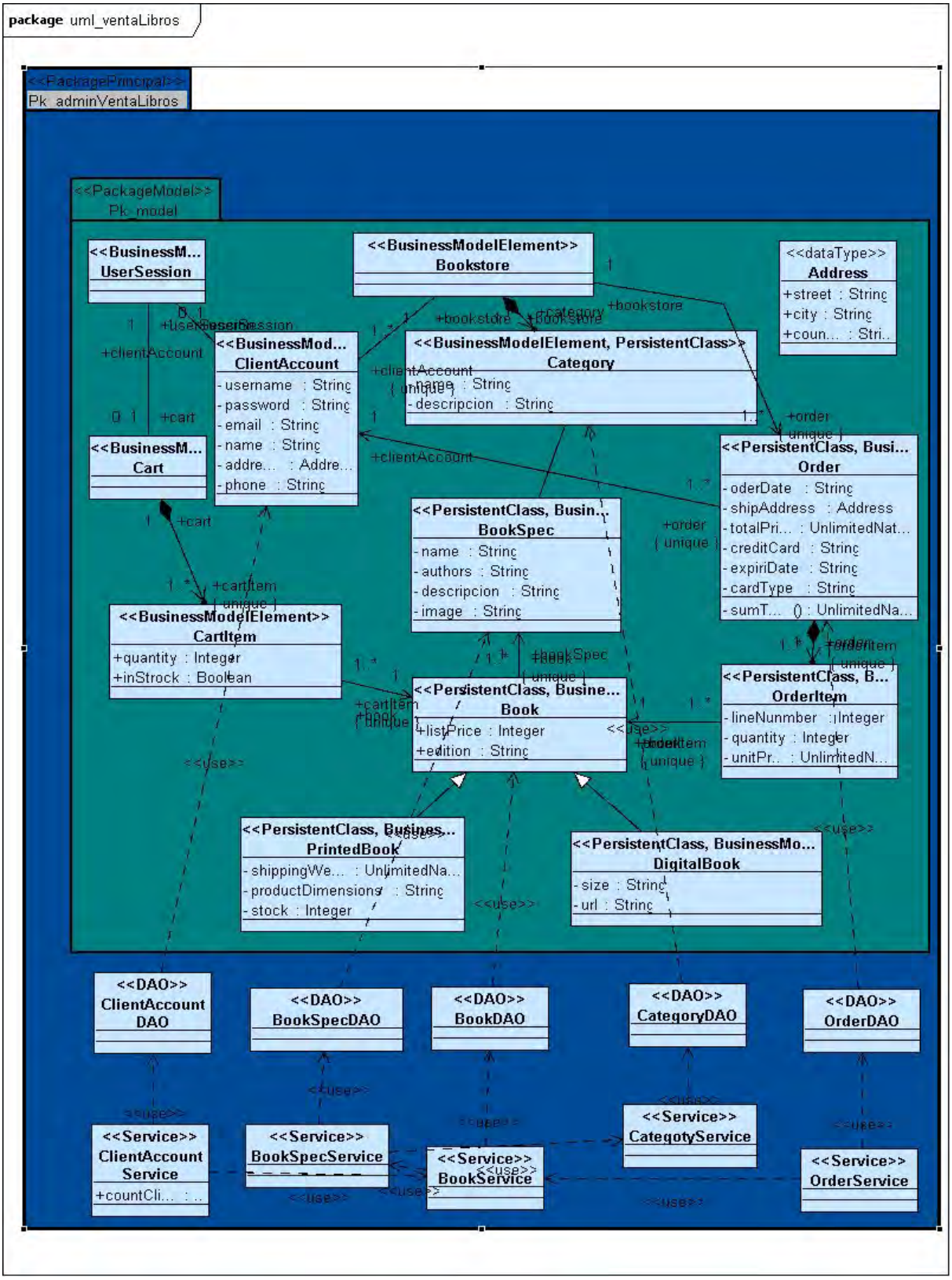


Figura 4.5. UML Venta de Libros

4.3.2 Desarrollo de reglas OCL para la validación de los modelos

▪ 4.3.2.1 Definiendo las reglas

Creamos el conjunto de reglas ocl que se utilizarán para validar nuestro modelo de entrada. Lo hacemos con la ayuda de la herramienta TopCased:

- Crear el archivo : new -> others -> Topcased -> OCL Rule File
- Definir el nombre y la ubicación del archivo
- Setear en "metamodel URI" el metamodelo de entrada -> http://www.eclipse.org/uml2/3.0.0/UML

Ya tenemos creado el archive de reglas, ahora tenemos que escribir nuestras reglas, las cuales mostramos a continuación:

- **Encabezado:**
  En el encabezado se debe indicar cuál es el metamodelo que se utiliza de referencia.

```
MainModel: http://www.eclipse.org/uml2/3.0.0/UML
```

- **Cuerpo:**
  Luego del encabezado se deben definir el conjunto de reglas OCL que se utilizarán para validar el modelo uml que se desea transformar. Básicamente se declara una función con la sintaxis '`context` nombre contexto `def`: nombre funcion() : return type =', donde context indica el tipo de objetos al que referencia la función, def el nombre de la función y luego de los dos puntos el tipo de retorno de la misma.

  A modo de ejemplo mostramos a continuación la implementación de las reglas definidas anteriormente para los estereotipos del proyecto. Cabe aclarar que además en la práctica se definieron reglas específicas del comportamiento de objetos

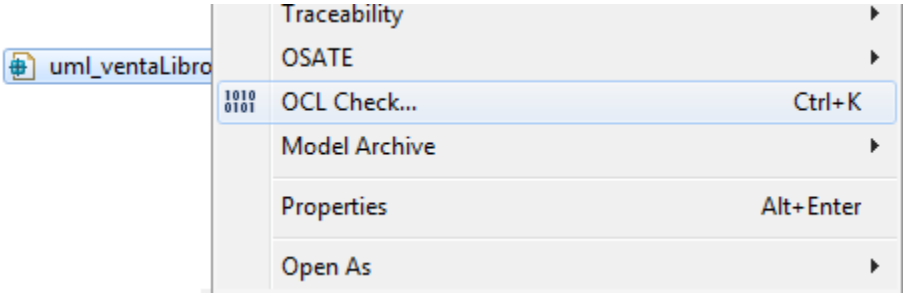
```
-- Los PackageModel tienen que estar dentro de un PackagePrincipal o de otro PackageModel
context Package inv incluidoPackageModel:
   self.isPackageModel() implies
         (self.nestingPackage.isPackageModel() or
          self.nestingPackage.isPackagePrincipal())
```

```
-- Todos los packages de un PackageModel deben ser tambien PackageModel
context Package inv packageModels:
   self.isPackageModel() implies
    (self.packagedElement -> select(e | e.oclIsTypeOf(Package))
         -> forAll (c| c.oclAsType(Package).isPackageModel()))


-- Todos los elementos de un PackageModel tienen que ser BussinessModelElement
context Package inv elementosBussinessModelElement:
   self.isPackageModel() implies
    (self.packagedElement -> select(e | e.oclIsTypeOf(Class))
         -> forAll (c| c.oclAsType(Class).isBusinessModelElement()))


-- Los BussinessModelElement tienen que estar dentro de un PackageModel
context Class inv packageDeBussinessModelElement:
   self.isBusinessModelElement() implies
         (self.owner.oclAsType(Package).isPackageModel())


-- Los PersistentClass deben ser BussinessModelElement
context Class inv persistentClassBussinessModelElement:
   self.isPersistentClass() implies
         (self.isBusinessModelElement())


-- Las relaciones de los Services deben ser de tipo dependencia Usage
context Class inv relacionesServicesUsage:
   self.isService() implies (self.clientDependency ->
       exists (c | not c.oclIsTypeOf(Usage)) = false)

-- Los Services deben estar relacionados al menos con un objeto
context Class inv relacionesServices:
         self.isService() implies (self.clientDependency -> notEmpty())

-- Los Services deben estar relacionados únicamente con objetos DAOs o Services
context Class inv servicesDao:
   self.isService() implies (self.clientDependency ->
               exists (c | c.oclIsTypeOf(Class) and not(c.oclAsType(Class).isDAO() or
   c.oclAsType(Class).isService())) = false)

-- Los DAO tienen que tener una relación de dependencia Usage
context Class inv relacionUsage:
   self.isDAO() implies (self.clientDependency ->
         exists (c | c.oclIsTypeOf(Usage)))


-- Los DAO deben estar relacionados únicamente con PersistentClass
context Class inv daosPersistentClass:
   self.isDAO() implies (self.clientDependency -> exists (c | c.oclIsTypeOf(Class)
and not c.oclAsType(Class).isPersistentClass()) = false)


-- Las Property deben tener un tipo
context Property inv tiposProperty:
   self.type -> notEmpty()
```

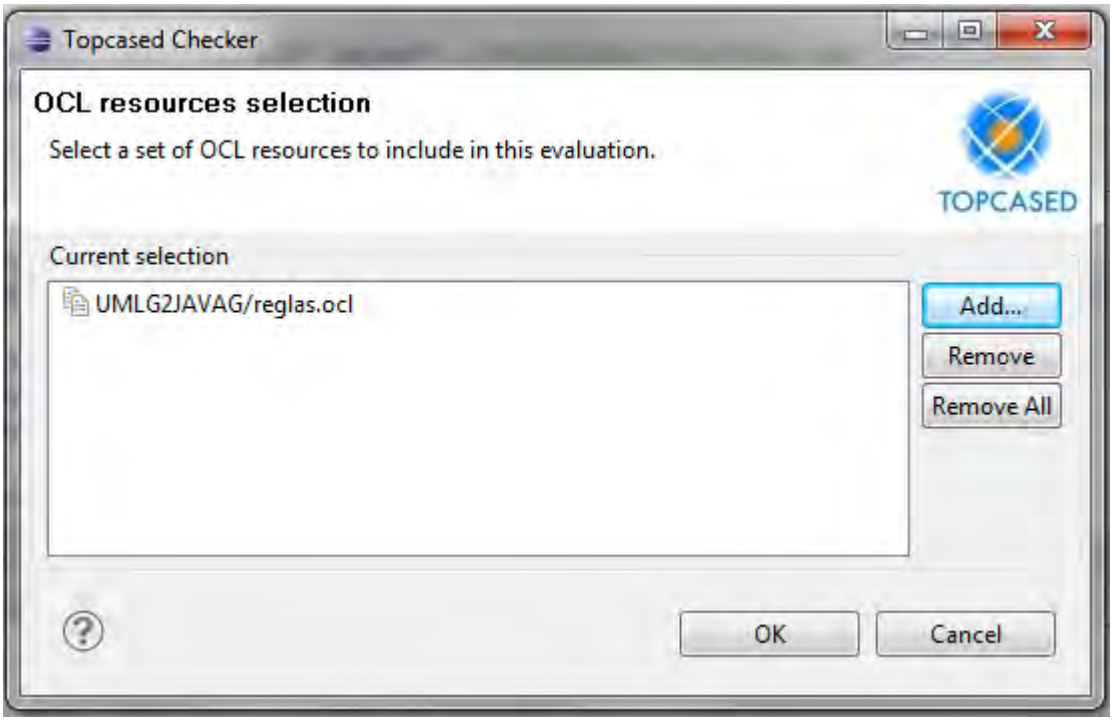
- 4.3.2.2 Validando el modelo de objetos

Validando el modelo de objetos

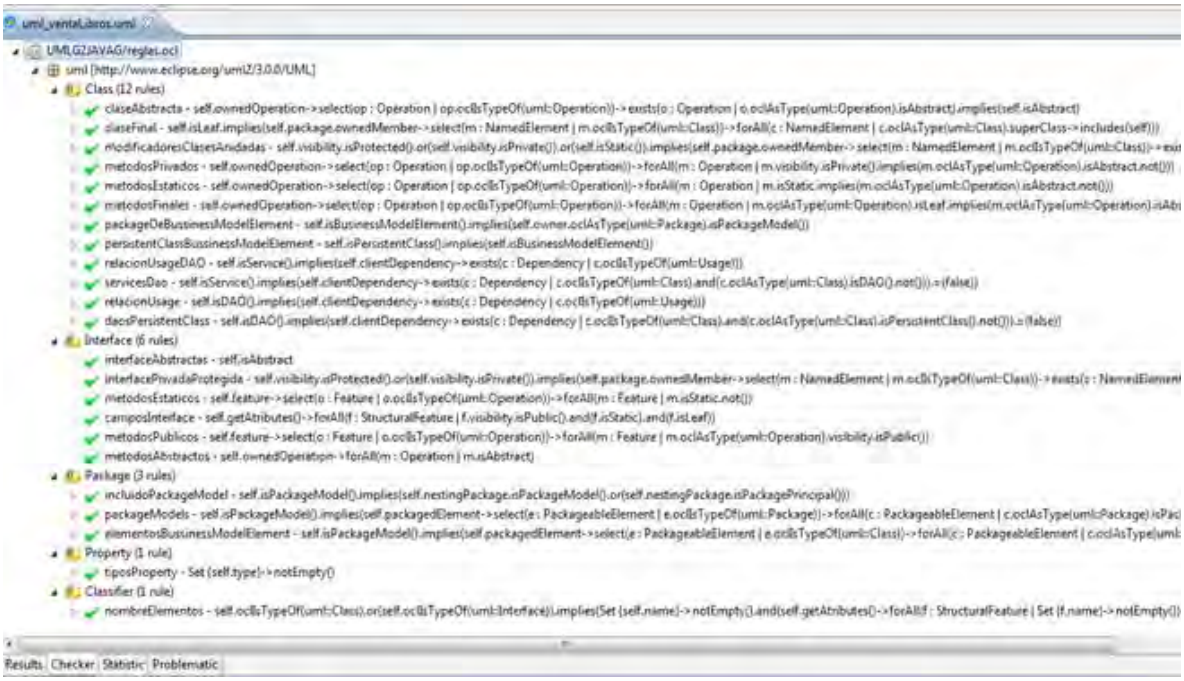
Antes de ejecutar la transformación debemos validar nuestro modelo para asegurarnos que cumpla con las reglas establecidas. . Para ello, nos paramos sobre nuestro modelo (uml_ventaLibros.uml), presionamos el botón derecho del mouse y seleccionamos la opción OCL Check…



Seleccionamos nuestro archivo de reglas (reglas.ocl) y lo ejecutamos presionando el botón OK

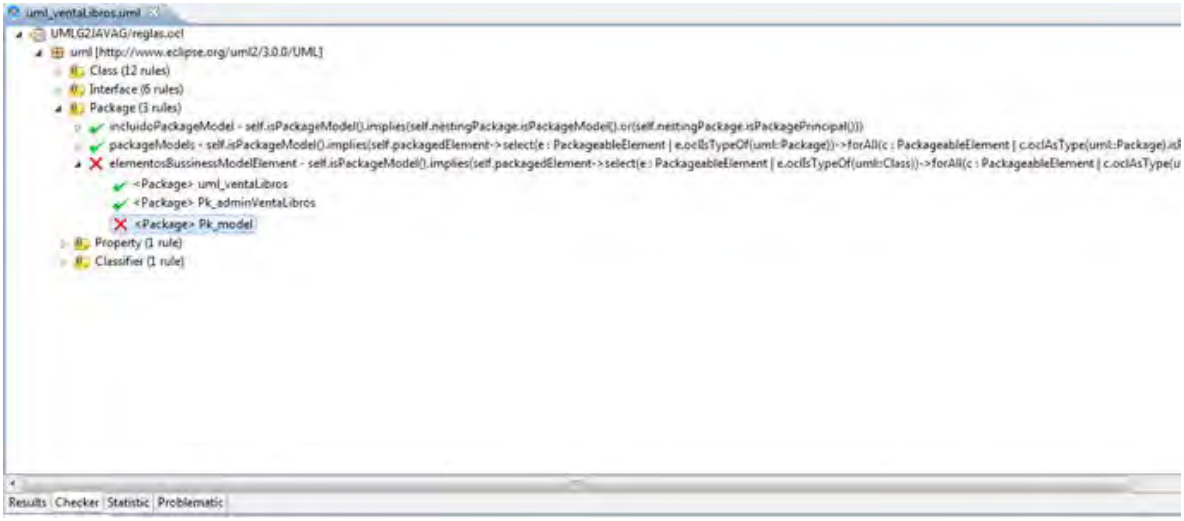


Como resultado de esta ejecución vamos a tener el siguiente resultado:



En esto caso obtenemos un resultado correcto ya que nuestro modelo cumple con todas las reglas establecidas.

