FACULTAD DE INFORMÁTICA | UNIVERSIDAD NACIONAL DE LA PLATA

Claudia Pons
Roxana Giandini
Gabriela Pérez

DESARROLLO DE SOFTWARE

dirigido por modelos

Conceptos teóricos y su aplicación práctica

Mc Graw Hill Educación

edulp Editorial de la Universidad de La Plata

DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS

Conceptos teóricos y su aplicación práctica

FACULTAD DE INFORMÁTICA / **UNIVERSIDAD NACIONAL DE LA PLATA**

DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS

Conceptos teóricos y su aplicación práctica

CLAUDIA PONS / ROXANA GIANDINI / GABRIELA PÉREZ

Mc Graw Hill Educación

edulp Editorial de la Universidad de La Plata

Pons, Claudia
Desarrollo de software dirigido por modelos : conceptos teóricos y su aplicación práctica / Claudia Pons ; Roxana Giandini ; Gabriela Pérez. - 1a ed. - La Plata : Universidad Nacional de La Plata, 2010.
280 p. ; 22x16 cm.

ISBN 978-950-34-0630-4

1. Informática. 2. Ingeniería de Software. I. Giandini, Roxana II. Pérez, Gabriela III. Título

Fecha de catalogación: 22/02/2010

DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS
Conceptos teóricos y su aplicación práctica

Claudia Pons | Roxana Giandini | Graciela Pérez

Diagramación: Andrea López Osornio

Editorial de la Universidad Nacional de La Plata
Calle 47 Nº 380 - La Plata (1900) - Buenos Aires - Argentina
Tel/Fax: 54-221-4273992
e-mail: editorial_unlp@yahoo.com.ar
www.unlp.edu.ar/editorial

La EDULP integra la Red de Editoriales Universitarias (REUN)

1º edición - 2010
ISBN Nº 978-950-34-0630-4

Impreso en Argentina

ÍNDICE

Índice de figuras

PRÓLOGO

El Desarrollo de Software Dirigido por Modelos (MDD en su acepción en inglés "Model-Driven Development" es una disciplina que está generando muchas expectativas como alternativa sobresaliente a los métodos convencionales de producción de software, más orientado al Espacio de la Solución que al Espacio del Problema. Después de muchos años intentándolo, parece que por fin la comunidad de la Ingeniería del Software acepta que un proceso robusto de producción de software debe estar soportado por Modelos Conceptuales y dirigido por las Transformaciones correspondientes entre Modelos definidas de forma precisa. Una gran cantidad de trabajos teóricos y prácticos acompañan a este movimiento. Existen también herramientas que lo hacen ya realidad a nivel comercial. Pero a menudo se olvida un componente fundamental para que su éxito sea una realidad tangible: la necesidad de disponer de material didáctico ágil, actualizado, preciso y riguroso, que permita entender los fundamentos del MDD, las abstracciones en que se basa, los estándares que lo soportan, los problemas que hay que abordar para ponerlo en práctica exitosamente, y las ventajas concretas derivadas de su adopción.

Ese es justamente el gran valor proporcionado por este libro. Sus autoras proyectan toda su amplia experiencia didáctica e investigadora en ambientes MDD sobre un documento que introduce al lector de una manera clara, sencilla, eficaz y eficiente en el mundo del MDD y de todo lo que representa. Sus contenidos están actualizados, e incluyen todos los temas que son hoy en día esenciales para entender las peculiaridades del Desarrollo Dirigido por Modelos. Modelos, transformaciones y herramientas, sus implicaciones metodológicas en un proceso software MDD, el rol del metamodelado, las herramientas de soporte a la creación de modelos de software y para la definición de transformaciones de modelos, las tecnologías más relevantes existentes en la actualidad, los lenguajes para definir las transformaciones, el testing dirigido por modelos… En definitiva, todos los aspectos ciertamente significativos del MDD y necesarios para entenderlo, están presentes, correcta-

mente desarrollados, y están descritos con rigor y claridad, combinando aspectos teóricos y prácticos de una manera equilibrada, utilizando ejemplos ciertamente clarificadores para que los conceptos lleguen al lector con nitidez. Todo ello hace de este libro una referencia seguro que indispensable en la docencia de las técnicas de Desarrollo de Software Dirigido por Modelos. Su estilo ameno, sistemático y siempre orientado a facilitar la entendibilidad del enfoque MDD permite concluir que se trata de un libro del que se van a poder beneficiar tanto estudiantes de pre y postgrado, como profesionales del software que quieran conocer de forma suficientemente exhaustiva las particularidades del MDD para poder explotarlo en entornos industriales.

Invito por lo tanto al lector a sumergirse con la ayuda de este libro en el apasionante mundo del Desarrollo Dirigido por Modelos, y a disfrutar de una obra que le permitirá conocer los secretos de un enfoque al que está intensamente e ilusionadamente orientada la Ingeniería del Software moderna.

Oscar Pastor, Noviembre 2009.

Director del Centro de Investigación en Métodos de Producción de Software (ProS). Dpto. de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, España.

INTRODUCCIÓN

A lo largo de estos años hemos visto surgir el Desarrollo de Software Dirigido por Modelos (MDD) como una nueva área dentro el campo de la ingeniería de software. MDD plantea una nueva forma de entender el desarrollo y mantenimiento de sistemas de software con el uso de modelos como principales artefactos del proceso de desarrollo. En MDD, los modelos son utilizados para dirigir las tareas de comprensión, diseño, construcción, pruebas, despliegue, operación, administración, mantenimiento y modificación de los sistemas.

En este libro explicamos los fundamentos de MDD, respondiendo a preguntas tales como "¿Qué son los modelos, cómo se construyen y cómo se transforman hasta llegar al código ejecutable?". También nos referimos a los estándares que soportan a MDD y discutimos las ventajas que se introducen en el ciclo de vida del software como consecuencia de adoptarlo.

El libro contiene un ejemplo completo de un desarrollo dirigido por modelos. El desarrollo comienza con la definición de un modelo abstracto expresado en UML y finaliza con el despliegue de un programa ejecutable escrito en Java. La transformación del modelo a código es realizada a través de la aplicación de transformaciones expresadas en un lenguaje estándar. Este ejemplo brinda un panorama completo y comprensible sobre los aspectos técnicos de MDD.

¿A quién está dirigido este libro?

Si bien este libro está dirigido principalmente a estudiantes de carreras de grado y postgrado relacionadas con la ingeniería de sistemas de software, también constituye una lectura recomendable para cualquier persona, con conocimientos básicos sobre sistemas, que esté interesada en incursionar en el desarrollo de software guiado por modelos.

Este libro intenta ser fácil de abordar, ya que presenta los temas de manera auto-contenida, gradual, sistemática y recurriendo a numerosos ejemplos.

Profesionales del software con mayor experiencia también pueden beneficiarse con la lectura de este libro, ya que obtendrán un mejor entendimiento de MDD que los ayudará a juzgar cuándo y cómo las ideas de MDD pueden ser aplicadas en sus proyectos.
Para entender este libro se requiere conocimiento básico sobre el paradigma de Orientación a Objetos y el lenguaje Unified Modeling Language (UML). Para comprender los ejemplos en detalle, contar con conocimiento sobre el lenguaje Object Constraint Language (OCL), Java, el framework Spring, SQL y Java Server Pages (JSP) es beneficioso pero no indispensable.

Otras fuentes de información sobre el tema

Más información sobre MDD puede consultarse en los libros mencionados en la bibliografía y en los siguientes lugares:

Congresos
Anualmente tienen lugar varios congresos y workshops relacionados con MDD, entre los que destacan International Conference on Model Driven Engineering Languages and Systems (MoDELS), European Conference on Model Driven Architecture (EC-MDA) y la International Conference on Model Transformation (ICMT).

OMG
En la página web del OMG (Object Management Group, www.omg.org) se pueden encontrar las especificaciones de estándares para MDA (MOF, XMI, etc.) y la página oficial de MDA <http:// www.omg.org/mda/>.

Eclipse
El proyecto Eclipse Modeling tiene como objetivo promover el desarrollo dirigido por modelos en la comunidad Eclipse. A partir de <http:// www.eclipse.org/modeling/> se puede acceder a todos los subproyectos destinados a tal fin (EMF, GMF, GMT, etc.).

Otros sitios en la web:
- <http://planet-mde.org/> Portal para la comunidad científica y docente de MDD.
- <http://www.model-transformation.org/> Incluye enlaces a congresos y grupos de investigación y una colección muy interesante de trabajos científicos.

- <http://www.dsmforum.org/> Dedicado al desarrollo basado en lenguajes específicos del dominio. Incluye información sobre ejemplos de aplicación en la industria, herramientas y eventos.

¿Cómo debe leerse este libro?

Si bien es recomendable leer el libro en forma completa y secuencial, podemos sugerir dos caminos abreviados de lectura apropiados para cada perfil de lector:

- Un camino para los gerentes de proyectos: los gerentes que no deseen adentrarse en los detalles de la tecnología que sustenta a MDD deben leer desde el capítulo 1 al 2 y luego los capítulos 11 y 12.
- Un camino para los desarrolladores de software: la gente interesada en la aplicación de MDD para desarrollar software debería leer desde el capítulo 1 hasta el capítulo 9.

También puede omitirse la lectura detallada del ejemplo que desarrollamos a partir del capítulo 5. Dicho ejemplo incluye varios niveles de modelos y transformaciones. El lector podría focalizarse sólo en uno de dichos niveles (por ejemplo, el nivel del modelo relacional), sin perjuicio de perder la información que el ejemplo transmite.

Material didáctico e implementación del ejemplo

La implementación completa y ejecutable del ejemplo del Bookstore que desarrollamos a partir del capítulo 5 puede ser descargada desde la siguiente página Web:

http://www.lifia.info.unlp.edu.ar/bookstore

Dicha página también incluye presentaciones y ejercicios cuyo objetivo es servir de material didáctico en cursos sobre MDD.

CAPÍTULO 1

1. Desarrollo de Software Dirigido por Modelos. Conceptos básicos

En este capítulo introduciremos los fundamentos del paradigma de desarrollo de software dirigido por modelos, relataremos sus orígenes y sus principales beneficios.

1.1 La crisis del software

Históricamente, el proceso de desarrollo de software ha resultado caro, riesgoso, incierto y demasiado lento para las condiciones de negocio modernas. Estos inconvenientes dieron origen al concepto de "crisis del software" que prácticamente surgió conjuntamente con la creación del software. La crisis del software es un término informático acuñado en 1968, en la primera conferencia organizada por la OTAN sobre desarrollo de software. La causa de esta crisis reside en la complejidad inherente a la tarea de construir sistemas de software, y también en los cambios constantes a los que tiene que someterse el software para adaptarse a las necesidades cambiantes de los usuarios y a las innovaciones tecnológicas.
Actualmente, la magnitud de este problema continúa creciendo, ya que se incrementan las demandas de funcionalidades más sofisticadas y de software más fiable (como Grady Booch apunta, en cierto modo "software runs the world" [Booch 04a]). Por lo tanto, es fundamental comprender donde se encuentran las fuentes de esta complejidad y lo que podemos hacer con ella. En las siguientes secciones estudiaremos de que forma los métodos de desarrollo de software actuales enfrentan a la crisis del software.

1.2 El desarrollo de software basado en modelos (MBD)

La ingeniería de software establece que el problema de construir software debe ser encarado de la misma forma en que los ingenieros construyen otros sistemas complejos, como puentes, edificios, barcos y aviones. La idea básica consiste en observar el sistema de software a construir como un producto complejo y a su proceso de construcción como un trabajo ingenieril. Es decir, un proceso planificado basado en metodologías formales apoyadas por el uso de herramientas.

Hacia finales de los años 70, Tom Demarco en su libro "Structured Analysis and System Specification" [Demarco 79] introdujo el concepto de desarrollo de software basado en modelos o MBD (por sus siglas en inglés Model Based Development). DeMarco destacó que la construcción de un sistema de software debe ser precedida por la construcción de un modelo, tal como se realiza en otros sistemas ingenieriles (Figura 1-1). El modelo del sistema es una conceptualización del dominio del problema y de su solución. El modelo se focaliza sobre el mundo real: identificando, clasificando y abstrayendo los elementos que constituyen el problema y organizándolos en una estructura formal.

La abstracción es una de las principales técnicas con la que la mente humana se enfrenta a la complejidad. Ocultando lo que es irrelevante, un sistema complejo se puede reducir a algo comprensible y manejable. Cuando se trata de software, es sumamente útil abstraerse de los detalles tecnológicos de implementación y tratar con los conceptos del dominio de la forma más directa posible. De esta forma, el modelo de un sistema provee un medio de comunicación y negociación entre usuarios, analistas y desarrolladores que oculta o minimiza los aspectos relacionados con la tecnología de implementación.

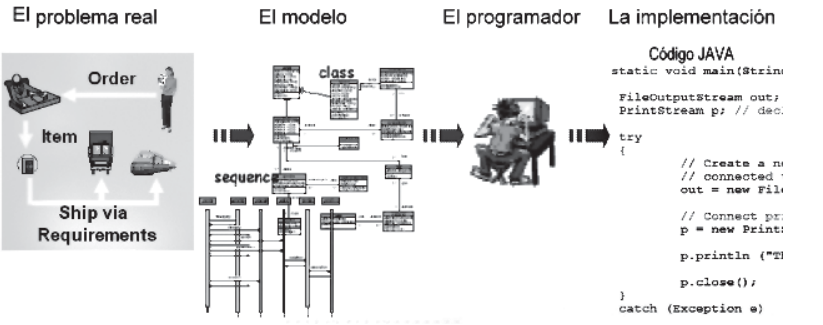

Figura 1-1. Desarrollo de software basado en modelos

Actualmente casi todos los métodos de desarrollo de software utilizan modelos. Lo que varía de un método a otro es la clase de modelos que deben construirse, la forma de representarlos y manipularlos. A grandes rasgos podemos distinguir dos tendencias principales: los métodos matemáticos y los métodos diagramáticos.

Métodos Matemáticos: Existen técnicas y herramientas formales que ayudan a modelar el problema y razonar sobre la solución de una manera precisa y rigurosa. Estos métodos utilizan lenguajes de especificación de naturaleza matemática, tales como: Z [DB 01], VDM [EF 94], B [BM 00] y OCL [OCL], los cuales permiten demostrar si la especificación cumple ciertas propiedades (ej., consistencia), derivar información implícita a partir de la especificación (ej., usando probadores de teoremas), derivar código automáticamente (ej., aplicando cálculos de refinamientos [BW 98]), verificar formalmente que el software satisface la especificación (ej., aplicando la lógica de Hoare [Hoare 69]). Es innegable que el desarrollo de software usando formalismos lleva a generar sistemas robustos y consistentes, en los cuales es posible tanto la validación de la especificación por parte del usuario (detección y corrección temprana de defectos), como la verificación del software. Sin embargo éstos no han sido adoptados masivamente en la industria debido a la complejidad de sus formalismos matemáticos que resultan difíciles de entender y comunicar.