Si modificamos el modelo por ejemplo le sacamos a la clase UserSession el estereotipo BussinessModelElement vamos a obtener este resultado:



En este caso nos muestra un error en la regla elementosBusinessModelElement, marcando además que el problema está en el package Pk_model

4.3.3 Desarrollo de transformación del PIM al PSM con ATL

▪ 4.3.3.1 Definiendo Metamodelos de dominio Source y Target

En esta etapa mostramos el desarrollo de la transformación ATL del modelo PIM al PSM. Definiendo los modelos Ecore de entrada y salida y las reglas de transformación aplicadas, para lograr un modelo PSM objetivo.

_ En primer lugar vamos a definir el dominio (metamodelo) de origen (source) que especifica el modelo de entrada **IN** y el dominio (metamodelo) de destino (target), que especifica el modelo de salida **OUT**. Siendo ambos instancias de MOF.

- **Metamodelo Origen:** En el metamodelo de origen utilizamos el estandar de UML 3.0.0 definido por EMF: uri: http://www.eclipse.org/uml2/3.0.0/UML, que representará el metamodelo de clases para nuestro modelo PIM, independiente de la plataforma que especifica el modelo de entrada **IN** en la transformación.

- **Metamodelo Destino:** Implementado a través del lenguaje Ecore, es un modelo reducido del metamodelo original para el lenguaje Java que **especificará** a nuestro modelo PSM derivado de la transformación, para luego a partir de él derivar a código Java.

▪ 4.3.3.1.1 Creamos el metamodelo JAVA.ecore

Como se mencionó en capítulos anteriores, dentro de Eclipse los metamodelos se especifican usando el meta-metamodelo ECore. Además, la definición de un metamodelo puede hacerse de cuatro maneras diferentes, obteniendo el mismo resultado:

- usando un editor gráfico para metamodelos ECore.
- a partir de un documento XML.
- a partir de un modelo UML.
- como interfaces de Java con Anotaciones.

En nuestra implementación, la creación del **metamodelo JAVA** se hizo a partir del editor gráfico.
Dicha definición fue completada y refactorizada para simplificar el procesamiento de las transformaciones.
En el mismo se representan los conceptos fundamentales del lenguaje Java tales como:

- Concepto de **interfaces**
- Relaciones de implementación entre clases e interfaces
- Limitación de herencia múltiple de clases
- Métodos constructores de las clases
- tipos de datos específicos del lenguaje JAVA

- definición del tipo **void** como tipo de retorno

Entre otros, siempre procurando no cambiar la semántica original.

- Para crear el modelo ECore, y no hacer más extensa la redacción con información redundante, dejamos un link explicado de forma simple, paso a paso en el apartado [*emf1], cómo crear un modelo Ecore. Mostrando a continuación el metamodelo Java resultante.

- Resultado del modelo JAVA.ecore:

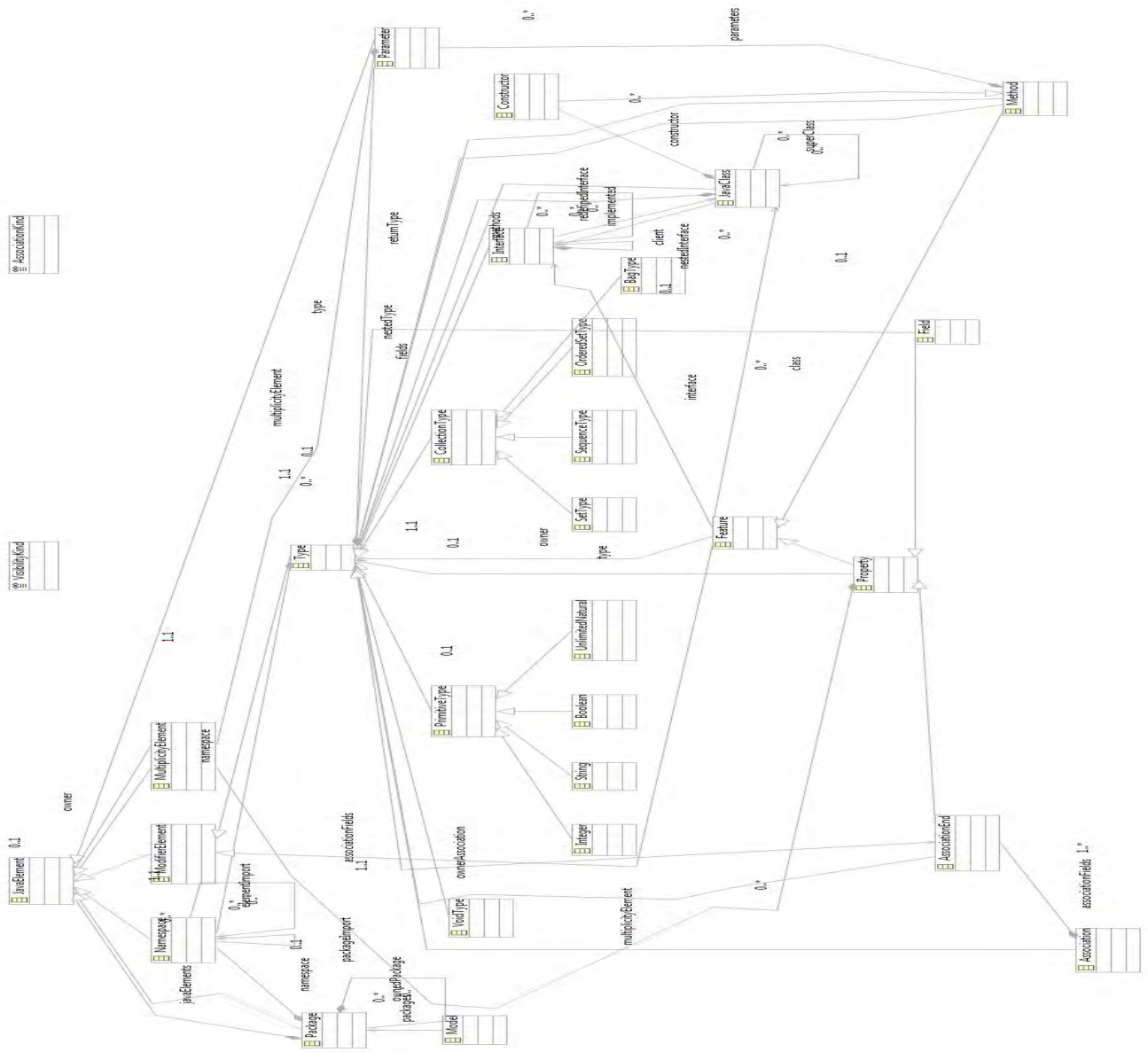


Figura 4.6. Metamodelo Java reducido.

4.3.3.2 Implementando la transformación del PIM al PSM con ATL

4.3.3.2.1 Definimos objetivo y comportamiento de la transformación

_ Explicaremos el objetivo y el comportamiento que derivan en nuestra transformación las clases pertenecientes al Perfil UServices.

- **Objetivo:**
  Deseamos en esta primer etapa de la transformación, desde el modelo PIM (UML+UServices : un modelo de clases bien formado pero independiente de la plataforma, con información adicional descriptiva de servicios, DAOs y objetos persistentes), derivar la implementación orientada a servicios, obteniendo en el modelo PSM de destino, el esqueleto del sistema, conformado por el árbol de paquetes , clases e interfaces que se definen bajo el siguiente criterio:

- **Comportamiento:**
  La transformación que vamos a definir, cuenta con un comportamiento común para los paquetes y objetos que no están estereotipados, los cuales se van a transformar a objetos JAVA y con un comportamiento particular para los paquetes y objetos estereotipados que denota el trabajo principal de la misma. El cual describimos a continuación para los objetos más destacados.
  Vamos a nombrar al modelo de entrada IN: ULM2 y al modelo de salida OUT:JAVA . A los objetos referenciados por el modelo al que pertenecen y el nombre de la clase de la siguiente manera UML2:Nombre.

- **Paquetes no estereotipados:**

  Objeto IN: UML2: Package
  Objeto OUT: JAVA: Package
  Características: Se transforma en un objeto 'Package' de JAVA con características similares al modelo de entrada, destacando la característica de:

  - **namespace:** que agrega un objeto 'namespace' como atributo del mismo, el cual se compone de un nombre que identifica el nombre extendido del package, conformado por el árbol de paquetes que contienen al mismo separados por '.' , ejemplo packagePadre.packageHijo.packageHijo2 , siendo packageHijo2 el nombre del objeto a transformar, packageHijo su inmediato contenedor y packagePadre el contenedor packageHijo, siendo la raíz del árbol. Un conjunto de elementos importados por el package, que puede estar vacío y un conjunto de packages también importados por el package, que puede estar vacío.

- **Clases no estereotipadas:**

  Objeto IN: UML2: Class
  Objeto OUT: JAVA: JavaClass
  Características: Se transforma en un objeto 'JAVACLASS' de JAVA con características particulares del lenguaje, destacando las siguientes:

- **Field:** Los atributos propios de la clase se transforman en objetos 'Field' de JAVA, con el mismo nombre y la detalle particular de cambiar la visibilidad de publica a privada, es una característica del lenguaje para resguardar el acceso a los mismos, creando para ello métodos de acceso y seteo, llamados 'getter' para leer el valor de la propiedad y 'setter' para cambiar el valor de la propiedad.

- **associationFields:** formamos la col de AssociationEnds como el conjunto de los atributos propios de la clase que tienen association (esto indica que son aquellos que derivan de una composición), unido el conjunto de los atributos que se sacan de las associations en las cuales dentro de su colección de navigableOwnedEnd tienen a esta clase como opuesta, esto indica que desde mi clase puedo navegar a este objeto, por lo tanto se transforma como un campo associationField.

- **Methods:** Se setea en la variable 'methods' una colección con los métodos indicados en el modelo transformados como methodos de java, con sus parámetros y tipos de retorno correspondiente a la regla de transformación de tipos de datos. Se crean además los métodos de acceso de cada atributo y associationEnd, definidos anteriormente, de la clase. Por ejemplo:

  _*Para el atributo y/o* associationEnds *de cardinalidad (1), ejemplo 'name' de tipo string, se crea el método 'public string getName() ' que retorna el valor del atributo name y public 'setName(String pname)' que setea el valor del atributo 'name' con el valor del parámetro 'pname' . Con el tipo de datos con su transformación correspondiente. Otro ejemplo: En el PIM, la clase Order se asocia con la clase ClientAccount con cardinalidad uno. Entonces en la clase Order del modelo destino, se generan los métodos getClientAccount(): ClientAccount y setClientAccount (clientAccount: ClientAccount) para poder acceder a dicho final de asociación*

  _ *Para el atributo y/o* associationEnds *de cardinalidad (*), se crea un método 'getter' para recuperar la colección y un método 'add'para agregar un objeto a la colección. Ejemplo: La clase Order del PIM se relaciona con las clase OrderItem con cardinalidad(*), por lo tanto se crea en la clase Order los métodos getOrderItems(), addOrderItem(OrderItem item), removeOrderItem(OrderItem item) .*

  _ Además se distingue los métodos constructores de la clase.

  _ Se crean y agregan a la colección los métodos definidos por las interfaces que la clase implementa, ya que el lenguaje JAVA obliga que la clase implemente todos los métodos derivados de las mismas.

- **isPersistent:** Se indica en una variable 'isPersistent' si la clase es o no persistente, en este caso será false, ya que es un objeto no estereotipado.

- **implemented:** Se indica en una variable 'implemented' la colección de objetos 'interfaces' que implementa la clase.

- **Interfaces no estereotipadas:**

  Objeto IN: UML2: Interface
  Objeto OUT: JAVA: Interface
  Características: Se transforma en un objeto 'INTERFACE' de JAVA con características particulares del lenguaje, destacando las siguientes:

  - **Field:** Los atributos propios de la interface son de tipos static (constantes en JAVA) , se transforman en objetos 'Field' de JAVA, con el mismo nombre .

  - **redefinedInterface:** Conjunto de interfaces que puede redefinir. A diferencia de las clases una interface puede derivar de otra o incluso de varias interfaces, en cuyo caso incorpora las declaraciones de todos los métodos de las interfaces de las que deriva.

- **Association no estereotipadas:**

  Objeto IN: UML2: Association
  Objeto OUT: JAVA: Association
  Caracteristicas: Se transforma en un objeto 'ASSOCIATION' de JAVA con características particulares del lenguaje.Representa a las relaciones entre objetos, guardando información destacable de las mismas, destacando las siguientes

  - **associationFields:** formamos la coleccion de AssociationEnds como el conjunto de los atributos propios de la clase que tienen association (esto indica que son aquellos que deriva de una composición), unido el conjunto de los atributos que se sacan de las associations en las cuales dentro de su coleccion de navigableOwnedEnd tienen a esta clase como opuesta, esto indica que desde mi clase puedo navegar a este objeto, por lo tanto se transforma como un campo associationField.

- **Métodos no estereotipados:**

  Objeto IN: UML2: BehavioralFeature
  Objeto OUT: JAVA: Method
  Caracteristicas: Se transforma en un objeto 'METHOD' de JAVA con características particulares del lenguaje, destacando las siguientes:

  - **returnType:** Se identifica el tipo de retorno del método en una variable a parte, 'returnType'.

  - **parameters:** Conjunto de parámetros del método.

    Objeto IN: UML2: BehavioralFeature
    Objeto OUT: JAVA: Constructor
    Caracteristicas: Se transforma en un objeto 'CONSTRUCTOR' de JAVA, en el caso que el método se identifique con el nombre de la clase, como característica particular del lenguaje son los métodos de construcción e inicialización de la clase. El constructor no tiene valor de retorno y su nombre es el mismo que el de la clase.

- **parameters:** Conjunto de parámetros del método, son valores de inicialización de la clase.

- **Atributos no estereotipados:**

  Objeto IN: UML2: StructuralFeature
  Objeto OUT: JAVA: Field
  Caracteristicas: Se transforma en un objeto 'FIELD' de JAVA con características particulares del lenguaje.Representa a los atributos propios de una clase.

- **visibility:** como ya comentamos, la visibilidad de un atributo se transforma en 'private'.

- **Tipos de Datos:** Tanto para los tipos de los atributos o métodos, se transforman de la siguiente manera.
  - Objeto IN: UML2: String Objeto OUT: JAVA: String
  - Objeto IN: UML2: Integer Objeto OUT: JAVA: int
  - Objeto IN: UML2: Real, Float Objeto OUT: JAVA: float
  - Objeto IN: UML2: Date Objeto OUT: JAVA: Date

  Cualquier otro tipo de dato se transforma a una clase con el mismo nombre, los mismos atributos y métodos para acceder a cada uno de esos atributos.

Hasta aquí mostramos las características relevantes de la transformación de los objetos no estereotipados del modelo de entrada a sus correspondientes del lenguaje JAVA en el modelo de salida. A continuación, pasaremos a detallar la transformación de objetos estereotipados, a partir de los cuales se crean paquetes, clases e interfaces que derivan de los mismos.

▪ Paquete estereotipado <<PackagePrincipal>>:

Objeto IN: UML2: Package
Objeto OUT: JAVA: Package
Caracteristicas: Se transforma en un objeto 'PACKAGE' de JAVA con las mismas características definidas para el package no estereotipado, pero a partir de este paquete principal, se crean un conjunto de paquetes, clases e interfaces que integran y definen el esqueleto de la arquitectura del sistema orientado a servicios que vamos a implementar, de la siguiente manera:
El paquete principal va a contener los siguientes paquetes, considerando 'namePrincipal' como ejemplo del nombre del paquete, servirá como parte de la nomenclatura:

- **namePrincipal_common:** En esta rama "common" se crearán todas las clases que definen comportamiento en común y abstractas del sistema. Este paquete contendrá los siguiente objetos:

  - **JavaClass AbstractPersistentObject:** clase Abstracta que determina un identificador 'id' y algún otro dato como versionado del objeto, de la cuál extenderán las clases persistentes.

  - **Package model:** Donde se transforman las clases del modelo de negocio, que son aquellas estereotipadas <<BusinessModelElement>> en el modelo de entrada.

    - **JavaClass UsserSession**.
    - **JavaClass ClientAccount**.
    - **JavaClass BookStore**.
    - **JavaClass BookSpec**.
    - **JavaClass Cart**.
    - **JavaClass CartItem**.
    - **JavaClass Book**.
    - **etc**

  - **Package service:** Donde se crean interfaces que definen comportamiento en común para los objetos servicios 'services' del sistema.

    - **Interface IServiceInterface**.
    - **Interface GenericServiceInterface**

- **Package impl:** sub paquete de **namePrincipal_common.service,** se guardan las clases abstractas y aquellas que implementan las interfaces que definen el comportamiento genérico de los objetos services.
    - **JavaClass AbstractService**.

- **Package filter:** Se crean las clases abstractas que definen comportamiento en común para los filtros de búsquedas usados por los DAOS en las consultas HQL en la administración de los datos de los objetos persistentes.

    - **JavaClass AbstractFilterParameter**
    - **JavaClass AbstractFilter**

    - **Package filter_parameter:** sub paquete de **namePrincipal_common. Package filter,** donde se guardan las clases **parámetro filter** usadas por las clases **filter,** siendo una por cada objeto persistente, la cual contiene el conjunto de variables que determina los parámetros de búsqueda para cada objeto.

        - **JavaClass ClientAccount_ParametroFilter**
        - **JavaClass BookSpec_ParametroFilter**
        - **JavaClass Book_ParametroFilter**
        - **JavaClass Order_ParametroFilter**
        - **etc**

- **Package DAO:** Donde se crean interfaces que definen comportamiento en común para los objetos daos 'DAOS' del sistema, quienes administran los datos persistentes.

    - **Interface AbstractDAOInterface**

    - **Package impl:** sub paquete de **namePrincipal_common.DAO,** donde se guardan las clases abstractas y aquellas que implementan las interfaces que definen el comportamiendo genérico de los objetos DAOS.

        - **JavaClass AbstractDAOImpl**

- **namePrincipal_server:** En esta rama "server" se crearán todas las clases que 'implementan' el comportamiento de la arquitectura planteada, se crean las interfaces junto con las clases de servicios y DAOS definidos en el modelo de entrada, y para cada una de ellas se plantea un conjunto de funciones particulares que define comportamiendo para la administración de sus objetos referentes en el modelo. Este paquete contendrá los siguiente objetos:

- **Package dao:** sub paquete de **namePrincipal_server,** donde se guardan las interfaces que implementan los DAOS del sistema, definidos en el modelo de entrada.

    - **Interface IClientAccountDAO_DaoImpl**
    - **Interface IBookSpecDAO_DaoImpl**
    - **Interface IBookDAO_DaoImpl**
    - **Interface IOrderDAO_DaoImpl**
    - **Interface ICategoryDAO_DaoImpl**

    - **Package impl:** sub paquete de **namePrincipal_server.dao,** donde se guardan las clases que implementan los Filters del sistema, las clases que implementan los filtros de búsqueda para cada objeto persistente.

        - **JavaClass ClientAccountDAO_DaoImpl**
        - **JavaClass BookSpecDAO_DaoImpl**
        - **JavaClass BookDAO_DaoImpl**
        - **JavaClass OrderDAO_DaoImpl**
        - **JavaClass CategoryDAO_DaoImpl**

        - **Package filter:** sub paquete de **namePrincipal_server.dao.impl,** donde se guardan las clases que implementan los Filters del sistema, las clases que implementan los filtros de búsqueda para cada objeto persistente.

            - **JavaClass ClientAccount_Filter**
            - **JavaClass BookSpec_Filter**
            - **JavaClass Book_Filter**
            - **JavaClass Order_Filter**
            - **etc**

- **Package service:** sub paquete de **namePrincipal_server,** donde se guardan las interfaces que implementan la definición publica del comportamiendo de los 'SERVICES' del sistema, definidos en el modelo de entrada.

    - **Interface IService_ClientAccountService**
    - **Interface IService_BookSpecService**
    - **Interface IService_BookService**
    - **Interface IService_Category**
    - **Interface IService_OrderService**

    - **Package impl:** sub paquete de **namePrincipal_server.service,** donde se guardan las clases que implementan los SERVICES del sistema.

- **JavaClass ClientAccountService_ServiceImpl**
- **JavaClass BookSpecService _ ServiceImpl**
- **JavaClass BookService _ ServiceImpl**
- **JavaClass CategoryService _ ServiceImpl**
- **JavaClass OrderService _ ServiceImpl**

- **Class estereotipada <<BusinessModelElement>>:**

Objeto IN: UML2: Class
Objeto OUT: JAVA: JavaClass
Caracteristicas: Se transforma en un objeto 'JAVACLASS' de JAVA con las mismas características definidas para el las clases no estereotipadas, explicadas anteriormente, salvo que en este caso como son clases que definen la lógica del modelo de negocios se guardan en el package '**namePrincipal_common.model**' y de esta manera quedan en un paquete independiente y global para el resto del sistema. De forma modular, este mismo conjunto de clases sirve para cualquier otra arquitectura que se desee implementar sobre el problema de negocio, independientes entre sí.

- **JavaClass UsserSession**.
- **JavaClass ClientAccount**.
- **JavaClass BookStore**.
- **JavaClass BookSpec**.
- **JavaClass Cart**.
- **JavaClass CartItem**.
- **JavaClass Book**.
- **Etc**

- **Class estereotipada <<PersistentClass>>:**

Objeto IN: UML2: Class
Objeto OUT: JAVA: JavaClass
Caracteristicas: Se transforma en un objeto 'JAVACLASS' de JAVA con las mismas características definidas para las clases no estereotipadas, identificando la propiedad 'isPersistent' como true.

- **JavaClass ClientAccount**.
- **JavaClass BookStore**.
- **JavaClass BookSpec**.
- **JavaClass Category**.
- **JavaClass Book**.
- **JavaClass Order**.
- **JavaClass OrderItem**.
- **Etc**

- **isPersistent:** Se indica en la propiedad 'isPersistent' si la clase es o no persistente, en este caso será **true**, ya que es un objeto que debe persistir en el repositorio de datos.

- **superClass:** toda clase persistente extiende de la clase abstracta '**AbstractPersistentObject' definida en el package 'namePrincipal_common'.**

A partir de este conjunto de clases además, se crean las clases parametroFilter y Filters correspondiente, una por cada clase persistente. Se guardan en el paquete '**namePrincipal_common.Package filter.Package filter_parameter'** y '**namePrincipal_server.Package dao.Package impl.package filter**' respectivamente.
A continuación se describen las características de las mismas.Tomamos como ejemplo el Filter '**JavaClass ClientAccount_Filter**'

- **JavaClass ClientAccount_Filter':** Como toda clase filter, se usa para definir los filtros de búsqueda que se van a utilizar en la administración de los datos de la clase por el DAO correspondiente. Se indica en una variable 'isPersistent' si la clase es o no persistente, en este caso será **true**, ya que es un objeto que debe persistir en el repositorio de datos.
    - **Field:** Los Filters tienen como atributo un objeto de tipo 'ParametroFilter' que utiliza para recibir seteados por parámetro los valores de búsqueda necesarios para la misma. De esta manera las clases encapsulan comportamiento genérico y dinámico, ya que se determina la búsqueda dependiendo los valores setados, que pueden ser diferentes en cada instancia del llamado. Ejemplo del caso:
        - **Field 'ClientAccount_ParametroFilter' parameter.**

    - **Methods:** Entre los métodos que se distinguen, se define la lógica de búsqueda por los atributos de la clase, y se crea un método llamado '**createFilter_nameAttribute**', siendo **nameAttribute** el nombre del atributo de la clase persistente, se crea uno por cada atributo de búsqueda. Ejemplo:
        - **'Method createFilter_username()'.**

▪ Class estereotipada <<DAO>>:

Objeto IN: UML2: Class
Objeto OUT: JAVA: JavaClass
Caracteristicas: Se transforma en un objeto 'JAVACLASS' de JAVA, la cual tendrá definido un comportamiento particular para la administración de datos del objeto persistente indicado por su relación en el modelo, para ello la clase extenderá de la clase predefinida '**AbstractDAOImpl**' la cual, implementa la interface '**AbstractDAOInterface**' que se encuentran en el package '**namePrincipal_common.Package dao'** y las mismas definen, como dijimos previamente, un comportamiento común para las clases DAOs, como por ejemplo save(Objet entity),

update(Objet entity), delete(Objet entity), para persistir, salvar información o borrar un objeto respectivamente.

- **JavaClass AbstractDAOImpl:** La implementación de la interface predefinida '**AbstractDAOInterface**' determinará los detalles particulares y semánticos de la arquitectura usada.

  - **Field GenericDAO genericDAO:** atributo que representa el objeto genérico a administrar que luego será instanciado por cada clase particular.

  - **Methods:** Entre los métodos que se distinguen. Ejemplo:

    - **'Method save(Object entity)'.**
    - **'Method update(Object entity)'.**
    - **'Method delete (Object entity)'.**

Por cada DAO se crea una interface nomenclada '**InameDAO_DAO'** siendo nameDAO el nombre del objeto <<DAO>> que se está transformando. Esta interface, define comportamiento particular del dao que se relaciona con la arquitectura de búsqueda por Filtros definida. Entonces las clases DAOS se guardan en el paquete '**namePrincipal_server.Package dao.Package impl'** y las interfaces en **namePrincipal_server.Package dao'.**
Tomamos como ejemplo el DAO '**JavaClass Order_DAOImpl**'.

- **JavaClass Order_ DAOImpl':**

  - **implemented:** implementa la interface

    Interface 'IOrder_DAO'.