Métodos Diagramáticos: Por otra parte, los procesos basados en modelos gráficos –como el UP [JBR 99] con su especialización RUP (Rational Unified process) [Krutchten 00]– constituyen una propuesta más amigable, fácil de utilizar y comprender que los métodos formales. El éxito de estos procesos se basa principalmente en el uso de lenguajes diagramáticos, tales como UML [UML 03] que transmiten un significado intuitivo. A diferencia de las notaciones matemáticas, estos lenguajes diagramáticos son aceptados más fácilmente por los desarrolladores de software. Además, no por ser amigables los modelos dejan de tener una base formal. En efecto la tienen, pero permanece oculta tras la notación gráfica.

1.2.1 El ciclo de vida basado en modelos

Si bien el desarrollo de software basado en modelos ha representado un paso importante en la ingeniería de software, la crisis sigue existiendo. El proceso sigue careciendo de características que nos permitan ase-

gurar la calidad y corrección del producto final. El proceso de desarrollo de software, como se utiliza hoy en día, es conducido por el diseño de bajo nivel y por el código. Un proceso típico es el que se muestra en la figura 1-2, el cual incluye seis fases:

- La conceptualización y la determinación de los requisitos del usuario
- Análisis y descripción funcional
- Diseño
- Codificación
- Testeo
- Emplazamiento

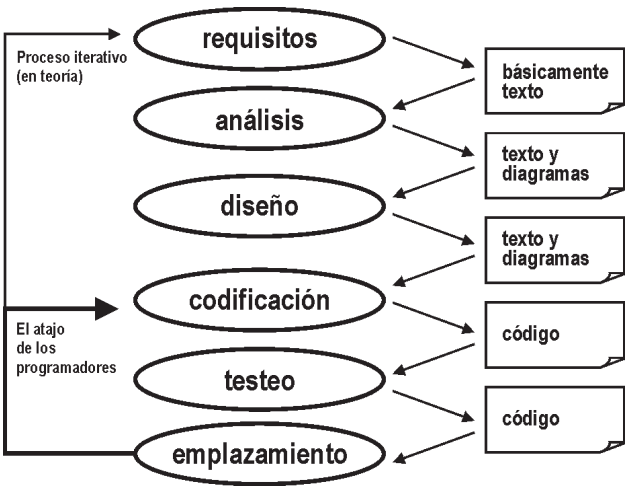

Figura 1-2. Ciclo de vida del desarrollo de software basado en modelos

En las primeras fases se construyen distintos modelos tales como los modelos de los requisitos (generalmente escritos en lenguaje natural), los modelos de análisis y los modelos de diseño (frecuentemente expresados mediante diagramas). Estas fases pueden realizarse de una manera iterativa-incremental o en forma de cascada. En la práctica cotidiana, los modelos quedan rápidamente desactualizados debido al atajo que suelen tomar los desarrolladores en el ciclo de vida de este proceso.

1.2.2 Los problemas del desarrollo basado en modelos

A continuación, analizamos algunos de los problemas más importantes encontrados durante el desarrollo de software basado en modelos y explicamos sus causas.

El problema de la productividad, el mantenimiento y la documentación:

En un proceso de desarrollo basado en modelos, la conexión entre los diagramas y el código se va perdiendo gradualmente mientras se progresa en la fase de codificación. Como puede verse en la figura 1-2, los programadores suelen hacer los cambios sólo en el código, porque no hay tiempo disponible para actualizar los diagramas y otros documentos de alto nivel. Además, el valor agregado de diagramas actualizados y los documentos es cuestionable, porque cualquier nuevo cambio se encuentra en el código de todos modos. Muchas veces se considera la tarea de la documentación como una sobrecarga adicional. Se cree que escribir código es productivo, pero hacer modelos y documentación no lo es. Entonces ¿por qué utilizar tiempo valioso en construir especificaciones de alto nivel? Sin embargo desechar los modelos y considerar únicamente al código puede complicar extremadamente la tarea de mantenimiento del sistema, especialmente luego de transcurrido un tiempo considerable desde la construcción del mismo. Dadas quinientas mil líneas de código (o aún mucho más), ¿por dónde se comienza a intentar entender cómo trabaja el sistema, especialmente cuando la gente que lo escribió ya no está disponible para transmitir su conocimiento?
Otra solución para este problema, en el nivel de código, es la posibilidad de generar documentación directamente desde el código fuente. Esta solución, sin embargo, resuelve únicamente el problema de la documentación en niveles inferiores. En cambio, la de alto nivel (tanto textos como diagramas) aún necesita ser mantenida manualmente. Dada la complejidad de los sistemas de software actuales, resulta imprescindible contar con documentación en los distintos niveles de abstracción. Planteado así, quedan dos alternativas: o utilizamos nuestro tiempo en las primeras fases del desarrollo del software construyendo la documentación y diagramas de alto nivel y nos preocupamos por mantenerlos actualizados, o utilizamos nuestro tiempo en la fase de mantenimiento descubriendo lo que hace realmente el software.

El problema de la flexibilidad a los cambios tecnológicos:

La industria del software tiene una característica especial que la diferencia de las otras industrias. Cada año (o en menos tiempo), aparecen nuevas tecnologías que rápidamente llegan a ser populares (a modo de ejemplo, se pueden listar *Java*, *Linux*, *XML, HTML, UML, .NET, JSP, ASP, PHP, flash*, servicios de *Web*, etc.). Y muchas compañías necesitan aplicar estas nuevas tecnologías por buenas razones:

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

Las nuevas tecnologías ofrecen beneficios concretos para las compañías y muchas de ellas no pueden quedarse atrás. Por lo tanto, tienen que pasar a utilizarlas cuanto antes. Como consecuencia del cambio, las inversiones en tecnologías anteriores pierden valor. Por consiguiente, el software existente debe migrar a una nueva tecnología, o a una versión nueva y diferente de la usada en su construcción. También puede darse otro caso en donde el software permanece utilizando la vieja tecnología, pero ahora necesita comunicarse con sistemas nuevos, los cuales sí han adoptado las nuevas tecnologías. Todas estas cuestiones nos plantean la existencia del problema de la flexibilidad tecnológica, al cual el desarrollo basado en modelos no aporta una solución integral.
En las siguientes secciones presentaremos un nuevo paradigma de desarrollo de software que enfrenta exitosamente a los problemas expuestos anteriormente.

1.3 El desarrollo de software dirigido por modelos (MDD)

El Desarrollo de Software Dirigido por Modelos MDD (por sus siglas en inglés: Model Driven software Development) se ha convertido en un nuevo paradigma de desarrollo software. MDD promete mejorar el proceso de construcción de software basándose en un proceso guiado por modelos y soportado por potentes herramientas. El adjetivo “dirigido” (*driven)* en MDD, a diferencia de “basado” (*based)*, enfatiza que este paradigma asigna a los modelos un rol central y activo: son al menos tan importan-

tes como el código fuente. Los modelos se van generando desde los más abstractos a los más concretos a través de pasos de transformación y/o refinamientos, hasta llegar al código aplicando una última transformación. La transformación entre modelos constituye el motor de MDD. Los puntos claves de la iniciativa MDD fueron identificados en [Booch 04b] de la siguiente forma:

1- El uso de un mayor nivel de **abstracción** en la especificación tanto del problema a resolver como de la solución correspondiente, en relación con los métodos tradicionales de desarrollo de software.
2- El aumento de confianza en la **automatización** asistida por computadora para soportar el análisis, el diseño y la ejecución.
3- El uso de **estándares** industriales como medio para facilitar las comunicaciones, la interacción entre diferentes aplicaciones y productos, y la especialización tecnológica.

La figura 1-3 muestra la parte del proceso de desarrollo de software en donde la intervención humana es reemplazada por herramientas automáticas. Los modelos pasan de ser entidades contemplativas (es decir, artefactos que son interpretadas por los diseñadores y programadores) para convertirse en entidades productivas a partir de las cuales se deriva la implementación en forma automática.

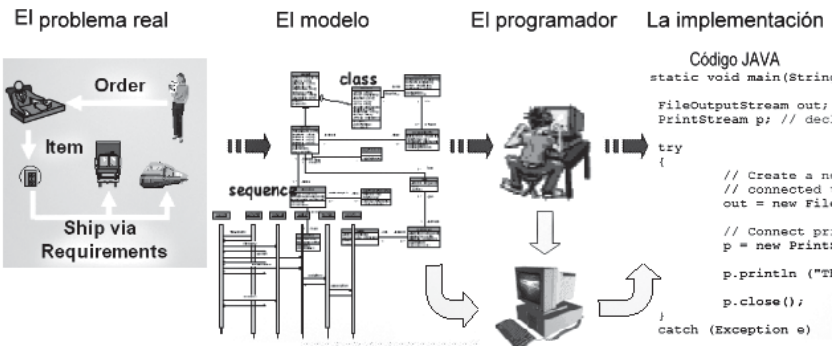

Figura 1-3. Desarrollo de software dirigido por modelos

Lo que sigue es un breve resumen de la naturaleza y la razón de cada uno de estos elementos clave.

Abstracción: El enfoque de MDD para incrementar los niveles de abstracción es definir lenguajes de modelado específicos de dominio cuyos conceptos reflejen estrechamente los conceptos del dominio del problema, mientras se ocultan o minimizan los aspectos relacionados con las tecnologías de implementación. Estos lenguajes utilizan formas sintácticas que resultan amigables y que transmiten fácilmente la esencia de los conceptos del dominio. Por otra parte, el modelo permite reducir el impacto que la evolución tecnológica impone sobre el desarrollo de aplicaciones, permitiendo que el mismo modelo abstracto se materialice en múltiples plataformas de software. Además, la propiedad intelectual usualmente asociada a una aplicación, deja de pertenecer al reino del código fuente y pasa a ser parte del modelo.
Automatización: La automatización es el método más eficaz para aumentar la productividad y la calidad. En MDD la idea es utilizar a las computadoras para automatizar tareas repetitivas que se puedan mecanizar, tareas que los seres humanos no realizan con particular eficacia. Esto incluye, entre otras, la capacidad de transformar modelos expresados mediante conceptos de alto nivel, específicos del dominio, en sus equivalentes programas informáticos ejecutables sobre una plataforma tecnológica específica. Además las herramientas de transformación pueden aplicar reiteradas veces patrones y técnicas con éxito ya comprobado, favoreciendo la confiabilidad del producto.
Estándares: MDD debe ser implementado mediante una serie de estándares industriales abiertos. Estas normas proporcionan numerosos beneficios, como por ejemplo la capacidad para intercambiar especificaciones entre herramientas complementarias, o entre herramientas equivalentes de diferentes proveedores. Los estándares permiten a los fabricantes de herramientas centrar su atención en su principal área de experticia, sin tener que recrear y competir con funcionalidades implementadas por otros proveedores. Por ejemplo, una herramienta que transforma modelos no necesita incluir una funcionalidad de edición de modelos. En lugar de ello, puede usar otra herramienta de edición de modelos de otro fabricante que se ajuste a un estándar común.

1.3.1 El ciclo de vida dirigido por modelos

El ciclo de vida de desarrollo de software usando MDD se muestra en la figura 1-4. Este ciclo de vida no luce muy distinto del ciclo de vida tradicional. Se identifican las mismas fases. Una de las mayores diferencias está en el tipo de los artefactos que se crean durante el proceso de desarrollo. Los artefactos son modelos formales, es decir, modelos que

pueden ser comprendidos por una computadora. En la figura 1-4, las líneas punteadas señalan las actividades automatizadas en este proceso.

MDD identifica distintos tipos de modelos:

- modelos con alto nivel de abstracción independientes de cualquier metodología computacional, llamados CIMs (Computational Independent Model),
- modelos independientes de cualquier tecnología de implementación llamados PIMs (Platform Independent Model),
- modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica, conocidos como PSMs (Platform Specific Model),
- y finalmente modelos que representan el código fuente en sí mismo, identificados como IMs (Implementation Model).

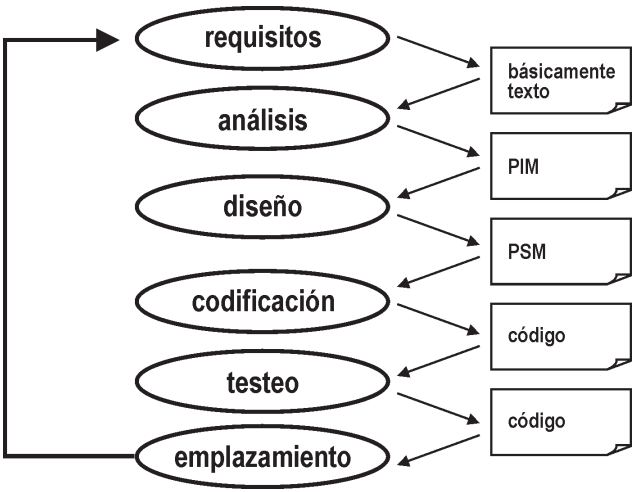

Figura 1-4. Ciclo de vida del desarrollo de software dirigido por modelos

En el proceso de desarrollo de software tradicional, las transformaciones de modelo a modelo, o de modelo a código son hechas mayormente con intervención humana. Muchas herramientas pueden generar código a partir de modelos, pero generalmente no van más allá de la generación de algún esqueleto de código que luego se debe completar manualmente.

En contraste, las transformaciones MDD son siempre ejecutadas por herramientas, como se muestra en la figura 1-5. Muchas herramientas pueden transformar un PSM a código; no hay nada nuevo en eso. Dado que un PSM es un modelo muy cercano al código, esta transformación no es demasiado compleja. Lo novedoso que propone MDD es que las transformaciones entre modelos (por ejemplo de un PIM a PSMs) sean automatizadas.

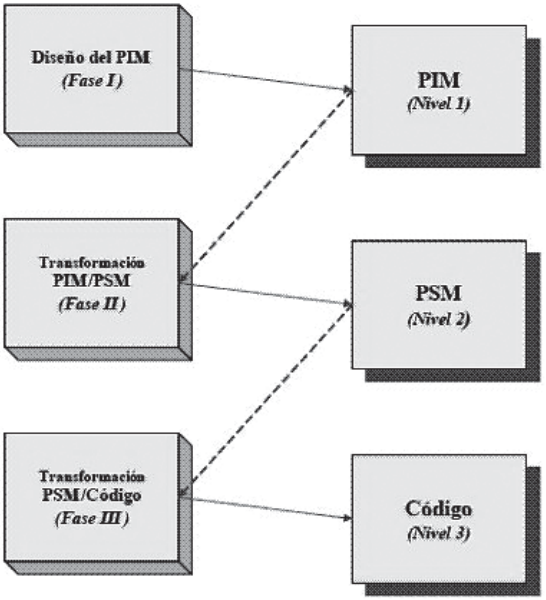

Figura 1-5. Los tres pasos principales en el proceso de desarrollo MDD.

1.3.2 Orígenes de MDD

Si bien MDD define un nuevo paradigma para el desarrollo de software, sus principios fundamentales no constituyen realmente nuevas ideas sino que son reformulaciones y asociaciones de ideas anteriores. MDD es la evolución natural de la ingeniería de software basada en modelos enriquecida mediante el agregado de transformaciones automáticas entre modelos. Por su parte, la técnica de transformaciones sucesivas tampoco es algo nove-

doso. Podemos remitirnos al proceso de abstracción y refinamiento presentado por Edsger W. Dijkstra en su libro "A Discipline of Programming" [Dijkstra 76] donde se define que refinamiento es el proceso de desarrollar un diseño o implementación más detallado a partir de una especificación abstracta a través de una secuencia de pasos matemáticamente justificados que mantienen la corrección con respecto a la especificación original. Es decir, de acuerdo con esta definición, un refinamiento es una transformación semánticamente correcta que captura la relación esencial entre la especificación (es decir, el modelo abstracto) y la implementación (es decir, el código). Consecuentemente podemos señalar que el proceso MDD mantiene una fuerte coincidencia, en sus orígenes e ideas centrales, con el seminal concepto de abstracción y refinamientos sucesivos, el cual ha sido estudiado extensamente en varias notaciones formales tales como Z [DB 01] y B [Lano 96] y diversos cálculos de refinamientos [BW 98]. En general estas técnicas de refinamiento se limitan a transformar un modelo formal en otro modelo formal escrito en el mismo lenguaje (es decir, se modifica el nivel de abstracción del modelo, pero no su lenguaje), mientras que MDD es más amplio pues ofrece la posibilidad de transformar modelos escritos en distintos lenguajes (por ejemplo, podemos transformar un modelo escrito en UML en otro modelo escrito en notación Entidad-Relación).

1.3.3 Beneficios de MDD

El desarrollo de software dirigido por modelos permite mejorar las prácticas corrientes de desarrollo de software. Las ventajas de MDD son las siguientes:

Incremento en la productividad: MDD reduce los costos de desarrollo de software mediante la generación automática del código y otros artefactos a partir de los modelos, lo cual incrementa la productividad de los desarrolladores. Notemos que deberíamos sumar el costo de desarrollar (o comprar) transformaciones, pero es esperable que este costo se amortice mediante el re-uso de dichas transformaciones.

Adaptación a los cambios tecnológicos: el progreso de la tecnología hace que los componentes de software se vuelvan obsoletos rápidamente. MDD ayuda a solucionar este problema a través de una arquitectura fácil de mantener donde los cambios se implementan rápida y consistentemente, habilitando una migración eficiente de los componentes hacia las nuevas tecnologías. Los modelos de alto nivel están libres de detalles de la implementación, lo cual facilita la adaptación a los cambios que pueda sufrir

la plataforma tecnológica subyacente o la arquitectura de implementación. Dichos cambios se realizan modificando la transformación del PIM al PSM. La nueva transformación es reaplicada sobre los modelos originales para producir artefactos de implementación actualizados. Esta flexibilidad permite probar diferentes ideas antes de tomar una decisión final. Y además permite que una mala decisión pueda fácilmente ser enmendada.

Adaptación a los cambios en los requisitos: poder adaptarse a los cambios es un requerimiento clave para los negocios, y los sistemas informáticos deben ser capaces de soportarlos. Cuando usamos un proceso MDD, agregar o modificar una funcionalidad de negocios es una tarea bastante sencilla, ya que el trabajo de automatización ya está hecho. Cuando agregamos una nueva función, sólo necesitamos desarrollar el modelo específico para esa nueva función. El resto de la información necesaria para generar los artefactos de implementación ya ha sido capturada en las transformaciones y puede ser re-usada.

Consistencia: la aplicación manual de las prácticas de codificación y diseño es una tarea propensa a errores. A través de la automatización MDD favorece la generación consistente de los artefactos.

Re-uso: en MDD se invierte en el desarrollo de modelos y transformaciones. Esta inversión se va amortizando a medida que los modelos y las transformaciones son re-usados. Por otra parte el re-uso de artefactos ya probados incrementa la confianza en el desarrollo de nuevas funcionalidades y reduce los riesgos ya que los temas técnicos han sido previamente resueltos.

Mejoras en la comunicación con los usuarios: los modelos omiten detalles de implementación que no son relevantes para entender el comportamiento lógico del sistema. Por ello, los modelos están más cerca del dominio del problema, reduciendo la brecha semántica entre los conceptos que son entendidos por los usuarios y el lenguaje en el cual se expresa la solución. Esta mejora en la comunicación influye favorablemente en la producción de software mejor alineado con los objetivos de sus usuarios.

Mejoras en la comunicación entre los desarrolladores: los modelos facilitan el entendimiento del sistema por parte de los distintos desarrolladores. Esto da origen a discusiones más productivas y permite mejorar los diseños. Además, el hecho de que los modelos son parte del sistema y no sólo documentación, hace que los modelos siempre permanezcan actualizados y confiables.

Captura de la experiencia: las organizaciones y los proyectos frecuentemente dependen de expertos clave quienes toman las decisiones respecto al sistema. Al capturar su experiencia en los modelos y en las transformaciones, otros miembros del equipo pueden aprovecharla sin requerir su presencia. Además este conocimiento se mantiene aún cuando los expertos se alejen de la organización.

Los modelos son productos de larga duración: en MDD los modelos son productos importantes que capturan lo que el sistema informático de la organización hace. Los modelos de alto nivel son resistentes a los cambios a nivel plataforma y sólo sufren cambios cuando lo hacen los requisitos del negocio.

Posibilidad de demorar las decisiones tecnológicas: cuando aplicamos MDD, las primeras etapas del desarrollo se focalizan en las actividades de modelado. Esto significa que es posible demorar la elección de una plataforma tecnológica específica o una versión de producto hasta más adelante cuando se disponga de información que permita realizar una elección más adecuada.

1.4 Propuestas Concretas para MDD

Dos de las propuestas concretas más conocidas y utilizadas en el ámbito de MDD son, por un lado MDA desarrollada por el OMG y por otro lado el modelado específico del domino (DSM acrónimo inglés de Domain Specific Modeling) acompañado por los lenguajes específicos del dominio (DSLs acrónimo inglés de Domain Specific Language). Ambas iniciativas guardan naturalmente una fuerte conexión con los conceptos básicos de MDD. Específicamente, MDA tiende a enfocarse en lenguajes de modelado basados en estándares del OMG, mientras que DSM utiliza otras notaciones no estandarizadas para definir sus modelos.

1.4.1 Arquitectura Dirigida por Modelos (MDA)

MDA [MDAG], acrónimo inglés de *Model Driven Architecture* (Arquitectura Dirigida por Modelos), es una de las iniciativas más conocida y extendida dentro del ámbito de MDD. MDA es un concepto promovido por el OMG *(*acrónimo inglés de *Object Management Group)* a partir de 2000, con el objetivo de afrontar los desafíos de integración de las aplicaciones y los continuos cambios tecnológicos. MDA es una arquitectura que

proporciona un conjunto de guías para estructurar especificaciones expresadas como modelos, siguiendo el proceso MDD.
En MDA, la funcionalidad del sistema es definida en primer lugar como un modelo independiente de la plataforma (Platform-Independent Model o PIM) a través de un lenguaje específico para el dominio del que se trate. En este punto aparece además un tipo de modelo existente en MDA, no mencionado por MDD: el modelo de definición de la plataforma (Platform Definition Model o PDM). Entonces, dado un PDM correspondiente a CORBA, .NET, Web, etc., el modelo PIM puede traducirse a uno o más modelos específicos de la plataforma (Platform-Specific Models o PSMs) para la implementación correspondiente, usando diferentes lenguajes específicos del dominio, o lenguajes de propósito general como Java, C#, Python, etc. El proceso MDA completo se encuentra detallado en un documento que actualiza y mantiene el OMG denominado la Guía MDA [MDAG].
MDA no se limita al desarrollo de sistemas de software, sino que también se adapta para el desarrollo de otros tipos de sistemas. Por ejemplo, MDA puede ser aplicado en el desarrollo de sistemas que incluyen a personas como participantes junto con el software y el soporte físico. Asimismo, MDA está bien adaptado para el modelado del negocio y empresas. Es claro que el alcance del uso de los conceptos básicos de MDA es más amplio que el expuesto en esta sección.
MDA está relacionada con múltiples estándares, tales como el Unified Modeling Language (UML), el Meta-Object Facility (MOF), XML Metadata Interchange (XMI), Enterprise Distributed Object Computing (EDOC), el Software Process Engineering Metamodel (SPEM) y el Common Warehouse Metamodel (CWM).
El OMG mantiene la marca registrada sobre MDA, así como sobre varios términos similares incluyendo Model Driven Development (MDD), Model Driven Application Development, Model Based Application Developtment, Model Based Programming y otros similares. El acrónimo principal que aún no ha sido registrado por el OMG hasta el presente es MDE, que significa Model Driven software Engineering. A consecuencia de esto, el acrónimo MDE es usado actualmente por la comunidad investigadora internacional cuando se refieren a ideas relacionadas con la ingeniería de modelos sin centrarse exclusivamente en los estándares del OMG.
Cumpliendo con las directivas del OMG, las dos principales motivaciones de MDA son la interoperabilidad (independencia de los fabricantes a través de estandarizaciones) y la portabilidad (independencia de la plataforma) de los sistemas de software; las mismas motivaciones que llevaron al desarrollo de CORBA. Además, el OMG postula como objetivo de MDA separar el diseño del sistema tanto de la arquitectura como de

las tecnologías de construcción, facilitando así que el diseño y la arquitectura puedan ser alterados independientemente. El diseño alberga los requisitos funcionales (casos de uso, por ejemplo) mientras que la arquitectura proporciona la infraestructura a través de la cual se hacen efectivos los requisitos no funcionales como la escalabilidad, fiabilidad o rendimiento. MDA asegura que el modelo independiente de la plataforma (PIM), el cual representa un diseño conceptual que plasma los requisitos funcionales, sobreviva a los cambios que se produzcan en las tecnologías de fabricación y en las arquitecturas de software. Por supuesto, la noción de transformación de modelos en MDA es central. La traducción entre modelos se realiza normalmente utilizando herramientas automatizadas, es decir herramientas de transformación de modelos que soportan MDA. Algunas de ellas permiten al usuario definir sus propias transformaciones. Una iniciativa del OMG es la definición de un lenguaje de transformaciones estándar denominado QVT [QVT] que aún no ha sido masivamente adoptado por las herramientas, por lo que la mayoría de ellas aún definen su propio lenguaje de transformación, y sólo algunos de éstos se basan en QVT. Actualmente existe un amplio conjunto de herramientas que brindan soporte para MDA. Sobre estos temas nos extenderemos en los siguientes capítulos.

1.4.2 Modelado Específico del Dominio (DSM y DSLs)

Por su parte, la iniciativa DSM (Domain-Specific Modeling) es principalmente conocida como la idea de crear modelos para un dominio, utilizando un lenguaje focalizado y especializado para dicho dominio. Estos lenguajes se denominan DSLs (por su nombre en inglés: Domain-Specific Language) y permiten especificar la solución usando directamente conceptos del dominio del problema. Los productos finales son luego generados desde estas especificaciones de alto nivel. Esta automatización es posible si ambos, el lenguaje y los generadores, se ajustan a los requisitos de un único dominio. Definimos como dominio a un área de interés para un esfuerzo de desarrollo en particular. En la práctica, cada solución DSM se enfoca en dominios pequeños porque el foco reductor habilita mejores posibilidades para su automatización y estos dominios pequeños son más fáciles de definir. Usualmente, las soluciones en DSM son usadas en relación a un producto particular, una línea de producción, un ambiente específico, o una plataforma.
El desafío de los desarrolladores y empresas se centra en la definición de los lenguajes de modelado, la creación de los generadores de código y la implementación de *frameworks* específicos del dominio, elementos

claves de una solución DSM. Estos elementos no se encuentran demasiado distantes de los elementos de modelado de MDD. Actualmente puede observarse que las discrepancias entre DSM y MDD han comenzado a disminuir. Podemos comparar el uso de modelos así como la construcción de la infraestructura respectiva en DSM y en MDD. En general, DSM usa los conceptos dominio, modelo, metamodelo, metametamodelo como MDD, sin mayores cambios y propone la automatización en el ciclo de vida del software. Los DSLs son usados para construir modelos. Estos lenguajes son frecuentemente -pero no necesariamente- gráficos. Los DSLs no utilizan ningún estándar del OMG para su infraestructura, es decir no están basados en UML, los metamodelos no son instancias de MOF, a diferencia usan por ejemplo MDF, el framework de Metadatos para este propósito.
Finalmente, existe una familia importante de herramientas para crear soluciones en DSM que ayudan en la labor de automatización. En tal sentido, surge el término "*Software Factories*", que ha sido acuñado por Microsoft. Su descripción en forma detallada puede encontrarse en el libro de Greenfields [GS04] del mismo nombre. Una *Software Factory* es una línea de producción de software que configura herramientas de desarrollo extensibles tales como Visual Studio Team System con contenido empaquetado como DSLs, patrones y *frameworks*, basados en recetas para construir tipos específicos de aplicaciones. Visual Studio provee herramientas para definir los metamodelos así como su sintaxis concreta y editores. En el capítulo 4 se verán con más detalle ésta y otras propuestas de herramientas que asisten en la tarea de construcción de modelos y lenguajes de modelado.