  - **Methods:** Entre los métodos que se distinguen, se redefinen los métodos, save, update, delete que hereda de la clase abstracta, se define además la lógica de búsqueda por los atributos de la clase, y se crea un método llamado '**createFilter_nameAttribute**', siendo **nameAttribute** el nombre del atributo de la clase persistente, se crea uno por cada atributo de búsqueda. Ejemplo:

    - **'Method save(Order entity)':** Persiste la información del objeto Order en la base de datos**.**
    - **'Method update(Order entity)':** Actualiza la información del objeto en la base de datos**.**
    - **'Method delete (Order entity)':** Borra la información del objeto en la base de datos**.**

    - **'Method get_Order(Integer id)':** Obtiene la información del objeto Order identificado por el parámetro id en la base de datos**.**

- **'Method get_Order (Order_ParametroFiltro parameterFilterOrder)':** Obtiene la información del objeto Order que coincide con el conjunto de valores definidos en el parámetro **parameterFilterOrder** en la base de datos**.**

- **'Method getAll_Order()':** Obtiene 'todos' los objetos de tipo Order en la base de datos**.**

- **'Method getAll_Order() (Order_ParametroFiltro parameterFilterOrder)':** Obtiene 'todos' los objetos de tipo Order que coinciden con el conjunto de valores definidos en el parámetro **parameterFilterOrder** en la base de datos**.**

- **'Method getAll_OrderItemByOrder() (Order_ParametroFiltro parameterFilterOrder)':** Obtiene 'todos' los objetos de tipo OrderItem, siendo éstos parte de la composición de la clase que coincide con el conjunto de valores de la orden definidos en el parámetro **parameterFilterOrder** en la base de datos**. Se crea un método getAll_+'nombreAttributo' +'By' +'nombreClase', para cada conjunto de atributos que componen la clase (existe una relación de asociación de composición con la misma).**

- **'Method getAll_OrderByClientAccount() (Order_ParametroFiltro parameterFilterOrder)'**: Se define un método **getAll_OrderBy+'nombreAttributo'** por cada atributo navegable desde la clase que obtiene 'todos' los objetos de tipo Order que pertenecen al objeto que conjunto de valores del atributo definido en el parámetro "**parameterFilterClientAccount''** en la base de datos**.**

- **De haberse definido otros métodos en la clase DAO se transforman como la transformación normal de métodos definida para el resto de las clases.**

- **Class estereotipada <<Services>>:**

  Objeto IN: UML2: Class
  Objeto OUT: JAVA: JavaClass
  Caracteristicas: Se transforma en un objeto 'JAVACLASS' de JAVA, la cual tendrá definido un comportamiento particular para la administración de datos de la capa de negocio. Definimos una clase abstracta '**AbstractService**', donde se definen métodos genéricos de los servicios, de la cual todos los servicios van a extender, la misma se encuentra en el paquete **namePrincipal_common.Package service.Package Impl**. Esta interface contiene como variable un objeto genérico DAO. Cada Servicio maneja un conjunto de datos sobre objetos particulares, para ello se relaciona con un DAO particular, quien servirá de apoyo para el acceso a datos de una manera transparente al servicio.

La clase '**AbstractService**' definirá los métodos públicos de acceso definidos por el dao, además de los getter y setter para acceder y setear la variable DAO correspondiente.

- **JavaClass AbstractService:** La implementación de la clase determinará los detalles particulares y semánticos de la arquitectura usada.

    - **Field AbstractDAOInterface abstratcDAO:** atributo que representa el objeto genérico DAO a utilizar que luego será instanciado por cada clase particular.

    - **Methods:** Entre los métodos que se distinguen. Ejemplo:

        - **'Method save(Object entity)'.**
        - **'Method update(Object entity)'.**
        - **'Method delete (Object entity)'.**
        - **'Method getAbstractDAO()'.**
        - **'Method setAbstractDAO (AbstractDAO abstractDAO)'.**

Además por cada Servicio se crea una interface nomenclada '**IService_nameService'** siendo nameService el nombre del objeto <<Service>> que se esta transformando. Esta interface, define comportamiento particular del servicio que se relaciona con la arquitectura de servicios definida. Entonces las clases SERVICES se guardan en el paquete '**namePrincipal_server.Package services.Package impl'** y las interfaces en **namePrincipal_server.Package services'.** Un servicio también puede definir comportamiento adicional sobre la lógica de negocio y pueden colaborar con otros servicios, por lo tanto podrá estar relacionado y hacer uso de otros servicios.
Tomamos como ejemplo el SERVICE '**JavaClass OrderService_ServiceImpl**'.

- **JavaClass 'OrderService _ ServiceImpl':**

    - **implemented:** implementa la interface

      Interface 'IService_OrderService'.

    - **superClass:** extiende la clase **JavaClass 'AbstractService'**

    - **Methods:** Entre los métodos que se distinguen, se define la lógica de búsqueda por los atributos de la clase, y se crean los métodos de seteo de la variable **dao**. Ejemplo:

        - **'Method getOrderDAO ()':** retorna la variable dao**.**
        - **'Method setOrderDAO (ClientAccountDAO_DaoImpl clientAccountDAO)':** setea la variable dao**.**

- **‘Method save(Order entity)’:** persiste la información del objeto Order en la base de datos**.**
- **‘Method update(Order entity)’:** actualiza la información del objeto en la base de datos**.**
- **‘Method delete (Order entity)’:** borra la información del objeto en la base de datos**.**

- **‘Method get_Order (Order_ParametroFiltro parameterFilterOrder)’:** Obtiene la información del objeto Order que coincide con el conjunto de valores definidos en el parámetro **parameterFilterOrder** en la base de datos**.**

- **‘Method getAll_Order()’:** obtiene ‘todos’ los objetos de tipo Order en la base de datos**.**

- **‘Method getAll_Order() (Order_ParametroFiltro parameterFilterOrder)’:** Obtiene ‘todos’ los objetos de tipo Order que coincide con el conjunto de valores definidos en el parámetro **parameterFilterOrder** en la base de datos**.**

- **‘Method getAll_OrderItemByOrder() (Order_ParametroFiltro parameterFilterOrder)’:** Obtiene ‘todos’ los objetos de tipo OrderItem, siendo éstos parte de la composición de la clase que coincide con el conjunto de valores de la orden definidos en el parámetro **parameterFilterOrder** en la base de datos**. Se crea un método getAll_+’nombreAttributo’ +’By’ +’nombreClase’, para cada conjunto de atributos que componen la clase (existe una relación de asociación de composición con la misma).**

- **‘Method getAll_OrderByClientAccount() (Order_ParametroFiltro parameterFilterOrder)’**: Se define un método **getAll_OrderBy+’nombreAttributo’** por cada atributo navegable desde la clase que obtiene ‘todos’ los objetos de tipo Order que pertenecen al objeto que conjunto de valores del atributo definido en el parámetro **parameterFilterClientAccount’’** en la base de datos**.**

- **De haberse definido otros métodos en la clase DAO se transforman como la transformación normal de métodos definida para el resto de las clases.**

Una vez completada la explicación del armado lógico de la transformación, donde pudimos expresar paso a paso el impacto de los objetos transformados; creamos clases, interfaces y paquetes para poder plasmar el modelo de la arquitectura de servicios. Vamos a mostrar en el siguiente punto, parte del archivo ‘.atl’ definido para nuestra transformación, llamado ‘U2J.atl’, explicaremos los detalles más importantes.

4.3.3.2.2 Detalles de implementación de la transformación ATL

En el desarrollo de la transformación definimos dos archivos principales, uno 'U2J.atl' donde escribiremos las reglas de transformación y otro 'U2Helpers.atl' donde definimos las funciones auxiliares que vamos a utilizar de soporte en las transformaciones. Tendremos un archivo adicional auxiliar 'UMLTypes.atl' con los tipos de datos que también nos sirve de soporte, desarrollado para simplificar el manejo de tipos de datos.

Archivo 'U2Helpers.atl':

Los helpers nos permiten definir funciones auxiliares que son de gran ayuda en las transformaciones. Las funciones están ligadas a un contexto de objetos y pueden servir para realizar cálculos, delimitar subconjuntos, realizar consultas booleanas, entre otras aplicaciones.

- **Encabezado:**
  En el encabezado se debe indicar que es una librería con la sintaxis 'library' y el nombre de la misma y se además la URI del metamodelo que refiere el universo de objetos que vamos a manejar.

```
-- @path MM=platform:/plugin/org.eclipse.uml2.uml/model/UML.ecore

library U2Helpers;
```

- **Cuerpo:**

  Luego del encabezado se debe definen el conjunto de funciones que conforman la librería. Básicamente se declara una función con la sintaxis '`helper context` nombre contexto `def`: nombre funcion() : return type =', donde context indica el tipo de objetos al que referencia la función, def el nombre de la función y luego de los dos puntos el tipo de retorno de la misma.

```
-- función que retorna el nombre extendido del package solicitado, separado por puntos.
helper context UML2!Package def: getExtendedName() : String =
       if not(self.name.oclIsUndefined()) then
             self.nestingPackage.getExtendedName() + '.'+ self.name
        else 'Model' endif ;


-- función que retorna el nombre extendido del package solicitado, separado por puntos.
helper context UML2!Classifier def: getQualifiedName() : String =
      self.owner.getExtendedName() +'.'+ self.name;


-- función que retorna el nombre extendido del package solicitado, separado por puntos.
helper context UML2!Interface def: getQualifiedName() : String =
      if  thisModule.inTypeElements->includes(self)  then  self.owner.getExtendedName()
+'.'+ self.name else  self.owner.getExtendedName() +'.'+ self.name+'Interface'endif;
```

```
-- Función que retorna una coleccion con los elementos de tipo AssociationEnds del classifier solicitado
helper context UML2!Classifier def: getElementImports() : Set(UML2!Namespace) =

      -- Acá formamos la col de namespace como el conjunto de los atributos propios de la clase que tienen association
      -- (esto indica que son aquellos que deriva de una composición), unido el conjunto de los atributos que se sacan de las associationes en las cuales
      -- dentro de su coleccion de navigableOwnedEnd tienen a esta clase como opuesta, esto indica que desde mi clase puedo navegar a este objeto, por lo incluyo
      -- en la colección de elementImports como namespace

   (   self.feature   ->select(a   |   a.oclIsTypeOf(UML2!Property))->select(c|   not c.association.oclIsUndefined() ) -> collect (x | thisModule.resolveTemp(x.type,'n' )))

        -> union
        ((UML2!Association.allInstances()->select(a  |  (a.navigableOwnedEnd  ->select  ( e|         e.getOppositeNavigator().type.equals(self))).notEmpty())->         collect(i| (i.navigableOwnedEnd -> select(x | not x.type.equals(self))))) -> flatten() -> collect (x | thisModule.resolveTemp(x.type,'n' )));



 -- Función que retorna una colección con los objetos con relación de conocimiento de la clase que están estereotipados como DAO
 helper context UML2!Classifier def : getDAOUsages():Set(UML2!Classifier)=
      self.clientDependency  ->  select(a|a.oclIsKindOf(UML2!Usage)  )->select(elem  | elem.supplier.first().isDAO())->collect(c | c.supplier.first());


 -- Función que retorna una colección con los objetos con relación de conocimiento de la clase que están estereotipados como Service
 helper context UML2!Classifier def : getServiceUsages():Set(UML2!Classifier)=
      self.clientDependency  ->  select(a|a.oclIsKindOf(UML2!Usage)  )->select(elem  | elem.supplier.first().isService())->collect(c | c.supplier.first());


 -- Función que retorna true si el elemento StructuralFeature es un Attribute de una clase BusinessModelElement , false en cc
helper context UML2!BehavioralFeature def: isMethodsStereotype() : Boolean  =
      if (not self.class.oclIsUndefined())then
          not self.class.isClassNoStereotype()
       else false endif;


 -- Función que retorna true si el elemento Property es un Attribute, false en cc
      helper context UML2!StructuralFeature def: isAttribute() : Boolean  =
      if  (not  self.class.oclIsUndefined()  and  self.association.oclIsUndefined()  and self.associationEnd.oclIsUndefined() )
      then true else false endif;
```

```
-- Función que retorna true si el elemento StructuralFeature es un Attribute de una
clase BusinessModelElement , false en cc
helper context UML2!StructuralFeature def: isAttributeStereotype() : Boolean  =


if (not self.class.oclIsUndefined())then
  not self.class.isClassNoStereotype()
 else false endif;


helper context UML2!Class def: isService(): Boolean =
      not self.getAppliedStereotype('UService::Service').oclIsUndefined();
helper context UML2!Class def: isDAO(): Boolean =
      not self.getAppliedStereotype('UService::DAO').oclIsUndefined();


helper context UML2!Class def: isBusinessModelElement(): Boolean =
not self.getAppliedStereotype('UService::BusinessModelElement').oclIsUndefined();


helper context UML2!Class def: isPersistentClassElement(): Boolean =
      not self.getAppliedStereotype('UService::PersistentClass').oclIsUndefined();


helper context UML2!Class def: isClassNoStereotype(): Boolean =
if(not     self.getAppliedStereotype('UService::Service').oclIsUndefined()     or     not
self.getAppliedStereotype('UService::DAO').oclIsUndefined())then
    false else true endif;


helper context UML2!Package def: isPackagePrincipal(): Boolean =
      not self.getAppliedStereotype('UService::PackagePrincipal').oclIsUndefined();


-- retorna false si el package tiene algún stereotype asociado, true en cc
helper context UML2!Package def: isPackageNoStereotype(): Boolean =
  if(self.getAppliedStereotypes()->asSet()->size()>0 )then
    false else true endif;


-- Función que retorna una coleccion con los elementos de tipo Class estereotipados con
Service del package solicitado
helper context UML2!Package def: getServices() : Set(UML2!Class) =
      self.ownedElement     ->select(c     |     c.oclIsTypeOf(UML2!Class)     and     not
c.getAppliedStereotype('UService::Service').oclIsUndefined());


-- Función que retorna una coleccion con los elementos de tipo Class estereotipados con
BusinessModelElement del package solicitado
helper context UML2!Package def: getBusinessModelElement() : Set(UML2!Class) =
      self.ownedElement     ->select(c     |     c.oclIsTypeOf(UML2!Class)     and     not
c.getAppliedStereotype('UService::BusinessModelElement').oclIsUndefined());
```

```
-- Función que retorna una coleccion con los elementos de tipo Class estereotipados con BusinessModelElement del package solicitado
helper context UML2!Package def: getPersistentClassElement() : Set(UML2!Class) =
      self.ownedElement     ->select(c     |     c.oclIsTypeOf(UML2!Class)     and     not c.getAppliedStereotype('UService::PersistentClass').oclIsUndefined());


-- Función que retorna una coleccion con los elementos de tipo Class estereotipados con BusinessModelElement del package solicitado y todos sus subpackages
helper context UML2!Package def: getAllPersistentClassElement() : Set(UML2!Class) =
      UML2!Class.allInstances()->   select(c   |   c.oclIsTypeOf(UML2!Class)   and   not c.getAppliedStereotype('UService::PersistentClass').oclIsUndefined())->flatten();


-- Función que retorna una coleccion con todos los elementos de tipo Class estereotipados con BusinessModelElement en el package y sus subpackages
helper context UML2!Package def: getAllDAOClassElement() : Set(UML2!Class) =
      UML2!Class.allInstances()->   select(c   |   c.oclIsTypeOf(UML2!Class)   and   not c.getAppliedStereotype('UService::DAO').oclIsUndefined())->flatten();


-- Función que retorna una coleccion con todos los elementos de tipo Class estereotipados con Service del package solicitado
helper context UML2!Package def: getAllServices() : Set(UML2!Class) =
      UML2!Class.allInstances()->select(c    |    c.oclIsTypeOf(UML2!Class)    and    not c.getAppliedStereotype('UService::Service').oclIsUndefined())->flatten();


-- Función que retorna una coleccion con los elementos de tipo Class estereotipados con BusinessModelElement del package solicitado
helper context UML2!Package def: getDAOSElement() : Set(UML2!Class) =
      self.ownedElement     ->select(c     |     c.oclIsTypeOf(UML2!Class)     and     not c.getAppliedStereotype('UService::DAO').oclIsUndefined());


-- Función que retorna una coleccion con los elementos de tipo Package estereotipados con PackageModel del package solicitado
helper context UML2!Package def: getPackageModels() : Set(UML2!Package) =
      self.ownedElement    ->select(c     |     c.oclIsTypeOf(UML2!Package)     and     not c.getAppliedStereotype('UService::PackageModel').oclIsUndefined());


-- Función que retorna una coleccion con los elementos de tipo Namespace que corresponden a los packageImport del namespace indicado
helper context UML2!Element def: getPackageImport() : Set(UML2!Namespace) =
self.packageImport  ->collect(a  |  if(a.importedPackage.isPackageNoStereotype())  then thisModule.resolveTemp(a.importedPackage,'n' ) else a.importedPackage.namespace endif)
;
```