1.5 Resumen

En este capítulo hemos reflexionado acerca de la magnitud del problema de construir software de calidad que pueda ser capaz de sobrevivir a la evolución de sus requisitos funcionales y que sea flexible a los cambios en la tecnología que lo sustenta. Hemos enunciado los principios del paradigma de desarrollo de software conocido como MDD que ofrece principios tendientes a mejorar los procesos de construcción de software. Sus postulados básicos son los siguientes:

- Los modelos asumen un rol protagónico en el proceso de desarrollo del software;
- Los modelos pasan de ser entidades contemplativas para convertirse en entidades productivas a partir de las cuales se deriva la implementación en forma automática.

CAPÍTULO 2

2. Los pilares de MDD: modelos, transformaciones y herramientas

En este capítulo analizaremos cuales son los elementos necesarios para poner en práctica el proceso MDD. A grandes rasgos, podemos decir que el proceso MDD se apoya sobre los siguientes pilares:

- Modelos con diferentes niveles de abstracción, escritos en lenguajes bien definidos.
- Definiciones de cómo un modelo se transforma en otro modelo más específico.
- Herramientas de software que den soporte a la creación de modelos y su posterior transformación.

2.1 ¿Qué es un modelo?

El acrónimo MDD enfatiza el hecho de que los modelos son el foco central de MDD. Los modelos que nos interesan son aquellos relevantes para el desarrollo de software, sin embargo estos modelos no sólo representan al software; cuando un sistema de software soporta un determinado negocio, el modelo de dicho negocio es también relevante. Pero, ¿qué significa exactamente la palabra “modelo”?

Necesitamos una definición que sea lo suficientemente general para abarcar varios tipos diferentes de modelos, pero que al mismo tiempo sea lo suficientemente específica para permitirnos definir transformaciones automáticas de un modelo a otro modelo.

En el ámbito científico un modelo puede ser un objeto matemático (ej., un sistema de ecuaciones), un gráfico (ej., un plano) o un objeto físico (ej., una maqueta). El modelo es una representación conceptual o física

a escala de un proceso o sistema, con el fin de analizar su naturaleza, desarrollar o comprobar hipótesis o supuestos y permitir una mejor comprensión del fenómeno real al cual el modelo representa y permitir así perfeccionar los diseños, antes de iniciar la construcción de las obras u objetos reales. Se utilizan con frecuencia para el estudio de represas, puentes, puertos, aeronaves, etc. Muchas veces, para obras complejas como por ejemplo una vivienda, se requiere la construcción de más de un modelo (figura 2-1). Por ejemplo, un modelo general de la disposición de los ambientes de la vivienda con sus puertas y ventanas, un modelo específico a una escala mayor para la instalación eléctrica y sanitaria, uno diferente para mostrar la estética de la fachada y el entorno, etc.

Figura 2-1. Modelos de una vivienda.

2.1.1 Tipos de modelos

La definición de modelo dada en la sección anterior es muy amplia pues abarca muchos tipos diferentes de modelos. Los sistemas de software involucran varios modelos interdependientes a diferentes niveles de abstracción (análisis, diseño, implementación), representando diferentes partes del sistema (interfaz con el usuario, base de datos, lógica del negocio, administración del sistema), diferentes requisitos (seguridad, desempeño, flexibilidad), o diferentes tareas (modelos de pruebas, modelos de emplazamiento). En muchos casos, es posible generar un modelo a partir de otro, por ejemplo pasando del modelo de análisis al modelo de diseño, o del modelo de la aplicación al modelo de pruebas. Existen varias formas de distinguir entre tipos de modelos, cada una de ellas se basa en la respuesta a alguna pregunta acerca del modelo, por ejemplo:

- ¿En qué parte del proceso de desarrollo de software es usado el modelo?
- ¿El modelo es abstracto o es detallado?
- ¿Qué sistema es descrito por el modelo? ¿Es un modelo del negocio o es un modelo de software?
- ¿Qué aspectos del sistema son descritos por el modelo? ¿Es un modelo de la estructura o es un modelo del comportamiento?
- ¿Es un modelo orientado a una tecnología específica o es un modelo independiente de la tecnología?

Las respuestas a estas preguntas nos permiten clasificar los diferentes tipos de modelos en categorías no disjuntas. En las siguientes secciones analizaremos en detalle esta clasificación.

Modelos a lo largo del proceso de desarrollo:

Durante el proceso de desarrollo de software se crean diferentes modelos (figura 2-2). Los modelos de análisis capturan sólo los requisitos esenciales del sistema de software, describiendo lo que el sistema hará independientemente de cómo se implemente. Por otro lado, los modelos de diseño reflejan decisiones sobre el paradigma de desarrollo (orientado a objetos, basado en componentes, orientado a aspectos, etc.), la arquitectura del sistema (distintos estilos arquitecturales), la distribución de responsabilidades (GRASP patterns [Larman 04], GoF patterns [GHJV94]), etc. Finalmente, los modelos de implementación describen cómo el sistema será construido en el contexto de un ambiente de

implementación determinado (plataforma, sistema operativo, bases de datos, lenguajes de programación, etc.). Si bien algunos modelos pueden clasificarse claramente como un modelo de análisis, o de diseño o de implementación, por ejemplo, un diagrama de casos de uso es un modelo de análisis, mientras que un diagrama de interacción entre objetos es un modelo de diseño y un diagrama de deployment es un modelo de implementación. En general, esta clasificación no depende del modelo en sí mismo sino de la interpretación que se de en un cierto proyecto a las etapas de análisis, diseño e implementación.

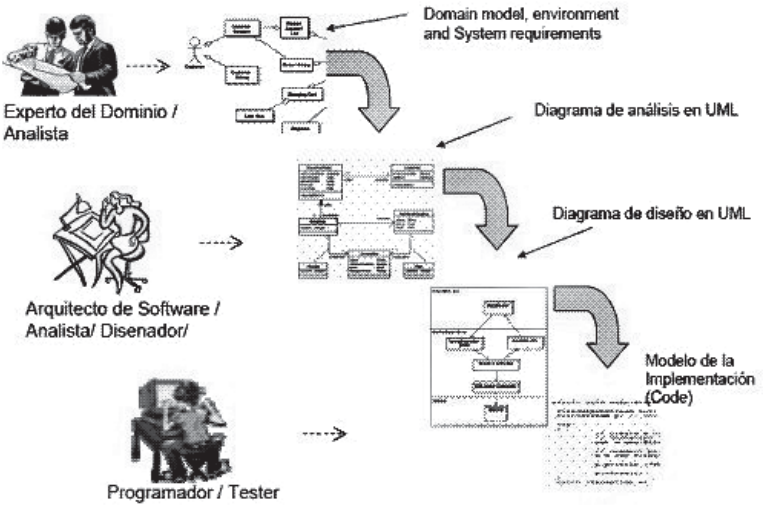

Figura 2-2. Diferentes modelos a lo largo del proceso de desarrollo de software

Modelos abstractos y modelos detallados:

La clasificación entre modelo abstracto y modelo detallado también posee cierto grado de subjetividad. En general las herramientas de modelado nos permiten visualizar a un mismo modelo en distintos niveles de detalle [Egyed 02]. Por ejemplo podemos visualizar un diagrama de clases en forma abstracta limitando la información que se despliega sobre cada

clase sólo a su nombre, y limitando la profundidad de las jerarquías de herencia y composición a un máximo de k niveles. Luego podemos visualizar el diagrama en más detalle, incluyendo los nombres de los métodos públicos de cada clase y/o expandiendo las jerarquías de clases.

Modelos de negocio y modelos de software:

Un modelo del negocio describe a un negocio o empresa (o parte de ellos). El lenguaje que se utiliza para construir el modelo del negocio contiene un vocabulario que permite al modelador especificar los procesos del negocio, los clientes, los departamentos, las dependencias entre procesos, etc. Un modelo del negocio no describe necesariamente detalles acerca del sistema de software que se usa en la empresa; por lo tanto a este modelo suele llamárselo "modelo independiente de la computación" (CIM). Cuando una parte del negocio es soportada por un sistema de software, se construye un modelo diferente para tal sistema. Este modelo de software es una descripción del sistema de software. El sistema de software y el sistema del negocio son conceptos diferentes en el mundo real, sin embargo los requisitos del sistema de software usualmente se derivan del modelo del negocio al cual el software brinda soporte. Para la mayoría de los negocios existen varios sistemas de software dándole soporte. Cada sistema de software se usa para asistir una parte del negocio. Por lo tanto, existe una relación entre un modelo del negocio y los diferentes modelos de software que describen al software que brinda soporte al negocio, como mostramos en la figura 2-3.

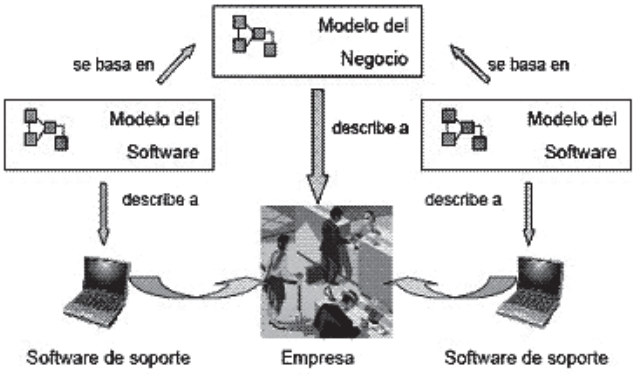

Figura 2-3. Modelos del negocio y modelos del software

Modelos estáticos (o estructurales) y modelos dinámicos (o de comportamiento):

La mayoría de los sistemas poseen una base estructural estática, definida por el conjunto de objetos que constituyen el sistema, con sus propiedades y sus conexiones. Por ejemplo, un banco posee clientes y varias sucursales, en cada sucursal se radican varias cuentas bancarias, cada cuenta bancaria tiene un identificador y un saldo, etc. Por otra parte los sistemas poseen una dinámica definida por el comportamiento que los objetos del sistema despliegan. Por ejemplo, un cliente puede radicar una cuenta en una sucursal del banco y luego puede depositar dinero en su cuenta. Para representar estas dos vistas diferentes pero complementarias del sistema necesitamos modelos estáticos (o estructurales) por un lado y modelos dinámicos (o de comportamiento) por el otro. Ambos tipos de modelos están fuertemente interrelacionados y juntos constituyen el modelo global del sistema. Por lo cual podemos agregar que también nos encontraremos con modelos híbridos conformados por sub-modelos estructurales y sub-modelos dinámicos. La figura 2-4 muestra un modelo compuesto por sub-modelos o vistas, algunas de ellas son estáticas (como el diagrama de clases) y otras dinámicas (como el diagrama de interacción). Todos estos modelos están escritos en el mismo lenguaje: UML. El lenguaje UML [UML 03] es muy expresivo pues está conformado por una familia de lenguajes que nos permite describir tanto los aspectos estáticos como los dinámicos de cada sistema.

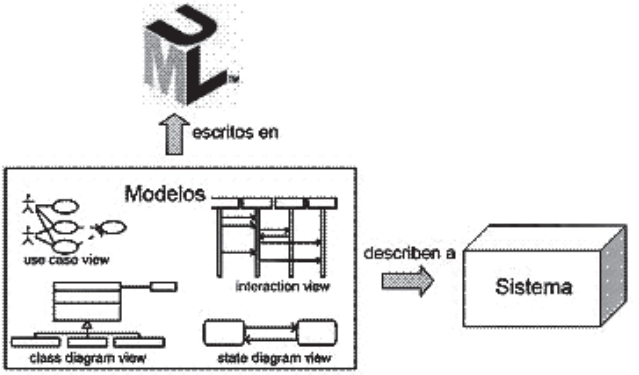

Figura 2-4. Diferentes sub-modelos conformando a un modelo global del sistema.

Por otra parte, existen lenguajes más específicos que sólo permiten modelar una vista del sistema, por ejemplo, los diagramas de entidad-relación se han utilizado extensamente para modelar la estructura de sistemas de software, mientras que el lenguaje de las máquinas de estados y la notación de redes de Petri [Peterson 81], entre otros, resultan adecuadas para modelar el comportamiento. La Figura 2-5 muestra una situación en la cual dos modelos del mismo sistema han sido escritos utilizando diferentes lenguajes.

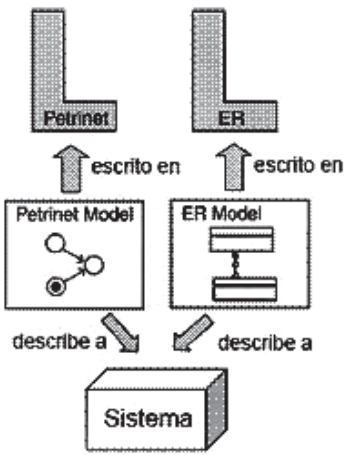

Figura 2-5. Diferentes modelos de un sistema escritos en diferentes lenguajes

Modelos independientes de la plataforma y modelos específicos de la plataforma:

Algunos modelos describen al sistema de manera independiente de los conceptos técnicos que involucra su implementación sobre una plataforma de software, mientras que otros modelos tienen como finalidad primaria describir tales conceptos técnicos. Teniendo en cuenta esta diferencia, los tipos de modelos que identifica MDD son:

- **El modelo independiente de la computación (CIM)** (en inglés Computation Independent Model). Un CIM es una vista del sistema desde un punto de vista independiente de la computación. Un CIM no muestra detalles de la estructura del sistema. Usualmente al CIM se lo llama modelo del dominio y en su construcción se utiliza un vocabulario que resulta familiar para los expertos en el dominio en cuestión. Se asume que los usuarios a quienes está destinado el CIM - los expertos de dominio - no poseen conocimientos técnicos acerca de los artefactos que se usarán para implementar el sistema. El CIM juega un papel muy importante en reducir la brecha entre los expertos en el dominio y sus requisitos por un lado, y los expertos en diseñar y construir artefactos de software por el otro.
- **El modelo independiente de la plataforma (PIM)** (en inglés, Platform Independent Model). Un PIM es un modelo con un alto nivel de abstracción que es independiente de cualquier tecnología o lenguaje de implementación. Dentro del PIM el sistema se modela desde el punto de vista de cómo se soporta mejor al negocio, sin tener en cuenta como va a ser implementado: ignora los sistemas operativos, los lenguajes de programación, el hardware, la topología de red, etc. Por lo tanto un PIM puede luego ser implementado sobre diferentes plataformas especificas.
- **El modelo específico de la plataforma (PSM)** (en inglés, Platform Specific Model). Como siguiente paso, un PIM se transforma en uno o más PSMs (Platform Specific Models). Cada PSM representa la proyección del PIM en una plataforma específica. Un PIM puede generar múltiples PSMs, cada uno para una tecnología en particular. Generalmente, los PSMs deben colaborar entre sí para una solución completa y consistente. Por ejemplo, un PSM para Java contiene términos como clase, interfaz, etc. Un PSM para una base de datos relacional contiene términos como tabla, columna, clave foránea, etc.
- **El modelo de la implementación (Código).** El paso final en el desarrollo es la transformación de cada PSM a código fuente. Ya que el PSM está orientado al dominio tecnológico específico, esta transformación es bastante directa.