Archivo 'U2J.atl':

- **Encabezado:**
  En el encabezado se definen los metamodelos que son utilizados en la transformación, UML2 y JAVA definen un identificador de los mismos y como se puede observar se expresa el path (URI, con los que fueron publicados) de cada uno en la parte superior.
  La sentencia 'create' define la signatura de la transformación, se especifican los modelos de entrada y de salida utilizando sus identificadores, en este caso tenemos dos modelos de entrada UML2 (INUML e INTYPES) y uno de salida JAVA (OUTJAVA), del tipo de los metamodelos definidos, que serán referenciados por dichos nombres en las transformaciones.

```
-- @path UML2=platform:/plugin/org.eclipse.uml2.uml/model/UML.ecore
-- @path JAVA=/UMLG2JAVAG/JAVA.ecore
-- @path UServices=/PluginProject/UServices.ecore

module U2J;
create OUTJAVA: JAVA from INUML: UML2, INTYPES: UML2;
```

- **Seguido del encabezado comienza la definición de las reglas de transformación:**

- **Elementos no estereotipados:**

Como primera regla, mostramos una regla abstracta, la misma define comportamiento para la transformación de elementos que extienden al tipo de elemento definido como entrada. Es una regla que puede ser extendida por varias reglas.
La regla abstracta 'Element', define que para un objeto de entrada de tipo '`NamedElement`' se transforma en un objeto de salida '`JavaElement`' definiendo las propiedades 'name' y 'visibility', en este caso tal como vienen en el objeto de entrada.

```
-- Regla abstracta para elementos que extienden a Element
abstract rule Element {
from
        e: UML2!NamedElement
to
        s: JAVA!JavaElement (
                name <- e.name,
                visibility <- e.visibility    ) }
```

```
-- Regla para la transformación de Package a Package
rule Package2Package extends Element{
        from e : UML2!Package (e.isPackageNoStereotype())

        to s : JAVA!Package (
            name        <- e.getName(),
                owner             <- e.nestingPackage,
                javaElements    <- e.packagedElement,
                ownedPackage <- e.nestedPackage ,
                namespace       <- n
          ),

                n: JAVA!Namespace(
                        name        <- e.getExtendedName(),
                        elementImport <- e.getElementImports(),
                        packageImport <- e.getPackageImport()
                ) }


-- Regla abstracta para elementos que extienden a TypedElement2Property
abstract rule TypedElement2Property extends Feature2Feature{
        from
                e : UML2!TypedElement
        to
                s : JAVA!Property(
                type   <- e.type,
                multiplicityElement <- mult
        ),

        mult: JAVA!MultiplicityElement(
                isUnique  <- e.isUnique,
                isOrdered <- e.isOrdered,
                upper     <- e.upper,
                lower     <- e.lower,
                owner     <- s
        )
}

-- Regla para la transformación de StructuralFeature a Field
rule StructuralFeature2Field extends TypedElement2Property{
        from e : UML2!StructuralFeature
        to s : JAVA!Field (
                ownerField     <- e.owner,
                -- LA VISIBILIDAD DE LOS CAMPOS SE PASAN A PRIVATE
                visibility     <- #private
                ) }
```

La regla **'Package2Package'** es una **'Matched Rule'**, la cual vimos que su llamado es implícito y automático para cada elemento que matchee con el objeto declarado de entrada para la regla. Genera un número y tipo de objetos destinos, los cuales son determinados en la transformación. El nombre de cada 'Matched Rule' debe ser único fichero ATL. Esta regla transforma los objetos 'Package', como particularidad se indica que matchea 'solo' con los elementos de tipo 'Package' pero que no estén estereotipados, para determinarlo se usa una función 'helper' definida en el fichero 'U2Helpers.atl' y se indica en la línea de entrada de la regla, aplicada a la variable e que representa el objeto fuente:

```
from e : UML2!Package (e.isPackageNoStereotype()).
```

Los 'Helpers' están resaltados con color naranja.

La regla **'StructuralFeature2Field'** también es una 'Matched Rule' que extiende de la regla abstracta **'TypedElement2Property'**, transforma los objetos **'StructuralFeature'** (Property) y define que el objeto de salida 'Field' tendrá la particularidad de tener la visibilidad 'private'.

```
-- Regla para la transformación de BehavioralFeature a Method
rule BehavioralFeature2Method extends Feature2Feature{
        from e : UML2!BehavioralFeature
        to s : JAVA!Method (
          ownerMethod   <- e.owner,

           returnType    <- if(not e.getReturnType().oclIsUndefined()) then e.getReturnType().type else
e.getReturnType() endif,
          parameters    <- e.ownedElement       )

}


-- Regla para la transformación de Interface a Interface
rule Interface2Interface extends Classifier2Type{
        from e : UML2!Interface
        to s : JAVA!Interface (
                redefinedInterface <- e.redefinedInterface,
                methods         <- e.getMethods(),
                name            <- if thisModule.inTypeElements->includes(e) then e.name else  e.name
+'Interface' endif
        ) }
```

```
-- Regla para la transformación de Class a JavaClass
 rule Class2JavaClass extends Classifier2Type{
        from e : UML2!Class(e.isClassNoStereotype())
        to s : JAVA!JavaClass (
                superClass  <- e.generalization ->collect(c | c.general)-> asSequence() -> first(),

                nestedType  <- e.nestedClassifier,
                methods     <- e.getMethods()-> union((e.getAttributes()-> collect(a|
thisModule.StructuralFeature2Getter(a))) -> union(e.getAttributes()-> collect(a|
thisModule.StructuralFeature2Setter(a)))) ,

                isPersistent <- e.isPersistentClassElement(),
                constructor <- e.getConstructorMethods(),
                implemented <- e.clientDependency -> select(a|a.oclIsKindOf(UML2!InterfaceRealization))
                                                        -> collect(c | c.contract)     )
        do{
         for(elem in s.methods) {
                        elem.owner <-s;
                        elem.ownerMethod <-s; }
         for(elem in s.implemented){
                        s.namespace.elementImport <-
                       s.namespace.elementImport.append(elem.namespace); }         }
}



---- Regla para la transformación de Association (Relaciones)
 rule Association2Association {
        from e : UML2!Association(not e.oclIsTypeOf(UML2!AssociationClass) )
        to s : JAVA!Association (
                name            <- e.name,
                package         <- e.package,
                qualifiedName    <- e.getQualifiedName(),
                isAbstract       <- e.isAbstract,
                isFinal         <- e.isLeaf,
                associationFields <- e.getAssociationEnds(),
            namespace       <- n

        ),
        n: JAVA!Namespace(
                        name <- e.getQualifiedName(),
                        elementImport <- e.getElementImports()
                )   }
```

Aunque son más simples y eficientes las reglas **'matched rules'**, muchas veces necesitamos realizar tareas más elaboradas, como por ejemplo crear parte de un objeto de salida, procesando datos del objeto de entrada y personalizando el resultado, y esto además se puede realizar más de una vez y para distintos contextos de entrada, entonces para ello tenemos las reglas llamadas 'called rules' que son ejecutadas por el llamado de alguna otra regla y no se activan por matcheo. Ejemplo de 'called rules' son:

- **Lazy rules** se pueden aplicar varias veces por cada partido.

- **Unique lazy rules** se aplican como máximo una vez por cada partido, y sólo si se referencia de otras reglas.

Por ejemplo las siguientes dos reglas son de tipo Lazy rules:

```
-- Regla para la transformación en un Method getter a partir de un Field
lazy rule StructuralFeature2Getter{
        from e : UML2!Element
        to sget : JAVA!Method (
            name       <- 'get' + e.name.firstToUpper(),
           returnType <- if(e.oclIsTypeOf(UML2!Class))then e else e.type endif,
                  visibility <-#public,
                  isStatic   <- false,
                  isFinal    <- false
                  )   }

-- Regla para la transformación en un Method setter a partir de un Field
lazy rule StructuralFeature2Setter {
        from e : UML2!Element
        to   sset : JAVA!Method (
                 name       <- 'set'+ e.name.firstToUpper(),
                 parameters <- param,
                  visibility <-#public,
                  isStatic   <- false,
                  isFinal    <- false   )          ,

        param: JAVA!Parameter(
                   name  <- e.name,
                   owner <- sset,
                   type  <- if(e.oclIsTypeOf(UML2!Class)) then e else e.type endif,
                   multiplicityElement <- mult ),

        mult: JAVA!MultiplicityElement(
                  owner       <- param,
                  isUnique    <- true,
                  isOrdered <- false,
                  upper       <- 1,
                  lower       <- 0  )    }
```

- **Elementos estereotipados:**

Ahora comenzamos a mostrar la parte de la transformación de los elementos no estereotipados, en este caso las transformaciones son más complejas y elaboradas, ya que por la mayoría de los objetos fuentes se derivan varios objetos destino, armamos la estructura de árbol derivada del 'package principal' y creamos todas las clases e interfaces necesarias para darle forma a la arquitectura de servicios de Spring. Se utilizan más reglas de tipo 'lazy y unique lazy rules'.

Mostraremos los detalles más distintivos de esta parte de la transformación:

```
-- Regla para la transformación de PackagePrincipal a Package
rule PkPrincipal2Package extends Element{
        from e : UML2!Package (e.isPackagePrincipal())

        using {
           ncommon : String = e.getName()+'_common';
           nscommon: String = e.getExtendedName()+'.'+ncommon;
           nserver : String = e.getName()+'_server';
           nsserver: String = e.getExtendedName()+'.'+nserver;    }

        --PAQUETE PRINCIPAL
        to s : JAVA!Package (
                name            <- e.getName(),
                namespace       <- n,
                owner           <- e.nestingPackage,
                ownedPackage <- e.nestedPackage      ),

           n : JAVA!Namespace(
                name <- e.getExtendedName(),
                packageImport <- e.getPackageImport()        ) ,

        --//**************PACKAGE COMMON ****************************************/

        --PACKAGE COMMON DENTRO DE PRINCIPAL
          s_common: JAVA!Package (
                name          <- ncommon,
                namespace     <- n_common,
                owner         <- s
               ),

               n_common : JAVA!Namespace(
                        name <- nscommon
               ) ,
```

```
-- PACKAGE MODEL DENTRO DE COMMON
  s_comMODEL: JAVA!Package (
            name              <- 'model',
            namespace      <- n_comMODEL,
            owner             <- s_common,
            ownedPackage <- e.getPackageModels()-> collect(a| thisModule.createPackage(a))   ),
  n_comMODEL: JAVA!Namespace(
            name <- nscommon+'.model'   ) ,

-- PACKAGE SERVICE DENTRO DE COMMON
  s_comSERVICE: JAVA!Package (
            name            <- 'service',
            namespace    <- n_comSERVICE,
            owner           <- s_common),
  n_comSERVICE: JAVA!Namespace(
            name <- nscommon+'.service'  ) ,

-- PACKAGE COMMON_DAO DENTRO DE COMMON
   s_comDAO: JAVA!Package (
            name        <- 'dao',
            namespace    <- n_comDAO,
            owner       <- s_common          ),
   n_comDAO: JAVA!Namespace(
            name <- nscommon+'.dao'      ),
```

Definimos los paquetes ‘model’, ’services’ y ’dao’ dentro del paquete ‘common’, de la misma manera se define el package ‘filter’.
A continuación se muestra la creación de algunas de las clases abstractas e interfaces de estos paquetes.

```
    abstracServImpl: JAVA!JavaClass(
                    name    <- 'AbstractService',
                    package <- s_comimplSer,
                    qualifiedName <- s_comimplSer.namespace.name+'.AbstractService',
                    isAbstract<- true,
                    namespace <- n_abstracServImpl,
                    fields    <- p       ),
            --creo el namespace del AbstractService
            n_abstracServImpl: JAVA!Namespace(
                 name        <- s_comimplSer.namespace.name+'.AbstractService',
                elementImport <- n_abstracIDAO      ) ,
            -- agrego a AbstractService un campo abstracDAO, privado a la clase
            p : JAVA!Property(
                    name      <- 'abstractDAO',
                    visibility <- #private,
                    type      <- abstracIDAO,
                    multiplicityElement <- mult1
     ),
```

```
        abstracGenServ: JAVA!Interface(
                name              <- 'GenericServiceInterface',
                package           <- s_comSERVICE,
                qualifiedName <- s_comSERVICE.namespace.name+'.GenericServiceInterface',
                namespace       <- n_abstracGenServ,
                isAbstract        <- true   ),
        n_abstracGenServ: JAVA!Namespace(
                 name <-  s_comSERVICE.namespace.name+'.GenericServiceInterface'
        ) ,
```

Y de esta misma manera seguimos creando todos los paquetes y clases abstractas e interfaces que definen el comportamiento común y abstracto del sistema, y conforman el package 'COMMON', a medida de la implementación propuesta en inciso anterior.

```
--//**************PACKAGE SERVER ****************************************/
  --PACKAGE SERVER DENTRO DE PRINCIPAL
      s_server : JAVA!Package (
                name            <- nserver,
                 namespace    <- n_server,
                 owner          <- s        ),
         n_server: JAVA!Namespace(
                name <- nsserver         ) ,
```

A partir del package SERVER, se desarrollan las clases de implementación de DAOS, SERVICES Y FILTERs, por una cuestión de simplificación en el informe mostraremos solo la rama de DAOS, el resto se implementa de manera similar y se puede ver en detalle en el archivo 'U2J.atl'.

```
      -- PACKAGE DAO DENTRO DE SERVER
         s_serDAO: JAVA!Package (
          name          <- 'dao',
          namespace    <- n_serDAO,
          owner          <- s_server      ),
            n_serDAO: JAVA!Namespace(
                 name <- nsserver+'.dao'        ) ,

      -- PACKAGE IMPL DENTRO DE DAO
         s_implDAO: JAVA!Package (
           name       <- 'impl',
          namespace    <- n_implDAO,
          owner       <- s_serDAO ,
          javaElements <- e.getAllDAOClassElement()-> collect(a| thisModule.createDAO_impl(a))       ),
      -- PACKAGE FILTER DENTRO DE DAO
         s_implFILTER: JAVA!Package (
                 name       <- 'filter',
                 namespace    <- n_implFILTER,
                 owner       <- s_implDAO ,
                 javaElements <- e.getAllPersistentClassElement()-> collect(a|
                                                                  thisModule.createFilter_imp(a))
```

En la línea javaElements <- e. getAllDAOClassElement ()-> collect(a| thisModule. createDAO _imp(a)) , creamos los objetos DAOS y se los asignamos al package 'dao.impl' como elementos propios. La función 'helper' '**getAllDAOClassElement**' obtiene el subconjunto de elementos del modelo fuente, que están estereotipados <<DAO>> y se crea un objeto 'DAO' por cada uno, mediante la called rule 'createDAO_imp(a)'.

En la línea javaElements <- e.getAllPersistentClassElement()-> collect(a| thisModule.createFilter_imp(a)) , creamos los objetos filtros y se los asignamos al package 'filter' como elementos propios. La función 'helper' '**getAllPersistentClassElement**' obtiene el subconjunto de elementos del modelo fuente, que están estereotipados <<PersistentClass>> y se crea un objeto 'filtro' por cada uno, mediante la called rule 'createFilter_imp(a)'.

En el sector **do** de la transformación inicializamos parte de los objetos de la siguiente manera:

```
do {

 -- Seteo como owner a los subPackages de s_comMODEL
  for(elem1 in s_comMODEL.ownedPackage) {
        elem1.owner <- s_comMODEL;
       thisModule.createOwnedNamespace( s_comMODEL, elem1);

  for(elem2 in elem1.javaElements) {
        elem2.package<-elem1;
        elem2.qualifiedName <- elem1.namespace.name+'.'+elem2.name;
       elem2.namespace.name<- elem1.namespace.name+'.'+elem2.name;

       -- actualizo los elementImport de las clases con los namespace de las interfaces que implementan
       if(elem2.isJavaClass()){
            for(elem in elem2.implemented){
                  elem2.namespace.elementImport <-
                  elem2.namespace.elementImport.append(elem.namespace);
                              }     }     }      }

-- seteo a todas las clases ParamFilters la superClass ParamFilterAbstract
 for (elem3 in s_comPARAMETER.javaElements){
        elem3.superClass             <- paramFilterAbstract;
        elem3.namespace.name   <- s_comPARAMETER.namespace.name+'.'+elem3.name;
        elem3.namespace.elementImport <-
       elem3.namespace.elementImport.append(paramFilterAbstract.namespace);
 }

 -- seteo a todas las clases Filters la superClass FilterAbstract
 for (elem4 in s_implFILTER.javaElements){
        elem4.superClass                    <- filterAbstract;
        elem4.namespace.name             <- s_implFILTER.namespace.name+'.'+elem4.name;
        elem4.namespace.elementImport <- elem4.namespace.elementImport.append(n_filterAbstract);
         } }
```

A continuación y para finalizar mostramos las reglas createFilter_imp(a) y createDAO_imp(a), como explicamos estas reglas son **unique lazy rules** porque se debe crear un objeto salida por cada elemento 'a' de entrada. De esta manera creamos los filtros, uno para cada elemento persistente, y los daos, uno para cada elemento estereotipado <<DAO>>.

```
--Regla para crear la clase Filtro a partir de una clase persistente
unique lazy rule createFilter_imp  {

 from
          e       :          UML2!Class
 to
      f : JAVA!JavaClass (
             name       <- e.name+'_Filter',
                    methods     <- (e.getAttributes()-> collect(a| thisModule.StructuralFeature2Filter(a))) ,
                    namespace   <- n

                  ),
          n: JAVA!Namespace(
                        name <- e.name+'_Filter'          )
         do{
                    f.namespace.elementImport <-
f.namespace.elementImport.append(thisModule.createParametroFilter(e).namespace);
                    f.namespace.elementImport <-f.namespace.elementImport.append(e.namespace);
                    for(elem in f.methods) {
                              elem.owner <-f;    } }
         }

-- Regla para crear la clase DAO y la interface IDAO correspondiente
unique lazy rule createDAO_impl  {

          from
                    e  :     UML2!Class(e.isDAO())
          to
                    dimpl : JAVA!JavaClass (
                              name          <- e.name+'_DaoImpl',
                              implemented  <- idao,
                              fields          <- e.getAttributes(),
                              methods     <- e.getMethods(),
                              namespace   <- n    ),
                    n: JAVA!Namespace(
                                        name <- e.name+'_DaoImpl'      ),
                    idao: JAVA!Interface (
                                        name<-  'I'+e.name+'_DAO',
                                        isAbstract <- true,
                                        namespace   <- ni),
```

```
do{

      for(elem in e.getUsages()) {

        thisModule.createMethodsDAO(idao,elem);
        dimpl.namespace.elementImport <-
dimpl.namespace.elementImport.append(thisModule.createParametroFilter(elem).namespace);
        dimpl.namespace.elementImport <-
dimpl.namespace.elementImport.append(thisModule.createFilter_imp(elem).namespace);
        dimpl.namespace.elementImport <-dimpl.namespace.elementImport.append(elem.namespace); }

       for(elem in dimpl.methods) {
                  elem.owner  <- dimpl;
                   elem.ownerMethod <- dimpl;      }

       for(elem in dimpl.fields) {
                           elem.owner <- dimpl;
                  }
        }
```

Las reglas que pintamos de azul son llamadas para crear más elementos, en el caso de 'createMethodsDAO(idao, elem)' se llama para la creación de los métodos de la clase dao correspondiente, los mismos son creados y relacionados con la clase.

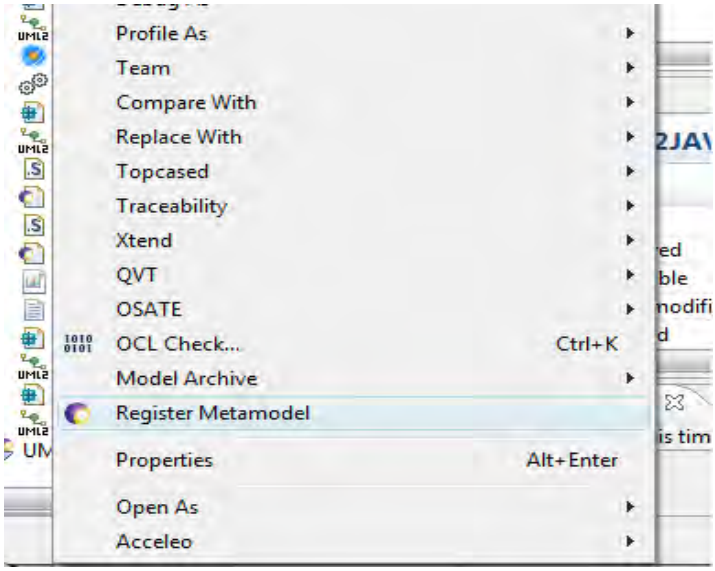
Bien hemos mostrado casi todas las reglas de transformación escritas en nuestro archivo, a grandes rasgos en la primer parte se encuentran las transformaciones de las clases no estereotipadas y se podría decir que son las más sencillas, luego tenemos las reglas que involucran a los objetos estereotipados y como habrán podido observar derivan al armado de todo el esqueleto de la arquitectura deseada. Debemos jugar con las reglas **lazy** porque en una misma regla por matcheo, se derivan la creación de varios objetos relacionados y como por ejemplo en el caso de las reglas 'StructuralFeature2Getter(a)' y 'StructuralFeature2Setter(a)' que sirven para crear los métodos getter y setter a partir de un atributo particular, son invocadas varias veces y desde más de una regla.

Ahora vamos a mostrar cómo ejecutar esta primer parte de la transformación y cuál es el resultado obtenido.

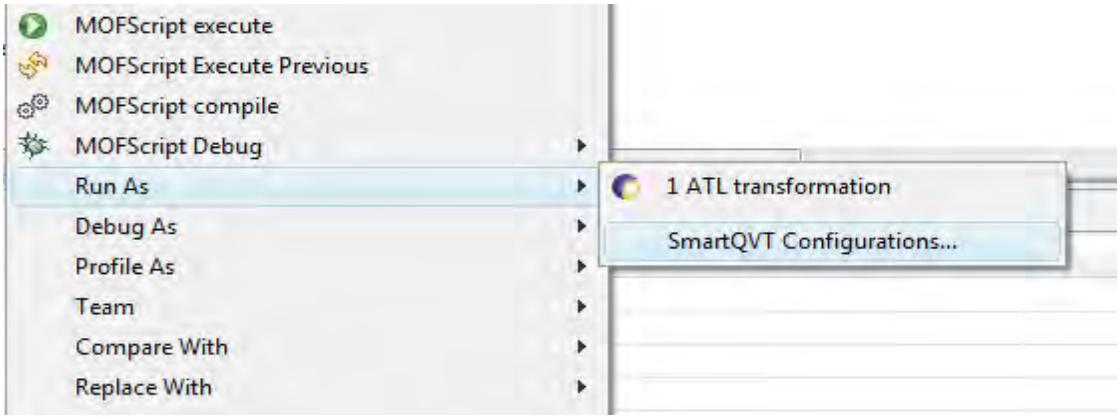
- ## 4.3.3.2.3 Probamos la transformación y mostramos el resultado obtenido

Ejecutando la transformación de modelo a modelo, del PIM al PSM:

Antes que nada debemos asegurarnos de tener los metamodelos que vamos a usar registrados y publicados en el ambiente. Para ello, nos paramos en este caso sobre el metamodelo 'JAVA.ecore', presionamos el botón derecho del mouse y ahí seleccionamos la opción 'Register Metamodel' y listo.

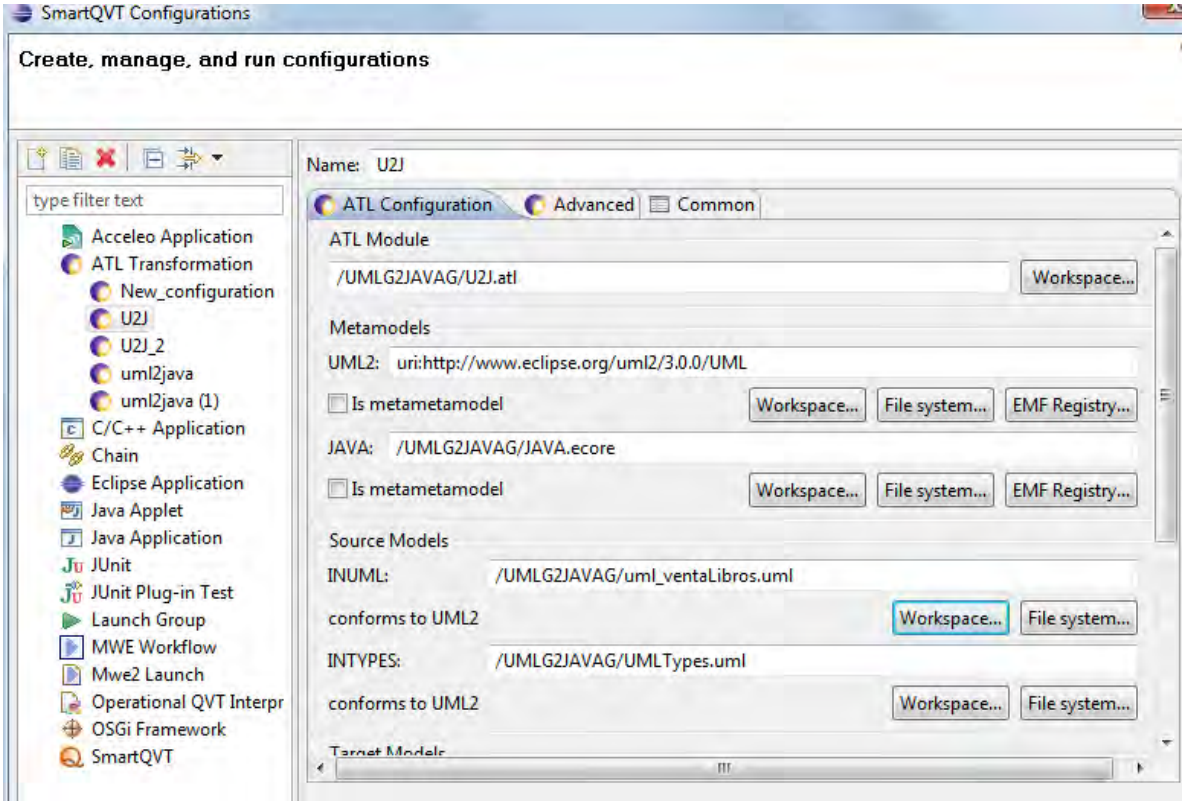


Ahora debemos configurar la ejecución del archivo 'U2J.atl', para ello hacemos click en el botón derecho sobre el archivo , hacemos click en 'Run As' y seleccionamos la opción 'SmartQVT Configurations':

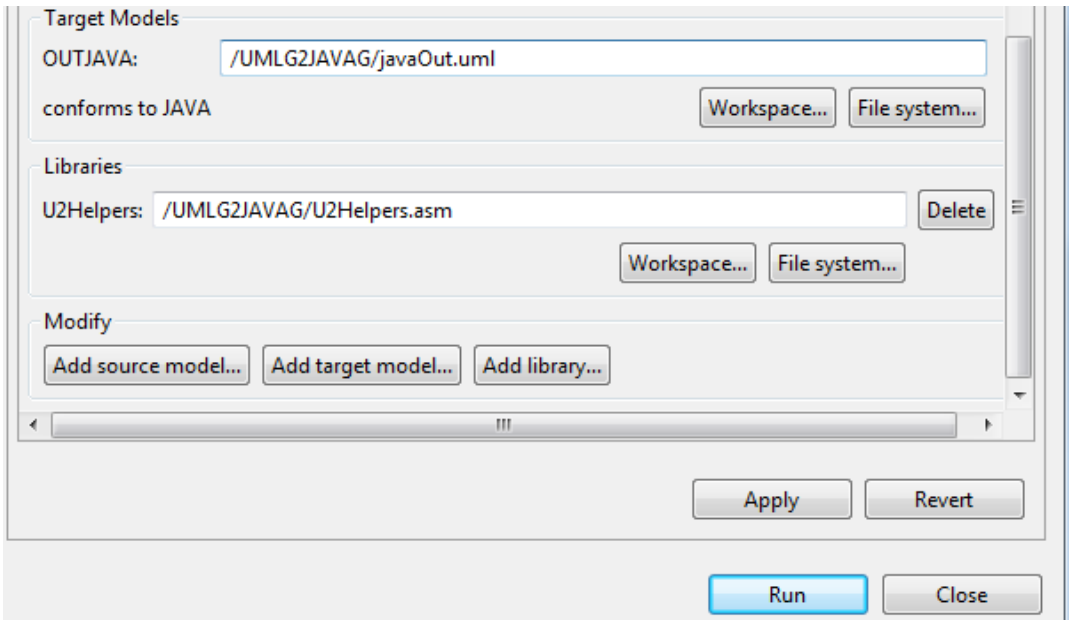


Nos abre una ventana de configuración:

En ATL Module seteamos el archivo '.atl' que deseamos ejecutar, luego en la parte titulada 'Metamodels' buscamos las URI y cargamos los metamodelos involucrados en la transformación, en este caso UML2 y JAVA como los nombramos en el encabezado de la transformación, luego en el sector 'Source Models' cargamos de la misma manera los modelos source, de entrada de la transformación 'uml_ventaLibros.uml' y 'UMLTypes.uml'.