El tipo de sistema descrito por un modelo es relevante para las transformaciones de modelos. La derivación completamente automática de PIMs a partir de un CIM no es posible, dado que la decisión acerca de cuáles partes del CIM serán soportadas por un sistema de software debe ser tomada por un humano.

2.1.2 Cualidades de los modelos

El modelo de un problema es esencial para describir y entender el problema, independientemente de cualquier posible sistema informático que se use para su automatización. El modelo constituye la base fundamental de información sobre la que interactúan los expertos en el dominio del problema y los desarrolladores de software. Por lo tanto es de fundamental importancia que exprese la esencia del problema en forma clara y precisa. Por otra parte, la actividad de construcción del modelo es una parte crítica en el proceso de desarrollo. Los modelos son el resultado de una actividad compleja y creativa y por lo tanto son propensos a contener errores, omisiones e inconsistencias. La validación y verificación del modelo es muy importante, ya que la calidad del producto final dependerá fuertemente de la calidad de los modelos que se usaron para su desarrollo. Demos una mirada más detallada a las cualidades que esperamos encontrar en los modelos.

- *Comprensibilidad.* El modelo debe ser expresado en un lenguaje que resulte accesible (es decir entendible y manejable) para todos sus usuarios.
- *Precisión.* El modelo debe ser una fiel representación del objeto o sistema modelado. Para que esto sea posible, el lenguaje de modelado debe poseer una semántica precisa que permita la interpretación unívoca de los modelos. La falta de precisión semántica es un problema que no solamente atañe al lenguaje natural, sino que también abarca a algunos lenguajes gráficos de modelado que se utilizan actualmente.
- *Consistencia.* El modelo no debe contener información contradictoria. Dado que un sistema es representado a través de diferentes sub-modelos relacionados debería ser posible especificar precisamente cuál es la relación existente entre ellos, de manera que sea posible garantizar la consistencia del modelo como un todo.
- *Completitud.* El modelo debe capturar todos los requisitos necesarios. Dado que en general, no es posible lograr un modelo completo desde el inicio del proceso, es importante poder incrementar el modelo. Es decir, comenzar con un modelo incompleto y expandirlo a medida que se obtiene más información acerca del dominio del problema y/o de su solución.
- *Flexibilidad.* Debido a la naturaleza cambiante de los sistemas actuales, es necesario contar con modelos flexibles, es decir que puedan ser fácilmente adaptados para reflejar las modificaciones en el dominio del problema.

- *Re-usabilidad*. El modelo de un sistema, además de describir el problema, también debe proveer las bases para el re-uso de conceptos y construcciones que se presentan en forma recurrente en una amplia gama de problemas.
- *Corrección.* El análisis de la corrección del sistema de software debe realizarse en dos puntos. En primer lugar el modelo en sí debe ser analizado para asegurar que cumple con las expectativas del usuario. Este tipo de análisis generalmente se denomina 'validación del modelo'. Luego, asumiendo que el modelo es correcto, puede usarse como referencia para analizar la corrección de la implementación del sistema Esto se conoce como 'verificación del software'. Ambos tipos de análisis son necesarios para garantizar la corrección de un sistema de software.

2.1.3 ¿Cómo se define un modelo?

El modelo del sistema se construye utilizando un lenguaje de modelado (que puede variar desde lenguaje natural o diagramas hasta fórmulas matemáticas). Los modelos informales son expresados utilizando lenguaje natural, figuras, tablas u otras notaciones. Hablamos de modelos formales cuando la notación empleada es un formalismo, es decir posee una sintaxis y semántica (significado) precisamente definidos. Existen estilos de modelado intermedios llamados semi-formales, ya que en la práctica los ingenieros de software frecuentemente usan una notación cuya sintaxis y semántica están sólo parcialmente formalizadas.
El éxito de los lenguajes gráficos de modelado, tales como el Unified Modeling Language (UML) [UML] se basa principalmente en el uso de construcciones gráficas que transmiten un significado intuitivo; por ejemplo un cuadrado representa un objeto, una línea uniendo dos cuadrados representa una relación entre ambos objetos. Estos lenguajes resultan atractivos para los usuarios ya que aparentemente son fáciles de entender y aplicar. Sin embargo, la falta de precisión en la definición de su semántica puede originar malas interpretaciones de los modelos, inconsistencia entre los diferentes sub-modelos del sistema y discusiones acerca del significado del lenguaje. Además, los lenguajes utilizados en MDD deben estar sustentados por una base formal de tal forma que las herramientas sean capaces de transformar automáticamente los modelos escritos en tales lenguajes.
Por otro lado, los lenguajes formales de modelado, tales como Z [Spivey 92] y VDM [Jones 90], poseen una sintaxis y semántica bien definidas. Sin embargo su uso en la industria es poco frecuente. Esto se debe a la

complejidad de sus formalismos matemáticos que son difíciles de entender y comunicar. En la mayoría de los casos los expertos en el dominio del sistema que deciden utilizar una notación formal, focalizan su esfuerzo sobre el manejo del formalismo en lugar de hacerlo sobre el modelo en sí. Esto conduce a la creación de modelos formales que no reflejan adecuadamente al sistema real.
La necesidad de integrar lenguajes gráficos, cercanos a las necesidades del dominio de aplicación con técnicas formales de análisis y verificación puede satisfacerse combinando ambos tipos de lenguaje. La idea básica para obtener una combinación útil consiste en ocultar los formalismos matemáticos detrás de la notación gráfica. De esta manera el usuario sólo debe interactuar con el lenguaje gráfico, pero puede contar con la base formal provista por el esquema matemático subyacente. Esta propuesta ofrece claras ventajas sobre el uso de un lenguaje informal así como también sobre el uso de un lenguaje formal, ya que permite que los desarrolladores de software puedan crear modelos formales sin necesidad de poseer un conocimiento profundo acerca del formalismo que los sustenta.
En sus orígenes los lenguajes gráficos como el UML carecían de una definición formal. A partir de su aceptación como estándar y de su adopción masiva en la industria, los investigadores se interesaron en construir una base formal para dichos lenguajes. El grupo pUML (siglas en inglés para “the precise UML group”) [pUML] fue creado en 1997 para reunir a investigadores y desarrolladores de todo el mundo con el objetivo de convertir al Unified Modelling Language (UML) en un lenguaje de modelado formal (es decir, bien definido). El grupo ha desarrollado nuevas teorías y técnicas destinadas a:

- clarificar y hacer precisa la sintaxis y la semántica de UML;
- razonar con propiedades de los modelos UML;
- verificar la corrección de diseños UML;
- construir herramientas para soportar la aplicación rigurosa de UML.

A partir de su creación, muchos de sus miembros han estado involucrados en el proceso de estandarización de UML y han organizado numerosas reuniones de trabajo y conferencias relacionadas con el desarrollo de UML como un lenguaje de modelado formal. Actualmente los lenguajes estandarizados por el OMG [OMG] poseen una sintaxis gráfica amigable sustentada por una definición formal. El OMG ha adoptado un lenguaje gráfico especial llamado Meta Object Facility [MOF], y un lenguaje textual basado en lógica de primer orden llamado Object Constraint Language [OCL]. Ambos lenguajes son formales y se utilizan para definir a los demás lenguajes que emergen en MDD. Nos explayaremos

sobre este tema en el Capítulo 3. Esto asegura que las herramientas serán capaces de leer y escribir sin ambigüedades a todos los lenguajes estandarizados por el OMG y además facilita la interoperabilidad entre las distintas herramientas.

2.1.4 El rol de UML en MDD

Generalmente UML es el lenguaje elegido para definir los modelos en MDD. UML es un estándar abierto y es el estándar de facto para el modelado de software. UML es un lenguaje de propósito general que podemos aplicar de varias formas, por ejemplo, las técnicas de modelado apropiadas para diseñar una aplicación de telefonía de tiempo real y las técnicas de modelado para desarrollar una aplicación de comercio electrónico son bastante diferentes, sin embargo podemos usar UML en ambos casos.

Extendiendo a UML mediante estereotipos y perfiles:

Un perfil de UML es un conjunto de extensiones que especializan a UML para su uso en un dominio o contexto particular. Algunos ejemplos de perfiles de UML son los siguientes:

- El perfil para testing:
  http://utp.omg.org/
- El perfil para Servicios de Software:
  http://soa.omg.org/
- El perfil para schedulability, desempeño y tiempo real:
  http://www.omg.org/technology/documents/formal/schedulability.htm

Los perfiles de UML son ortogonales entre sí, por lo tanto varios perfiles pueden aplicarse simultáneamente. Por ejemplo, si estamos modelando aspectos de seguridad dentro de una aplicación que posee una arquitectura orientada a servicios podríamos aplicar conjuntamente le perfil de servicios de software y el perfil de seguridad.
Un perfil de UML define un conjunto de estereotipos que extienden a los conceptos de UML. Por ejemplo el perfil UML para bases de datos introduce un estereotipo llamado <<primaryKey>> que puede ser aplicado a un atributo de una clase UML para indicar que el atributo representa la clave primaria en una tabla de la base de datos. El uso de un estereotipo transmite información semántica adicional que resulta entendible tanto para el modelador humano como para las herramientas automáticas

que procesan a los modelos. Mediante el estereotipo somos capaces de distinguir entre los atributos UML que son claves primarias y los otros atributos que no lo son.

El valor de un lenguaje de modelado común:

El uso de UML como lenguaje en MDD tiene los siguientes beneficios:

- El uso de perfiles de UML para MDD nos permite aprovechar la experiencia que se ha ganado en el desarrollo de este lenguaje. Esto significa que podemos proveer un ambiente de modelado personalizado sin afrontar el costo de diseñarlo e implementarlo desde cero.
- UML es un estándar abierto y el estándar de facto para el modelado de software en la industria.
- UML ha demostrado ser durable, su primera versión apareció en 1995.
- El éxito de UML ha propiciado la disponibilidad de muchos libros y cursos de muy buena calidad.
- La mayoría de los estudiantes de las carreras relacionadas con la ingeniería de software han aprendido algo de UML en la universidad.
- Existen muchas herramientas maduras que soportan UML.
- Muchas organizaciones necesitan desarrollar varios tipos diferentes de software. Si podemos usar una solución de modelado en común, configurada de forma apropiada para describir cada uno de esos dominios de software, entonces será más fácil lidiar con su posterior integración.

Sin embargo, no todas son ventajas al usar UML como lenguaje de modelado. También podemos señalar las siguientes desventajas:

- Algunas de las herramientas de modelado que soportan UML no permiten la definición de perfiles; las herramientas para UML que sí soportan perfiles sólo lo hacen superficialmente, no permitiendo aprovechar todo el potencial de este mecanismo de extensión.
- UML por ser un lenguaje de propósito general es muy amplio. Incluye decenas de elementos de modelado que además pueden combinarse de formas diversas. En cambio, los lenguajes específicos de un dominio suelen ser pequeños (es decir, contienen pocos conceptos). El modelador sólo necesita entender esos pocos conceptos, en cambio si usamos un lenguaje basado en UML necesitaremos como pre-requisito un conocimiento bastante completo sobre UML.

2.2 ¿Qué es una transformación?

El proceso MDD, descrito anteriormente, muestra el rol de varios modelos, PIM, PSM y código dentro del framework MDD. Una herramienta que soporte MDD, toma un PIM como entrada y lo transforma en un PSM. La misma herramienta u otra, tomará el PSM y lo transformará a código. Estas transformaciones son esenciales en el proceso de desarrollo de MDD. En la figura 2-6 se muestra la herramienta de transformación como una caja negra, que toma un modelo de entrada y produce otro modelo como salida.

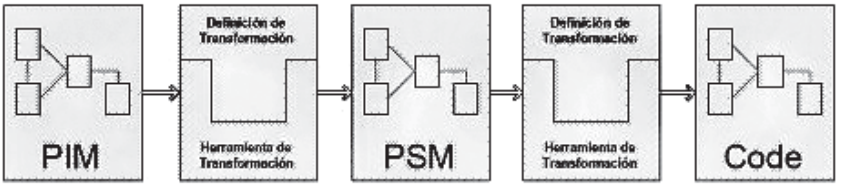

Figura 2-6. Las definiciones de transformaciones dentro de las herramientas de transformación

Si abriéramos la herramienta de transformación y mirásemos dentro, podríamos ver qué elementos están involucrados en la ejecución de la transformación. En algún lugar dentro de la herramienta hay una definición que describe como se debe transformar el modelo fuente para producir el modelo destino. Esta es la definición de la transformación. La figura 2-6 muestra la estructura de la herramienta de transformación. Notemos que hay una diferencia entre la transformación misma, que es el proceso de generar un nuevo modelo a partir de otro modelo, y la definición de la transformación.

Para especificar la transformación, (que será aplicada muchas veces, independientemente del modelo fuente al que será aplicada) se relacionan construcciones de un lenguaje fuente en construcciones de un lenguaje destino. Se podría, por ejemplo, definir una transformación que relaciona elementos de UML a elementos Java, la cual describiría como los elementos Java pueden ser generados a partir de los elementos UML. Esta situación se muestra en la figura 2-7.

En general, se puede decir que una definición de transformación consiste en una colección de reglas, las cuales son especificaciones no ambiguas de las formas en que un modelo (o parte de él) puede ser usado para crear otro modelo (o parte de él).

Figura 2-7. Definición de transformaciones entre lenguajes.

Las siguientes definiciones fueron extraídas del libro de Anneke Kepple [KWB 03]:

- *Una transformación es la generación automática de un modelo destino desde un modelo fuente, de acuerdo a una definición de transformación.*
- *Una definición de transformación es un conjunto de reglas de transformación que juntas describen como un modelo en el lenguaje fuente puede ser transformado en un modelo en el lenguaje destino.*
- *Una regla de transformación es una descripción de como una o más construcciones en el lenguaje fuente pueden ser transformadas en una o más construcciones en el lenguaje destino.*

2.2.1 ¿Cómo se define una transformación?

Una transformación entre modelos puede verse como un programa de computadora que toma un modelo como entrada y produce un modelo como salida. Por lo tanto las transformaciones podrían describirse (es decir, implementarse) utilizando cualquier lenguaje de programación, por ejemplo Java. Sin embargo, para simplificar la tarea de codificar transformaciones se han desarrollado lenguajes de más alto nivel (o específi-

cos del dominio de las transformaciones) para tal fin, tales como ATL [ATL] y QVT [QVT]. Este tema será desarrollado en detalle en el Capítulo 6.

2.2.2 Un ejemplo de transformación

Exhibiremos un ejemplo de una transformación de un modelo PIM escrito en UML a un modelo de implementación escrito en Java. Transformaremos el diagrama de clases de un sistema de venta de libros (Bookstore) en las clases de Java correspondientes a ese modelo. La figura 2-8 muestra gráficamente la transformación que se intenta realizar, la cual consta de 2 pasos: Transformaremos el PIM en un PSM y luego transformaremos el PSM resultante a código Java.

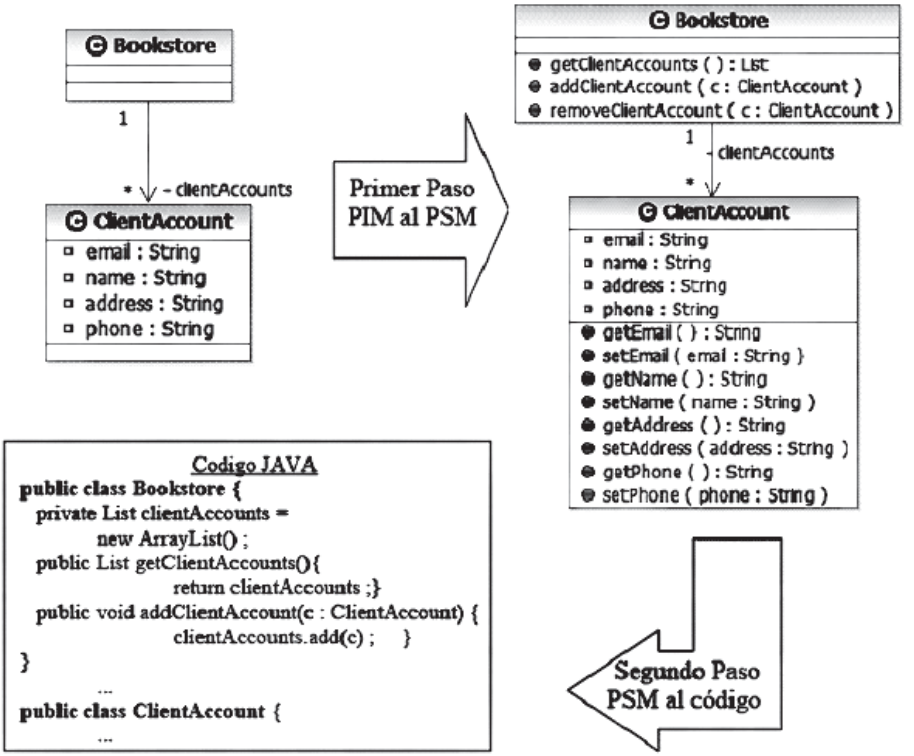

Figura 2-8. Ejemplo de transformación de PIM a PSM y de PSM a código.

Tanto el PIM como el PSM son modelos útiles ya que proveen el nivel de información adecuado para diferentes tipos de desarrolladores y otras personas involucradas en el proceso de desarrollo de software. Existe una clara relación entre estos modelos. La definición de la transformación que debe aplicarse para obtener el PSM a partir del PIM consiste en un conjunto de reglas. Estas reglas son:

- Para cada clase en el PIM se genera una clase con el mismo nombre en el PSM.
- Para cada relación de composición entre una clase llamada classA y otra clase llamada classB, con multiplicidad 1 a n, se genera un atributo en la clase classA con nombre classB de tipo Collection.
- Para cada atributo público definido como attributeName:Type en el PIM los siguientes atributos y operaciones se generan como parte de la clase destino:

    - Un atributo privado con el mismo nombre: attributeName: Type
    - Una operación pública cuyo nombre consta del nombre del atributo precedido con “get” y el tipo del atributo como tipo de retorno: getAttributeName(): Type
    - Una operación pública cuyo nombre consta del nombre del atributo precedido con “set” y con el atributo como parámetro y sin valor de retorno: setAttributeName(att: Type).

El siguiente paso consistirá en escribir una transformación que tome como entrada el PSM y lo transforme a código Java. Combinando y automatizando ambas transformaciones podremos generar código Java a partir del PIM.

2.3 Herramientas de soporte para MDD

La puesta en práctica del proceso MDD requiere de la disponibilidad de herramientas de software que den soporte a la creación de modelos y transformaciones. La figura 2-9 brinda un panorama de los distintos puntos en los que el proceso MDD necesita ser soportado por herramientas. En particular, necesitamos los siguientes elementos:

- Editores gráficos para crear los modelos ya sea usando UML como otros lenguajes de modelado específicos de dominio.
- Repositorios para persistir los modelos y manejar sus modificaciones y versiones.
- Herramientas para validar los modelos (consistencia, completitud, etc.).

- Editores de transformaciones de modelos que den soporte a los distintos lenguajes de transformación, como QVT o ATL.
- Compiladores de transformaciones, debuggers de transformaciones.
- Herramientas para verificar y/o testear las transformaciones.

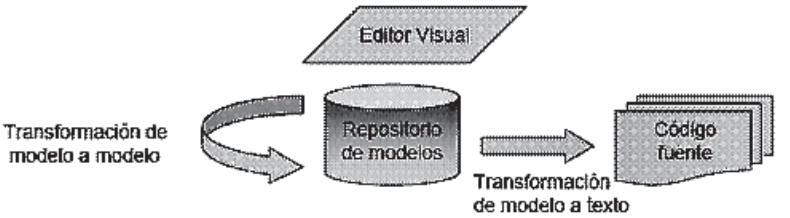

Figura 2-9. Herramientas de soporte para MDD.

2.4 Resumen

En este capítulo hemos descrito los elementos fundamentales que constituyen al proceso MDD, en particular:

- Modelos a diferentes niveles de abstracción, escritos en un lenguaje estándar y bien definido. Estos modelos deben ser consistentes, precisos y suficientemente completos. Algunos modelos serán muy abstractos, mientras que otros serán detallados y/o orientados a tecnologías específicas.
- Uno o más lenguajes estándar y bien definidos que nos permitan expresar modelos.
- Definiciones de cómo un modelo se transforma en otro modelo más específico. Las definiciones de estas transformaciones estarán estandarizadas y serán de dominio público, de manera que puedan ser reutilizadas por numerosos proyectos de desarrollo de software. Para ello se requiere que las transformaciones sean configurables o adaptables a las necesidades de cada usuario.
- Un lenguaje en el cual escribir la definición de las transformaciones. Este lenguaje será similar a un lenguaje de programación, en el sentido de que será interpretado por herramientas automáticas, y por lo tanto será un lenguaje formal.

- Herramientas de software que den soporte a la creación de modelos y transformaciones. Cada lenguaje de modelado debe contar con editores gráficos amigables para crear los modelos y repositorios para persistir los modelos y manejar sus modificaciones y versiones. Por otra parte se contará con herramientas que implementen la ejecución de las definiciones de transformación. Preferiblemente, estas herramientas deberán ofrecer la posibilidad de configurar o ajustar los efectos de la transformación a las necesidades específicas del usuario.

En los últimos años se ha observado un rápido avance en el desarrollo de los elementos mencionados arriba. Sin embargo algunos aspectos aún se encuentran en proceso de definición y/o de maduración. En los siguientes capítulos cada uno de estos elementos será examinado en más detalle.

CAPÍTULO 3

3. Definición formal de lenguajes de modelado. El rol del metamodelo

En este capítulo explicaremos los mecanismos para definir la sintaxis de los lenguajes con los cuales se crean los modelos de un sistema. En particular, discutiremos la técnica de metamodelado, la arquitectura de cuatro capas de modelado definida por el OMG y por qué el metamodelado es tan importante en el contexto de MDD.

3.1 Mecanismos para definir la sintaxis de un lenguaje de modelado

Hace algunos años, la sintaxis de los lenguajes se definía casi exclusivamente usando Backus Naur Form (BNF). Este formalismo es una meta sintaxis usada para expresar gramáticas libres de contexto, es decir, una manera formal de describir cuáles son las palabras básicas del lenguaje y cuáles secuencias de palabras forman una expresión correcta dentro del lenguaje. Una especificación en BNF es un sistema de reglas de la derivación, escrito como:

```
<símbolo> ::= <expresión con símbolos>
```

donde <símbolo> es un *no-terminal*, y la expresión consiste en secuencias de símbolos separadas por la barra vertical, '|', indicando una opción, cada una de las cuales es una posible substitución para el símbolo a la izquierda. Los símbolos que nunca aparecen en un lado izquierdo son *terminales*.
El BNF se utiliza extensamente como notación para definir la sintaxis (o gramática) de los lenguajes de programación. Por ejemplo, las siguien-

tes expresiones BNF definen la sintaxis de un lenguaje de programación simple:

```
<Programa> ::= "Begin" <Comando> "end"
<Comando> ::=
<Asignacion>|<Loop>|<Decision>|<Comando>";"<Comando>
<Asignacion>  ::=  variableName ":=" <Expresion>
<Loop> ::=  "while" <Expresión> "do" <Comando> "end"
<Decision> ::=
"if"<Expresión>"then"<Comando>"else"<Comando>"endif"
<Expresión> ::= ...
```

El siguiente código es una instancia válida de esta definición:

```
Begin
    if y=0
          then
              result:=0
          else
              result:=x;
              while (y>1) do result:=result+x; y:=y-1 end
    endif
end
```

Este método es útil para lenguajes textuales, pero dado que los lenguajes de modelado en general no están basados en texto, sino en gráficos, es conveniente recurrir a un mecanismo diferente para definirlos. Las principales diferencias entre los lenguajes basados en texto y los lenguajes basados en gráficos son las siguientes:

- **Contenedor vs. referencia**: En los lenguajes textuales una expresión puede formar parte de otra expresión, que a su vez puede estar contenida en otra expresión mayor. Esta relación de contenedor da origen a un árbol de expresiones. En el caso de los lenguajes gráficos, en lugar de un árbol se origina un grafo de expresiones ya que una sub-expresión puede ser referenciada desde dos o más expresiones diferentes.
- **Sintaxis concreta vs. sintaxis abstracta**. En los lenguajes textuales la sintaxis concreta coincide (casi) exactamente con la sintaxis abstracta mientras que en los lenguajes gráficos se presenta una marcada diferencia entre ambas.
- **Ausencia de una jerarquía clara en la estructura del lenguaje**. Los lenguajes textuales en general se aprehenden leyéndolos de arri-

ba hacia abajo y de izquierda a derecha, en cambio los lenguajes gráficos suelen asimilarse de manera diferente dependiendo de su sintaxis concreta (por ejemplo, comenzamos prestando atención al diagrama más grande y/o colorido). Esto influye en la jerarquía de la sintaxis abstracta, ocasionando que no siempre exista un orden entre las categorías sintácticas.

Por lo tanto, en los últimos años se desarrolló una técnica específica para facilitar la definición de los lenguajes gráficos, llamada "*metamodelado*". Veamos en que consiste esta nueva técnica: usando un lenguaje de modelado, podemos crear modelos; un modelo especifica que elementos pueden existir en un sistema. Si se define la clase Persona en un modelo, se pueden tener instancias de Persona como Juan, Pedro, etc. Por otro lado, la definición de un lenguaje de modelado establece que elementos pueden existir en un modelo. Por ejemplo, el lenguaje UML establece que dentro de un modelo se pueden usar los conceptos Clase, Atributo, Asociación, Paquete, etc. Debido a esta similitud, se puede describir un lenguaje por medio de un modelo, usualmente llamado "*metamodelo*". El metamodelo de un lenguaje describe que elementos pueden ser usados en el lenguaje y como pueden ser conectados.
Como un metamodelo es también un modelo, el metamodelo en sí mismo debe estar escrito en un lenguaje bien definido. Este lenguaje se llama metalenguaje. Desde este punto de vista, BNF es un metalenguaje. En la figura 3-1 se muestra gráficamente esta relación.

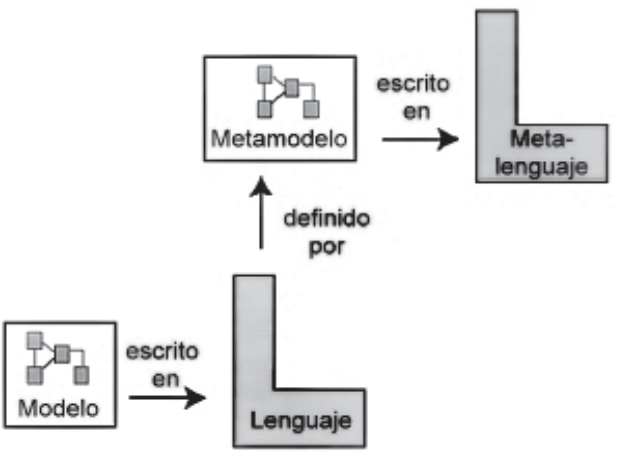

Figura 3-1. Modelos, Lenguajes, Metamodelos y Metalenguajes

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

3.1.1 La arquitectura de 4 capas de modelado del OMG

El metamodelado es entonces un mecanismo que permite definir formalmente lenguajes de modelado, como por ejemplo UML. La Arquitectura de cuatro capas de modelado es la propuesta del OMG orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos.
Los cuatro niveles definidos en esta arquitectura se denominan: M3, M2, M1, M0 y los describimos a continuación. Para entender mejor la relación entre los elementos en las distintas capas, presentamos un ejemplo utilizando el lenguaje UML. En este ejemplo modelamos un sistema de venta de libros por Internet, que maneja información acerca de los clientes y de los libros de los cuales se dispone.

Nivel M0: Instancias

En el nivel M0 se encuentran todas las instancias “reales” del sistema, es decir, los objetos de la aplicación. Aquí no se habla de clases, ni atributos, sino de entidades físicas que existen en el sistema.
En la figura 3-2 se muestra un diagrama de objetos UML donde pueden verse las distintas entidades que almacenan los datos necesarios para nuestro sistema. Se tiene una librería virtual, llamada ‘El Ateneo’. Esta librería tiene un cliente, Juan García, del cual queremos guardar su nombre de usuario, su palabra clave, su nombre real, su dirección postal y su dirección de e-mail. Además, la librería tiene una categoría de libros con nombre ‘Novela’. Por último, la librería tiene un libro con título ‘Cien Años de Soledad’ de la cual se conoce su autor.
Todas estas entidades son instancias pertenecientes a la capa M0.