Y en el sector 'Target Models' cargamos de la misma manera el nombre y uri del modelo destino que deseamos crear. En este caso 'JavaOut.uml' (el archivo destino puede ser también un archivo '.xml').
Y en el sector titulado 'libraries' se cargan las librerías a utilizar, en nuestro caso cargamos los helpers 'U2Helpers.atl'.



Finalmente ejecutamos la transformación apretando el botón 'Run'.

Como resultado obtenemos un archivo '.uml' conformado por el esqueleto deseado de la aplicación y coincidente con el modelo objetivo mostrado en la figura 4.5, página 119.
Este modelo resultante de la ejecución del PIM al PSM, llamado 'javaOut.uml', será el modelo source, en la siguiente parte de la transformación de modelo a texto, para finalmente obtener nuestro proyecto como clases java.

A continuación explicaremos la segunda parte de la transformación de PSM a Código.

4.3.4 Desarrollo de transformación de modelo a texto, derivando a código Java

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

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

1- Crear un proyecto

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

2- Especificar el metamodelo de entrada

Como metamodelo de entrada pueden usarse algunos definidos ya por los proveedores de la herramienta, o definir metamodelos nuevos. En nuestro ejemplo vamos a utilizar el metamodelo de JAVA definido anteriormente

3 - Escribir la transformación MOFScript

Al tener organizado el modelo de entrada con la estructura definida en la figura 4.3, página 109 podemos no solo generar las clases java con sus propiedades y métodos, sino también los archivos hbm para las clases persistes y los archivos de configuración con los daos y los servicios correspondientes

A continuación mostramos la transformación realizada para generar los archivos hbm de los objetos persistes y los archivos de configuración spring-daos.xml y spring-servicios.xml.

Para esto cuando comenzamos nuestra transformación creamos los encabezados de nuestros archivos xml de la siguiente manera:

```
createConfigs () {
      createContext ()
     createHibernate ()
openServicios ()
   openDaos ()
   openMySQL ()
}

openDaos () {
          file daos(proyecto + "/config/spring-daos.xml")
          println('<?xml version="1.0" encoding="UTF-8"?>')
          println('<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
          "http://www.springframework.org/dtd/spring-beans.dtd">')
          println('<beans>')
}

openServicios () {
  file servicios (proyecto + "/config/spring-servicios.xml")
  println('<?xml version="1.0" encoding="UTF-8"?>')
  println('<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">')
  newline(1)
  println('<beans>')
}
```

Luego cuando generamos una clase miramos el nombre y la propiedad isPersistent de la misma para saber si esta representa un objeto persiste, un servicio o un dao y de esta manera incluirla en el archivo xml correspondiente o generar el hbm

```
uml.JavaClass::getClass () {
  if (self.isPersistent)  {
          self.createHbm()
          self.addMySQL()
  }
  if (self.name.endsWith("_DaoImpl"))
          self.addDao()
  if (self.name.endsWith("_ServiceImpl"))
          self.addService()
          self.createJava()
 }

uml.JavaClass::addDao()  {
file daos(proyecto + "/config/spring-daos.xml")  tab(1)
println('<bean id="' + self.name.substringBefore("_DaoImpl").firstToLower()
+ '" class="' + self.qualifiedName + '">')          tab(2)
println('<property name="hibernateTemplate">')  tab(3)
println('<ref bean="hibernateTemplate"/>')      tab(2)
println('</property>')                           tab(1)
```

```
println('</bean>')
}

uml.JavaClass::addService()      {
 file servicios (proyecto + "/config/spring-servicios.xml")
 newline(1)  tab(1)
 println('<bean id="' + self.name.firstToLower()
            + 'Target" class="' + self.qualifiedName + '">')
 self.associationFields->forEach(c:uml.JavaClass)
{  tab(2)
 println('<property name="' + c.name.substringBefore("_DaoImpl").firstToLower()
  + '">')  tab(3)
  println('<ref bean="' + c.name.substringBefore("_DaoImpl").firstToLower() + '"/>')
  tab(2) println('</property>')
  tab(1) println('</bean>')
}
newline(1)       tab(1)
println('<bean id="' + self.name.firstToLower() + '" parent="abstractTransactionDefinition">')   tab(2)
println('<property name="proxyInterfaces">')   tab(3)
println('<list>')            tab(4)
println('<value>' + self.qualifiedName + '</value>')       tab(3)
println('</list>')                    tab(2)
println('</property>')              tab(2)
println('<property name="target">')              tab(3)
println('<ref bean="' + self.name.firstToLower() + 'Target"/>')   tab(2)
println('</property>')              tab(1)
println('</bean>')
}


uml.JavaClass::createHbm() {
    String  fileName  =  self.package.getDirectoryHibernate()  +  "/"  +  self.name.removeInvalidCharacters()    +
    ".hbm.xml"
    file (fileName)
    println('<?xml version="1.0"?>')
    print('<hibernate-mapping package=')
    println('"' + self.package.getPackageHibernate() + '">')            tab(1)
    if (!self.superClass.isEmpty())
     println('<union-subclass abstract="false" catalog="catalog" dynamic-insert="true" dynamic-update="true"
entity-name="' + self.name + '" extends="' + self.superClass.name + '" lazy="true|false" name="' + self.name + '"
node="element-name" persister="' + self.name + '" proxy="ProxyInterface" schema="schema" subselect="SQL
expression" table="' +  self.name + 's">')
    else
      println('<class name="'+self.name+'" table="' +  self.name + 's">') tab(2)
      println('<id name="id" type="long" unsaved-value="0">')      tab
      println('<column                    name="id'+self.name+'"sql-type="BIGINT"                    not-null="true"/>')
    println('<generator class="native"/>')       tab(2)
     println('</id>')
     self.createPropertiesHbm(fileName)
```

```
        if (!self.superClass.isEmpty())
        {
                self.superClass -> forEach(c:uml.JavaClass){
                        c.createPropertiesHbm(fileName)
        }       }       tab(1)
        if (!self.superClass.isEmpty())
                println('</union-subclass>')
        else
                println('</class>')
                println('</hibernate-mapping>')
  }
```

Una vez que transformamos todas las clases debemos cerrar los archivos de la siguiente manera:

```
        closeDaos() {
          file daos(proyecto + "/config/spring-daos.xml")
          println('</beans>')
        }

        closeServicios () {
          file servicios (proyecto + "/config/spring-servicios.xml")
          println('</beans>')
        }
```

Como resultado de estas transformaciones obtenemos los siguientes archivos. Ejemplificamos con la clase ClientAccount:

Spring-daos.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
       "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<bean id="clientAccountDAO" class="Model.uml_ventaLibros.Pk_adminVentaLibros.
Pk_adminVentaLibros_server.dao.impl.ClientAccountDAO_DaoImpl">
            <property name="hibernateTemplate">
                  <ref bean="hibernateTemplate"/>
            </property>
</bean>
```

Spring-servicios.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">


<beans>

<bean id="clientAccountService_ServiceImplTarget"
class="Model.uml_ventaLibros.Pk_adminVentaLibros.
Pk_adminVentaLibros_server.service.impl.ClientAccountService_ServiceImpl">

<bean id="clientAccountService_ServiceImpl"  parent="abstractTransactionDefinition">
 <property name="proxyInterfaces">
  <list>
 <value>Model.uml_ventaLibros.Pk_adminVentaLibros.
Pk_adminVentaLibros_server.service.impl.ClientAccountService_ServiceImpl</value>
 </list>
 </property>
      <property name="target">
        <ref bean="clientAccountService_ServiceImplTarget"/>
      </property>
</bean>
…….
</beans>
```

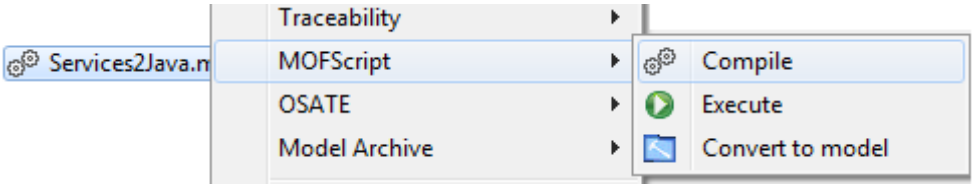
Book.hbm.xml

```
<?xml version="1.0"?>
<hibernate-mapping package="Uml_ventaLibrosProject.src.Model.uml_ventaLibros.
Pk_adminVentaLibros.Pk_adminVentaLibros_server.hibernate.model.Pk_model">

<class name="Book" table="Books">
<id name="id" type="long" unsaved-value="0">
            <column name="idBook" sql-type="BIGINT" not-null="true"/>
            <generator class="native"/>
</id>
<property name="listPrice" column="LISTPRICE" not-null="false" />
<property name="edition"   column="EDITION" not-null="false" />
<property name="bookSpec"  column="BOOKSPEC" not-null="false" />
   <joined-subclass name="PrintedBook" table="CREDIT_PAYMENT">
      <key column="idBook "/>
         <property name="shippingWeigth" column="SHIPPINGWEIGTH" not-null="false" />
         <property name="productDimensions" column="PRODUCTDIMENSIONS"
                  not-null="false" />
         <property name="stock" column="STOCK" not-null="false" />
       </joined-subclass>
      <joined-subclass name="DigitalBook" table="CASH_PAYMENT">
                  <key column="idBook "/>
            <property name="size" column="SIZE" not-null="false" />
            <property name="url" column="URL" not-null="false" />
       </joined-subclass>
</class>
</hibernate-mapping>
```

- ## 4.3.4.1 Probamos la transformación a código verificando el esqueleto del sistema obtenido.

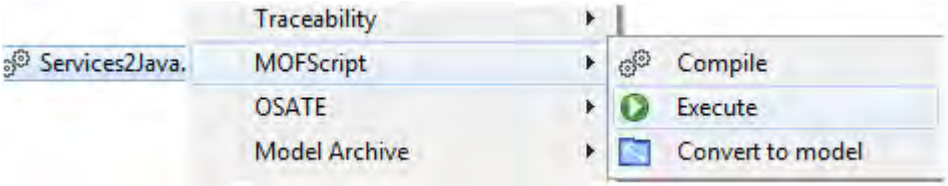
4 - Compilar

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



5 - Ejecutar la transformación

La transformación se ejecuta mediante una opción en el menú contextual sobre el archivo de transformación, como se muestra en la siguiente figura. Al momento de ejecutarla, se debe ingresar el archivo que representa al modelo.
La salida de la transformación puede imprimirse en la consola, o se puede especificar dentro de archivos, como es este caso, donde se genera un archivo por cada una de las clases JAVA generada, uno por cada archivo de configuración (xml) y uno por cada clase persistente (hbm.xml)



Como resultado obtenemos un conjunto de archivos java y hbm.xml organizados según la estructura de packages definidos en el modelo de entrada. Además se obtiene un directorio config el cual posee todos los archivos de configuración del proyecto.

4.3.5 Desarrollo del plugin de ejecución completa de las dos transformaciones

Una vez que tenemos hechas las dos transformaciones debemos buscar la manera de automatizar el proceso para poder realizar todas las acciones en un solo paso de manera transparente al usuario. Para esto vamos a utilizar la herramienta ANT.

Ant es un programa cuya función es organizar todo el proceso de generación de código. En lugar de escribir instrucciones a manera de scripts, lo que se hace, es indicarle "tareas" (Ant Tasks) a realizar sobre un proyecto, a través de un archivo XML (build.xml).

Algunas ventajas de utilizar Ant son:

- Está hecho en java, así que es portable. Si hacemos nuestro fichero **build.xml** y tenemos instalado ant en windows, en linux o en imac, funciona todo bien sin necesidad de tocar nada.
- Nos permite definir fácilmente varias tareas a hacer y fijar las dependencias entre ellas.
- Casi todos los entornos de desarrollo integran o tienen posibilidad de integrar ant (eclipse por ejemplo). De esta forma, las tareas que hayamos definido para línea de comandos de ms-dos/shell de linux, las tenemos accesibles directamente desde el entorno de desarrollo.