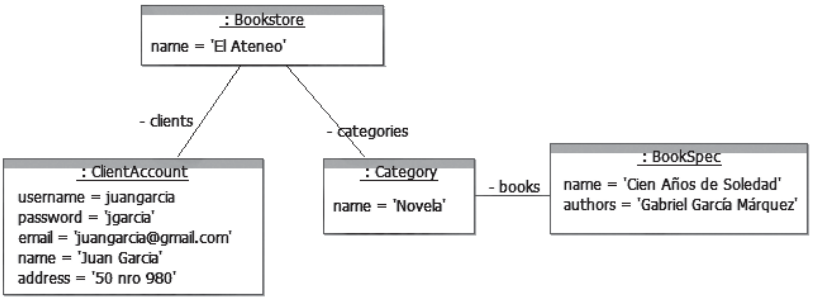

Figura 3-2. Entidades de la capa M0 del modelo de cuatro capas

Nivel M1: Modelo del sistema

Por encima de la capa M0 se sitúa la capa M1, que representa el modelo de un sistema de software. Los conceptos del nivel M1 representan categorías de las instancias de M0. Es decir, cada elemento de M0 es una instancia de un elemento de M1. Sus elementos son modelos de los datos. En el nivel M1 aparece entonces la entidad Librería, la cual representa las librerías del sistema, tales como 'El Ateneo', con los atributos nombre y dirección. Lo mismo ocurre con las entidades Cliente y Libro. En la figura 3-3 se muestra un modelo de clases UML para este ejemplo.

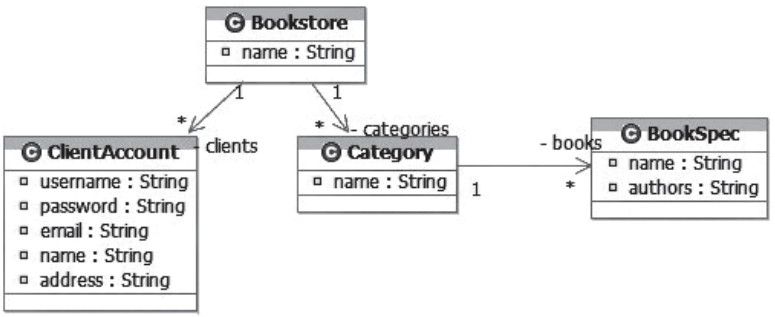

Figura 3-3. Modelo del sistema

El objeto con nombre 'El Ateneo' puede verse ahora como una instancia de Librería (Bookstore). De la misma manera, 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 figura 3-4 se muestra la relación entre el nivel M0 y el nivel M1.

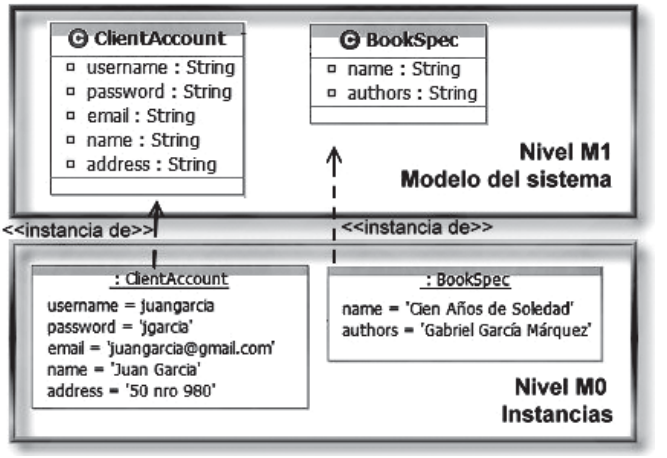

Figura 3-4. Relación entre el nivel M0 y el nivel M1

Nivel M2: Metamodelo

Análogamente a lo que ocurre con las capas M0 y M1, los elementos del nivel M1 son a su vez instancias del nivel M2. Esta capa recibe el nombre de metamodelo. La figura 3-5 muestra una parte del metamodelo UML. En este nivel aparecen conceptos tales como Clase (Class), Atributo (Attribute) y Operación (Operation).

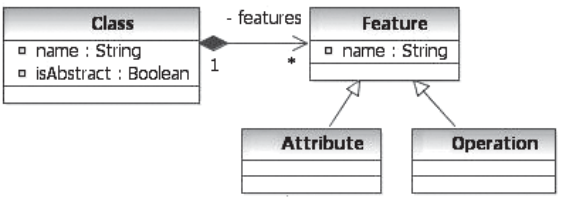

Figura 3-5. Parte del metamodelo UML

Siguiendo el ejemplo, la entidad ClientAccount será una instancia de la metaclase Class del metamodelo UML. Esta instancia tiene cinco objetos relacionados a través de la meta asociación feature, por ejemplo una instancia de Attribute con name='username' y type='String' y otra instancia de Attribute con name='password' y type='String'.
La figura 3-6 muestra la relación entre los elementos del nivel M1 con los elementos del nivel M2.

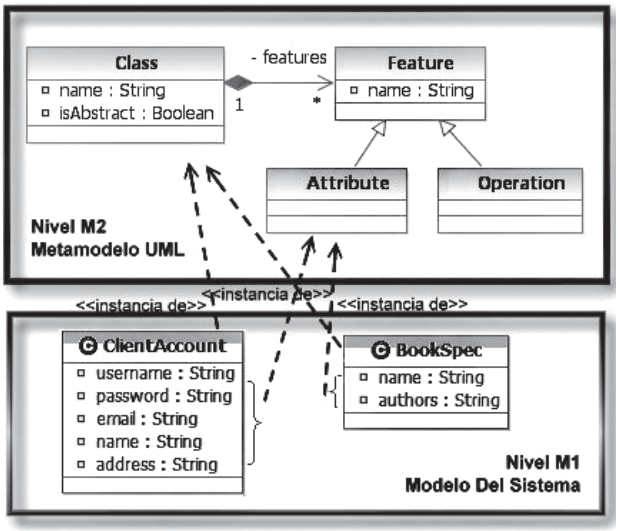

Figura 3-6. Relación entre el nivel M1 y el nivel M2

Nivel M3: Meta-metamodelo

De la misma manera podemos ver los elementos de M2 como instancias de otra capa, la capa M3 o capa de meta-metamodelo. Un meta-metamodelo es un modelo que define el lenguaje para representar un metamodelo. La relación entre un meta-metamodelo y un metamodelo es análoga a la relación entre un metamodelo y un modelo.
M3 es el nivel más abstracto, que permite definir metamodelos concretos. Dentro del OMG, MOF es el lenguaje estándar de la capa M3. Esto supone que todos los metamodelos de la capa M2, son instancias de MOF.
La figura 3-7 muestra la relación entre los elementos del metamodelo UML (Nivel M2) con los elementos de MOF (Nivel M3). Puede verse que las entidades de la capa M2 son instancias de las metaclases MOF de la capa M3.

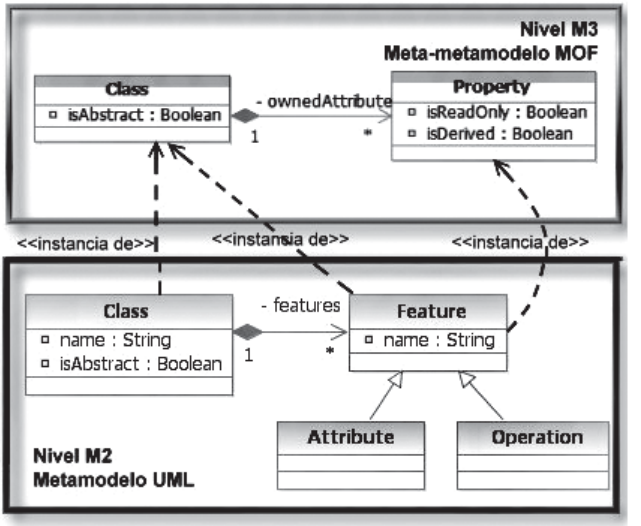

Figura 3-7. Relación entre el nivel M2 y el nivel M3

No existe otro meta nivel por encima de MOF. Básicamente, el MOF se define a sí mismo.
Por último, como vista general, la figura 3-8 muestra las cuatro capas de la arquitectura de modelado, indicando las relaciones entre los elementos en las diferentes capas. Puede verse que en la capa M3 se encuentra el meta-metamodelo MOF, a partir del cual se pueden definir distintos metamodelos en el nivel M2, como UML, Java, OCL, etc. Instancias de estos metamodelos serán los elementos del nivel M1, como modelos UML, o modelos Java. A su vez, instancias de los elementos M1 serán los objetos que cobran vida en las corridas del sistema.

3.1.2 El uso del metamodelado en MDD

Las razones por las que el metamodelado es tan importante en MDD son:
- En primer lugar, necesitamos contar con un mecanismo para definir lenguajes de modelado sin ambigüedades [CESW 04] y permitir que una herramienta de transformación pueda leer, escribir y entender los modelos.
- Luego, las reglas de transformación que constituyen una definición de una transformación describen como un modelo en un lenguaje fuente puede ser transformado a un modelo en un lenguaje destino. Estas reglas usan los metamodelos de los lenguajes fuente y destino para definir la transformación.

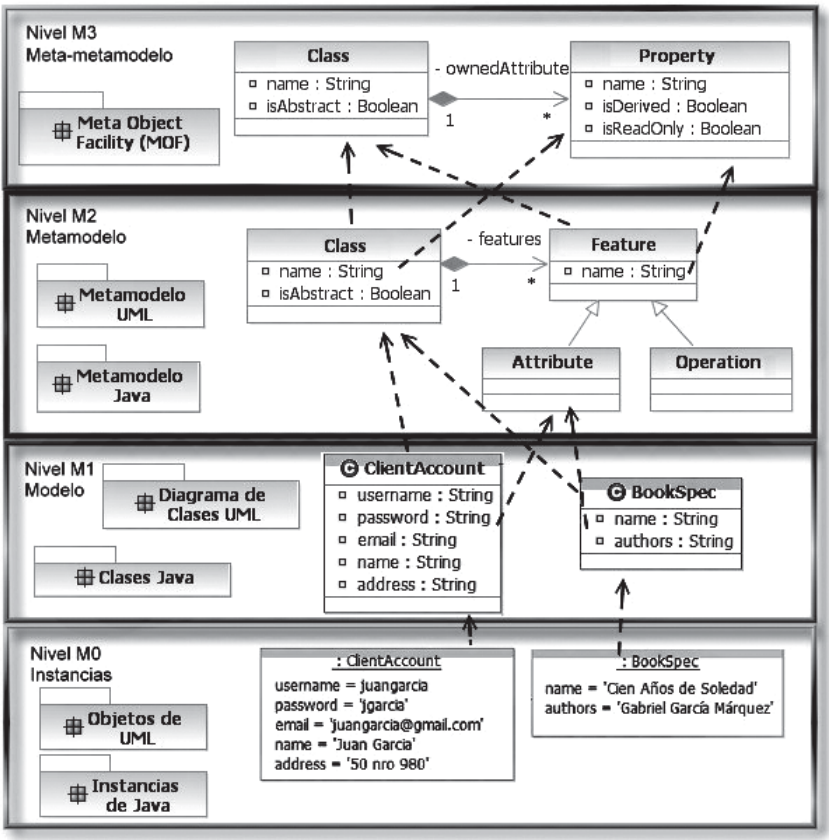

Figura 3-8. Vista general de las relaciones entre los 4 niveles

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

La figura 3-9 muestra como se completa MDD con la capa de metamodelado. La parte baja de la figura muestra los elementos con los que la mayoría de los desarrolladores trabaja habitualmente. En el centro se introduce el metalenguaje para definir nuevos lenguajes. Un pequeño grupo de desarrolladores, usualmente con más experiencia, necesitarán definir lenguajes y las transformaciones entre estos lenguajes. Para este grupo, entender el metanivel será esencial a la hora de definir transformaciones.

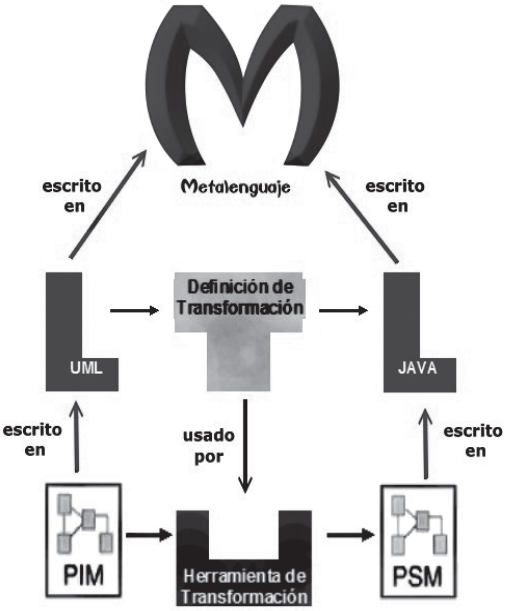

Figura 3-9. MDD incluyendo el metalenguaje

3.2 El lenguaje de modelado más abstracto: MOF

El lenguaje MOF, acrónimo de Meta-Object Facility, es un estándar del OMG para la ingeniería conducida por modelos. Como se vio anteriormente, MOF se encuentra en la capa superior de la arquitectura de 4 capas. Provee un meta-meta lenguaje que permite definir metamodelos en la capa M2. El ejemplo más popular de un elemento en la capa M2 es el metamodelo UML, que describe al lenguaje UML.

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

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

Actualmente, la definición de MOF está separada en dos partes fundamentales, EMOF (*Essential MOF*) y CMOF (*Complete MOF*), y se espera que en el futuro se agregue SMOF (*Semantic MOF*). La figura 3-10

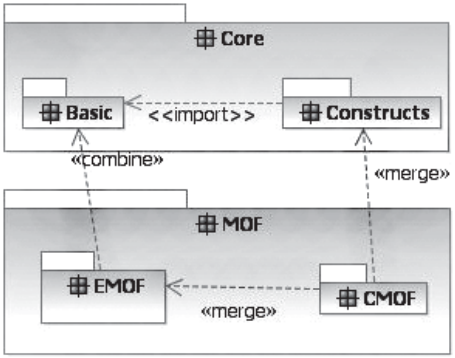

Figura 3-10. El lenguaje MOF

muestra la relación entre EMOF y CMOF. Ambos paquetes importan los elementos de un paquete en común, del cual utilizan los constructores básicos y los extienden con los elementos necesarios para definir metamodelos simples, en el caso de EMOF y metamodelos más sofisticados, en el caso de CMOF. La figura 3-11 presenta los principales elementos contenidos en el paquete EMOF.