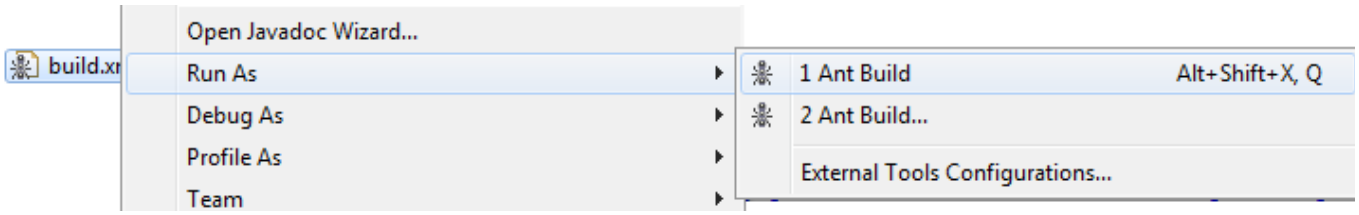
A continuación mostramos el archivo build.xml realizado para ejecutar la transformación completa.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project name="UMLG2JAVAG" default="run" basedir=".">

   <!-- Definimos los path de los archivos a utilizar -->
   <property name="pathIN" value="../UMLG2JAVAG/uml_ventaLibros.uml" />
   <property name="pathOUT" value="../UMLG2JAVAG/javaOutLibrosAnt.uml" />

   <property name="pathINTYPES" value="../UMLG2JAVAG/UMLTypes.uml" />
   <property name="pathUService" value="../plugServices/UService.uml" />
   <property name="pathJAVA" value="../UMLG2JAVAG/JAVA.ecore" />
   <property name="ATL_COMPILER" value="atl2006" />
   <property name="atl.launcher" value="EMF-specific VM"/>

   <target name="run" >
         <!-- Cargamos los modelos y metamodelos necesarios para la
              transformación -->
         <atl.loadModel name="UML2" metamodel="MOF"
                        nsUri="http://www.eclipse.org/uml2/3.0.0/UML" />
         <atl.loadModel name="UService" metamodel="MOF" path="${pathUService}" />
         <atl.loadModel name="INUML" metamodel = "UML2" path="${pathIN}" />
```

```
            <atl.loadModel name="INTYPES" metamodel = "UML2" path="${pathINTYPES}" />
            <atl.loadModel name="JAVA" metamodel="MOF" nsUri="http://JAVA"
                           modelhandler="EMF" />

            <atl.launch path="../UMLG2JAVAG/U2J.asm" >
                  <!-- Incluimos la librería U2Helpers -->
                  <library name="U2Helpers" path="U2Helpers.asm"/>

                  <!-- Definimos nuestros modelos de entrada y salida -->
                  <inmodel model="INUML" name="INUML" />
                  <inmodel model="INTYPES" name="INTYPES" />
                  <outmodel model="OUTJAVA" name="OUTJAVA" metamodel="JAVA"
                            path="${pathOUT}" >
                  </outmodel>
                  <option name="IS_REFINING" value="false" />
                  <option name="OPTION_CLEAR" value="true"/>
                  <option name="OPTION_DERIVED" value="true"/>

                  <option name="supportUML2Stereotypes" value="true"/>
                  <option name="printExecutionTime" value="true"/>
                  <option name="allowInterModelReferences" value="false"/>
                  <option name="step" value="false"/>
            </atl.launch>

            <!-- Guardamos el modelo de salida -->
            <atl.saveModel model="OUTJAVA" path="${pathOUT}" />

            <!-- Ejecutamos la transformación a texto -->
            <mofscript template="Services2Java.m2t"
                  templatesDir="../UMLG2JAVAG/"
                  outputdir="..">
                        <model metamodel="${pathJAVA}" model="${pathOUT}" />
            </mofscript>
      </target>
</project>
```

- ## 4.3.5.1 Probamos el plugin de ejecución

El plugin se ejecuta mediante una opción en el menú contextual sobre el archivo build.xml, como se muestra en la siguiente figura.



Como resultado obtenemos: un archivo uml con la estructura definida en la figura 4.3, un conjunto de archivos java y hbm.xml organizados según la estructura de packages definidos en el modelo de entrada, y un directorio config el cual posee todos los archivos de configuración del proyecto.

5 Conclusiones

La experiencia personal adquirida durante la realización de este proyecto, ha sido especialmente enriquecedora. Realmente resulta gratificante, ver que hemos llegado al final del proyecto, cumpliendo todos los objetivos trazados en un principio.
Durante el desarrollo de la tesis, adquirimos una serie de conocimientos, que hasta ahora resultaban desconocidos o muy novedosos, y que al final, se convirtieron en conceptos que no olvidaremos fácilmente.
En líneas generales, la realización de este proyecto ha resultado en una experiencia positiva y provechosa a nivel personal, aportándonos conocimientos muy valiosos.

Desde un principio nuestro deseo al realizar la tesis era hacer un proyecto práctico que nos sea de gran utilidad en nuestro trabajo profesional. Nos resultaron interesantes los beneficios que ofrece el paradigma MDA y decidimos explorarlos.
MDA plantea el desarrollo de software a partir del modelo, el cual se basa en la utilización del modelado en UML que es el estándar más utilizado para el diseño de modelos y en proyectar un conjunto de transformaciones entre modelos para obtener como objetivo final un esquema del código del sistema.
Teniendo estas bases, elegimos para nuestro proyecto sistematizar la estructura orientada a la arquitectura de software de servicios, utilizando como herramienta el poderoso framework Spring, el cual ofrece un pegamento sencillo y potente de varias tecnologías mediante el paradigma MVC (Model View Controller). El objetivo se planteó en función de promover la 'correcta' utilización de la arquitectura y el 'buen diseño' de sistemas orientados a servicios, como también y no menos importante la agilización en reducir tiempos del desarrollo del software, contando con varios pasos resueltos de diseño y organización al comenzar la codificación.

Tanto MDA como Spring plantean un esquema modular e independiente para el desarrollo del sistema, pudiendo adoptar de forma flexible varias tecnologías en un mismo proyecto y/o diseñar partes independientes que luego puedan interactuar con otras para complementar funcionalidad. También ambas se pueden desarrollar bajo la plataforma de Eclipse, la cual nos ofrece un amplio conjunto de funcionalidad integrando numerosas herramientas de desarrollo de software mediante la utilización de plugins que son como pequeños módulos que resuelven cierta funcionalidad particular. Siendo además una plataforma de desarrollo de código abierto.

Por lo tanto MDA nos ofreció un conjunto de herramientas muy interesantes para construir un nuevo plugin que permite, a partir de un modelo de clases UML, decorado con un perfil orientado a servicios, construir un conjunto de transformaciones del mismo y llegar a obtener codificado el esqueleto del sistema, con la estructura y la definición de las clases intervinientes para la capa de negocio, de datos, archivos de configuración y la funcionalidad más importante definida explícitamente.

Es así como mediante la utilización de perfiles de UML2, construimos el perfil UServices, que permite identificar en el modelo los objetos que deben persistir en la base de datos, los Servicios para la capa de negocio, los DAOs para la capa de datos y sus relaciones. Con el lenguaje ATL para la transformación de modelo a modelo, resolvemos el paso intermedio a un modelo JAVA orientado a Servicios, donde ya nos queda la estructura armada con los paquetes, nuevas clases definidas y la funcionalidad de las capas de negocios y datos, como también los filtros de consultas de objetos persistentes. Finalmente con MofScript realizamos la transformación a texto, logrando el proyecto codificado y agregando los archivos de mapeo y configuración (utilizando la tecnología del mapeador Hibernate).

De esta manera el sistema queda planteado, como un esqueleto y parte de su funcionalidad, dejando para los programadores completar funcionalidad puntual, reduciendo así errores de diseño y mucho tiempo de desarrollo. Como también logrando que cualquier cambio futuro de diseño sea guiado nuevamente desde el modelo para impactar en el código y así nunca perder la consistencia de información del modelo al código y viceversa.

Como vimos en el correr de la tesis intervinieron en nuestro proyecto la integración de varias tecnologías que MDA nos permite desarrollar de una manera modular. Siendo así muy flexible también para poder desarrollar y derivar nuevas transformaciones y agregar el día de mañana módulos a este proyecto, como puede ser la parte de la vista (View) en la cual no nos detuvimos, para hacerlo cada vez más completo y funcional.

Es así que concluimos felizmente y resultando beneficiosa la dedicación aportada en este tiempo al estudio de MDA, que nos brinda un amplísimo abanico de posibilidades para desarrollar, demostrándonos que finalmente se puede lograr la “Arquitectura dirigida por Modelos”, de forma segura, potente y provechosa para el desarrollo de software cotidiano.

Esperamos de todo corazón haber aportado con este estudio, un granito más para el gran mundo de MDA.

6 Trabajos a Futuro

Por último cabe destacar un punto dedicado a futuras mejoras y evoluciones del proyecto, puesto que como es lógico en el mundo de la informática, un proyecto nunca se acaba totalmente.

Siempre hay alguna cosa que mejorar, ya sea porque se quieren añadir nuevos módulos o porque aparezcan nuevas tecnologías y versiones en el mercado, y como es lógico con el paso del tiempo, la tecnología o versión utilizada para el desarrollo del proyecto quede obsoleto.

Por este motivo, si se quiere seguir dando funcionalidad a la aplicación, hay que actualizar la versión o la tecnología utilizada inicialmente, para adaptarla a tecnologías más recientes, todo esto claro, a largo plazo.

A corto plazo, se van a enumerar algunos de los puntos más importantes que se pueden mejorar:

- ✓ **Optimizar la generación a código JAVA para mantener la trazabilidad con respecto a los cambios realizados. De manera que al modificar el modelo y ejecutar nuevamente las transformaciones se actualicen solo las modificaciones nuevas sin perder los fuentes previos del proyecto que no sufrieron modificación y así mantener la consistencia y la integridad entre código fuente y modelo.**

- ✓ **Realizar un editor gráfico para generar los diagramas uml y que a su vez nos permita visualizar el modelo de salida de una manera más amigable.**

- ✓ **Agregar módulos a la transformación que contengan funcionalidad para las vistas y controladores del MVC utilizados por spring.**

- ✓ **Analizar la posibilidad de hacer reingeniería inversa de manera que el modelo UML se mantenga actualizado luego de hacer modificaciones en el código fuente.**

Éstas son sólo algunas de las posibles mejoras y ampliaciones que se pueden realizar.
Pero como se ha comentado antes, las posibilidades son infinitas, y la mejora de una aplicación informática puede no acabar nunca.

Bibliografía

- ***Arquitectura de Servicios.***
  - *WIKIPEDIA (La enciclopedia libre)* http://es.wikipedia.org/wiki/Arquitectura_orientada_a_servicios
  - *BOX, Don. A brief history of SOAP.* http://webservices.xml.com/pub/a/ws/2001/04/04/soap.html
  - *MOLLER, Anders. SCHAWARTZBACH, Michael I. Interactive Web Services with Java, JSP, Servlets, JWIG, SOAP, WSDL, UDDI.*
  - *Anders. SCHWARTZBACH, Michael I. "Schema Languages".*http://www.brics.dk/ixwt/IXWT_C04c.pdf *In An Introduction to XML and Web Techonologies.*
  - *MOLLER, Anders. SCHWARTZBACH, Michael I. "Chapter 11. Web Services".* http://www.brics.dk/ixwt/webservices.pdf *In An Introduction to XML and Web Techonologies. OGBUJI, Uche. Using WDSL in SOAP applications: An introduction to WDSL for programmers.* http://www-128.ibm.com/ developerworks/library/ws-soap/?dwzone=ws
  - *VASUDEVAN, Venu. A Web Services Primer.* http://webservices.xml.com/pub/a/ws/2001/04/04/webservices/ index.html
  - *W3C. Semantic Web Activity.* http://www.w3.org/2001/sw/
  - *W3C. Simple SOAP.* http://www.w3.org/TR/soap12-part1/
  - *W3C. SOAP Message Transmision Optimization (MTOM).* http://www.w3.org/TR/soap12-mtom/
  - *W3C. Web Services Addredsing (WS-Addressing).* http://www.w3.org/TR/ws-addr-core/
  - *W3C. Web Services Description Language (WSDL).* http://www.w3.org/TR/wsdl20/
  - *W3C. Web Services Choreography Description Language (WS-CDL).* http://www.w3.org/TR/ws-cdl-10/WEBServices Int
- ***Spring Framework.***
  - http://es.wikipedia.org/wiki/Spring_Framework
- ***Hibernate.***
  - http://www.davidmarco.es/tutoriales/hibernate-reference/index.html#tutorial-firstapp-mapping
  - http://www.javatutoriales.com/2009/05/hibernate-parte-1-persistiendo-objetos.html
  - http://www.javatutoriales.com/2009/06/hibernate-parte-4-relaciones-uno-muchos.html
  - http://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/inheritance.html
- ***MDD***
  - May Dehayni y Louis Féraud, "An Approach of Model Transformation Based on Attribute Grammars," in Object- Oriented Information Systems.: Springer Berlin / Heidelberg, 2003.
  - Charles W. Krueger, "Software Reuse," in ACM Comput. Surv. 24 (2), 1992, pp. 131-183.
  - http://es.wikipedia.org/wiki/Model-driven_architecture
  - ***Model-Driven Architecture in Practice****. A Software Production Environment Based on Conceptual Modeling*, Oscar Pastor, Juan Carlos Molina, 2007, ISBN: 978-3-540-71867-3
  - *MDA Distilled, Principles of Model Driven Architecture*, Stephen Mellor, Kendall Scott, Axel Uhl, Dirk Weise, Addison-Wesley Professional, 2004, ISBN 0-201-78891-8
  - *The MDA Journal: Model Driven Architecture Straight From The Masters*, Meghan Kiffer, ISBN 0-929652-25-8
  - *Model Driven Architecture*, Springer-Verlag, ISBN 3-540-28240-8
  - *Model Driven Architecture: Applying MDA to Enterprise Computing*, David S. Frankel, John Wiley & Sons, ISBN 0-471-31920-1
  - *Model Driven Architecture With Executable UML*, Chris Raistrick, Paul Francis, John Wright, Colin Carter, Ian Wilkie Cambridge University Press, ISBN 0-521-53771-1
  - http://www.ecured.cu/index.php/Arquitectura_dirigida_por_modelos
  - JAVA: http://mat21.etsii.upm.es/ayudainf/aprendainf/java/java2.pdf

- ***OCL***
  - http://www.lifia.info.unlp.edu.ar/papers/2003/Becker2003.pdf
  - http://caosd.lcc.uma.es/spl/hydra/documents/PFC_JRSalazar.pdf
  - *Warmer J., Kleppe A. The Object Constraint Language. Second Edition.models ready for MDA. Object Technology. Addison-Wesley, 2003.*
  - *Fowler M. Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999.*
  - *Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Final Adopted Specification, 2005.*
- ***ATL***
  - ***ATL Documentation***
  - User Manual, Starter Guide
  - http://www.eclipse.org/m2m/atl/doc/
  - Open Model CourseWare
  - http://www.eclipse.org/gmt/omcw/resources/chapter10/
  - Ejemplos (unos 100)
    - ATL Transformations
      - http://www.eclipse.org/m2m/atl/atlTransformations/
    - ATL Use Cases
      - http://www.eclipse.org/m2m/atl/usecases/
  - Problemas y soluciones
    - http://wiki.eclipse.org/index.php/ATL_Language_Troubleshooter
- ***UML***
  - *INRIA. (2005, Marzo) ATL TRANSFORMATION EXAMPLE* - Class to Relational. http://www.eclipse.org/m2m/atl/atlTransformations/Class2Relational/ExampleClass2Relational[v00.01].pdf
  - http://osgibundlegenerator.wikispaces.com/TopCased
- ***EMF***
  - http://es.wikipedia.org/wiki/Framework_de_modelado_Eclipse
  - [*emf1] http://www.vogella.com/articles/EclipseEMF/article.html