3.2.1 MOF vs. BNF

Si bien la meta sintaxis BNF y el meta lenguaje MOF son formalismos creados con el objetivo de definir lenguajes textuales y lenguajes gráficos respectivamente, Ivan Porres y otros en [AP 03] y [WK 05] han demostrado que ambos formalismos poseen igual poder expresivo, siendo posible la transformación bi-direccional de sentencias cuya sintaxis fue expresada en BNF y sus correspondientes modelos cuya sintaxis fue expresada en MOF. Por ejemplo, ya hemos visto como expresar la sintaxis abstracta de UML usando MOF; el siguiente código escrito en BNF define al mismo extracto de UML mostrado anteriormente (compárese con la figura 3-5):

```
<Class>::= "name" String "isAbstract" Bool
           "feature"<Feature>*
<Feature>   ::=  <Attribute> | <Operation>
<Attribute> ::= "name" String "type" String
<Operation> ::= "name" String "type" String
```

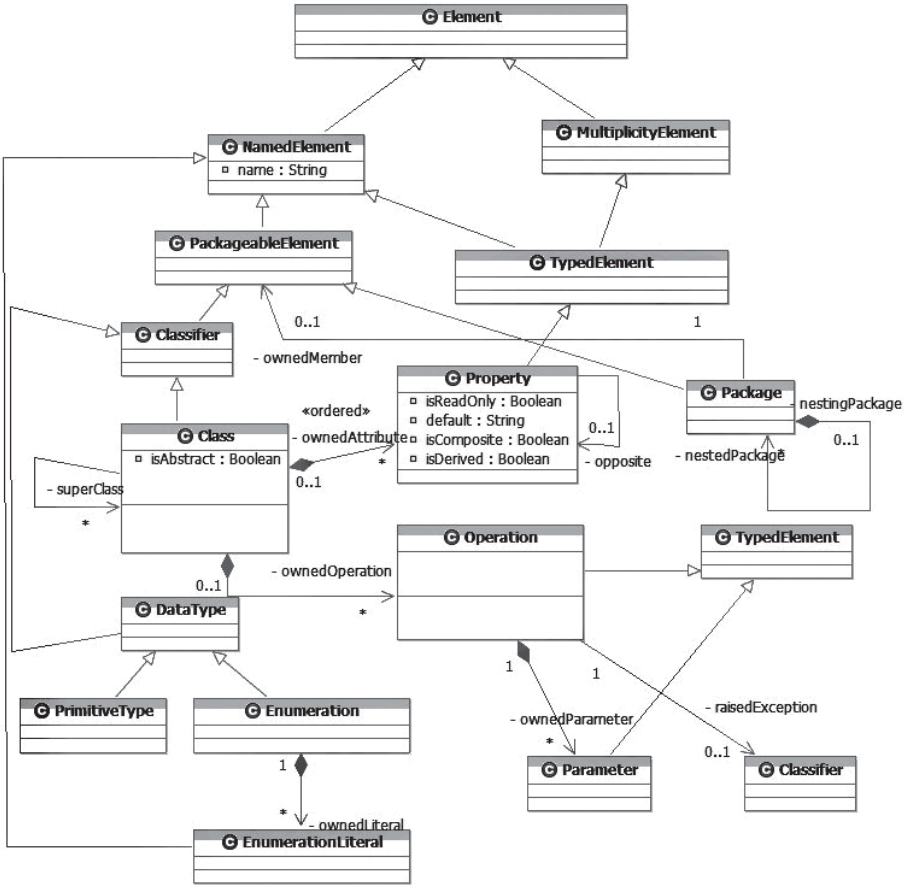

Figura 3-11. Sintaxis abstracta de MOF

3.2.2 MOF vs. UML

Actualmente la sintaxis de UML está definida usando MOF, sin embargo UML fue creado antes que MOF. Inicialmente UML no estaba formalmente definido, su sintaxis sólo estaba descrita informalmente a través de ejemplos. MOF surgió posteriormente, con el objetivo de proveer un marco formal para la definición de UML y otros lenguajes gráficos.
La sintaxis concreta de MOF coincide con la de UML, lo cual resulta algo confuso para los principiantes en el uso de estos lenguajes. Además MOF contiene algunos elementos también presentes en UML, por ejem-

plo, ambos lenguajes tienen un elemento llamado Class. A pesar de que los elementos tienen el mismo nombre y superficialmente describen al mismo concepto, no son idénticos y no deben ser confundidos.

3.2.3 Implementación de MOF-Ecore

El metamodelo MOF está implementado mediante un plugin para Eclipse [Eclipse] llamado Ecore [EMF]. Este plugin respeta las metaclases definidas por MOF. Todas las metaclases mantienen el nombre del elemento que implementan y agregan como prefijo la letra "E", indicando que pertenecen al metamodelo Ecore. Por ejemplo, la metaclase EClass implementa a la metaclase Class de MOF.
La primera implementación de Ecore fue terminada en Junio del 2002. Esta primera versión usaba como meta-metamodelo la definición estándar de MOF (v1.4). Gracias a las sucesivas implementaciones de Ecore y basado en la experiencia ganada con el uso de esta implementación en varias herramientas, el metalenguaje evolucionó. Como conclusión, se logró una implementación de Ecore realizada en Java, más eficiente y con sólo un subconjunto de MOF, y no con todos los elementos como manipulaban las implementaciones hechas hasta ese momento. A partir de este conocimiento, el grupo de desarrollo de Ecore colaboró más activamente con la nueva definición de MOF, y se estableció un subconjunto de elementos como esenciales, llegando a la definición de EMOF, que fue incluida como parte del MOF (v2.0).
MOF y Ecore son conceptualmente muy similares, ambos basados en el concepto de clases con atributos tipados y operaciones con parámetros y excepciones. Además ambos soportan herencia múltiple y utilizan paquetes como mecanismo de agrupamiento. Para entender en detalle las diferencias entre MOF y Ecore, puede consultarse el trabajo de Anna Gerber [GR 03].

3.3 Ejemplos de metamodelos

MOF puede ser usado para definir metamodelos de lenguajes orientados a objetos, como es el caso de UML, y también para otros lenguajes no orientados a objetos, como es el caso de las redes de Petri o los lenguajes para servicios web.
En esta sección mostramos algunos ejemplos de metamodelos, en particular la figura 3-12 muestra el metamodelo simplificado del lenguaje UML donde se especifica que un modelo UML está formado por

paquetes (Package), donde cada paquete está integrado por clases (Clase) que poseen atributos (Attribute). Los atributos tienen un nombre (name) y un tipo (type), que puede ser un tipo de dato primitivo (PrimitiveDataType) o una clase.

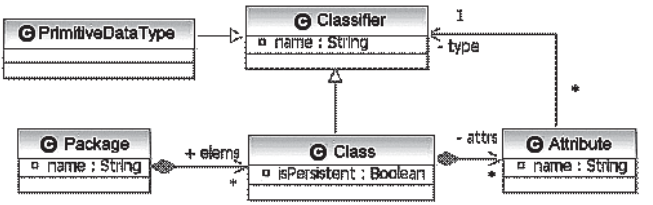

Figura 3-12. Metamodelo simplificado de UML

El metamodelo del lenguaje RDBMS que mostramos en la figura 3-13 indica que un modelo relacional consta de esquemas (Schema), donde cada esquema está compuesto por tablas (Table) que tienen columnas (Column). Las columnas tienen un nombre (name) y un tipo de dato (type). La tabla tiene una clave primaria (pkey) y cero o más claves foráneas (fkeys). Cada clave foránea (Kkey) hace referencia a columnas en otras tablas.

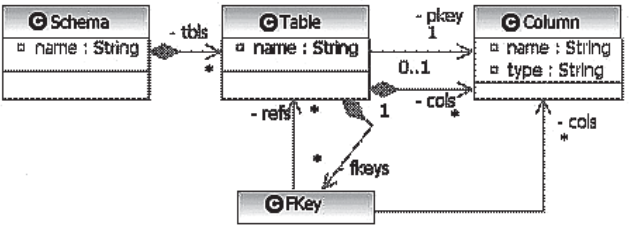

Figura 3-13. Metamodelo simplificado del lenguaje de las bases de datos relacionales (RDBMS)

3.4 El rol de OCL en el metamodelado

Los lenguajes gráficos de modelado han sido ampliamente aceptados en la industria. Sin embargo su falta de precisión ha originado la necesidad de utilizar otros lenguajes de especificación, como OCL [WK 03], para definir restricciones adicionales.
Cualquier modelo puede ser enriquecido mediante información adicional o restricciones sobre sus elementos. Estas restricciones pueden ser escritas en lenguaje natural, pero en ese caso pueden ser mal interpretadas y además no es posible chequear su validez en el modelo. Otra alternativa es utilizar lenguajes formales, sin embargo estos resultan difíciles de usar y no son bien recibidos por los modeladores de sistemas.
OCL fue desarrollado para solucionar este problema. Es un lenguaje formal que por su naturaleza orientada a objetos resulta fácil de entender para las personas con conocimiento de lenguajes orientados a objetos tales como Java o Smalltalk. OCL es un lenguaje puro de especificación, el cual garantiza estar libre de efectos laterales. Cuando se evalúa una expresión OCL, simplemente retorna un valor, sin hacer modificaciones en el modelo.
OCL no es un lenguaje de programación, es decir, no es posible escribir la lógica de un programa o flujos de control. No se pueden invocar procesos o activar operaciones que no sean de consulta.
OCL es un lenguaje tipado, por lo tanto, cada expresión tiene un tipo. Para que una expresión esté bien formada, debe ajustarse a las reglas de operaciones entre tipos del lenguaje; por ejemplo, no puede compararse un Integer con un String.
Con el uso de OCL se ofrece al diseñador la posibilidad de crear modelos precisos y completos del sistema en etapas tempranas del desarrollo. Sin embargo para estimular su uso en la industria es necesario contar con herramientas que permitan la edición y evaluación de las especificaciones expresadas en OCL.

3.4.1 Especificación de restricciones en OCL

OCL permite especificar diferentes tipos de restricciones con las cuales podemos definir reglas de buena formación (conocidas como wfr por su nombre en inglés: well-formedness rules) para el modelo.
Veamos las principales construcciones sintácticas del OCL:

Invariantes

Si una expresión OCL es una invariante para un elemento, debe resultar verdadera para todas las instancias del elemento en todo momento. El elemento sobre el cual predica la expresión OCL, forma parte de la invariante y se escribe bajo la palabra clave ***context*** seguido del nombre del elemento. La etiqueta ***inv***: declara que la restricción es una invariante. Por ejemplo, podría interesarnos especificar que todos los clientes de nuestro Bookstore han definido su password. En este caso la invariante debe definirse sobre la clase ClientAccount de la siguiente manera:

```
context ClientAccount
inv: self.password<>"
```

En general, la sintaxis de una invariante es:

```
context [VariableName:] TypeName
inv: OclExpression
```

Pre y Post condiciones

Una pre o post condición debe estar ligada a una operación o método. Se muestra agregando "pre" o "post" en la declaración según corresponda. En este caso, la instancia contextual *self* se refiere al elemento al que pertenece la operación. En el caso de ser una pre condición, la restricción establece una condición que debe cumplirse antes de ejecutar la operación. En el caso de ser una post condición, la restricción establece una condición que debe cumplirse después de ejecutar la operación.
En general, la sintaxis para definir pre y post condiciones es la siguiente:

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
pre: oclExpression
post: oclExpression
```

Si una expresión OCL es definida como invariante, la restricción debe cumplirse en todo momento, en cambio en las pre condiciones y post condiciones debe cumplirse antes y después de ejecutar la operación, según sea el caso. En una expresión OCL utilizada como post condición los elementos se pueden decorar con el postfijo "@pre" para hacer referencia al valor del elemento al comienzo de la operación. La variable especial *result* se refiere al valor de retorno de la operación.

Por ejemplo, la siguiente expresión OCL describe la pre y post condición de una operación de la clase ClientAccount que permite modificar el valor de la palabra clave del cliente:

```
context ClientAccount::modifyPassword (oldPassword: String,
newPassword: String): Boolean
pre : oldPassword=self.password
post: self.password=newPassword and result=true
```

Package Context

OCL brinda un mecanismo de empaquetamiento para agrupar y organizar las expresiones. Las expresiones agrupadas deben ser encerradas entre las palabras claves 'package' y 'endpackage', Es posible definir paquetes anidados. La sintaxis genérica es la siguiente:

```
package PackageName::SubPackageName::...
      context ClassName inv:
      ... alguna invariante ...

      context ClassName::operationName(..)
      pre: ... alguna pre condición ...
endpackage
```

3.4.2 Usando OCL para expresar reglas de buena formación

OCL puede utilizarse para expresar cuáles son las condiciones estructurales que las construcciones del lenguaje deben satisfacer para estar bien formadas. Estas reglas pueden definirse en cada uno de los niveles de la arquitectura de 4 capas, como veremos a continuación.

Reglas de buena formación a nivel modelo:

Las invariante a nivel modelo aseguran que los objetos a ser instanciados cumplen determinadas propiedades. Por ejemplo, en el contexto de ClientAccount, una regla de buena formación sería la siguiente.

```
context ClientAccount
    inv: self.email.contains('@')
```

Self en este caso se refiere a una instancia de la cuenta de un cliente. Esta invariante debe cumplirse para cada uno de los clientes concretos, es decir, para cada una de las instancias de la clase ClientAccount. Esta invariante chequea que los emails de los clientes contengan el carácter '@'.
Puede observarse que para definir una propiedad sobre los objetos del nivel M0 es necesario escribir una invariante en la clase de la cuál esos objetos son instancias.

Reglas de buena formación a nivel metamodelo:
De manera análoga, si quisiéramos definir restricciones para los modelos, deberíamos escribir invariantes en las clases de las cuales los modelos son instancias, es decir en los metamodelos. Estas restricciones predican sobre los meta-elementos definidos en el metamodelo, por ejemplo, dado el metamodelo relacional de la figura 3-13, se predica sobre Table, Schema, etc. Un ejemplo para este tipo de reglas es que en una tabla no deben existir columnas con nombre repetido o que la clave primaria debe corresponderse con una columna de la tabla. Un ejemplo concreto es el siguiente:

```
context Table
   inv:
    —[1] Las columnas en una tabla deben tener distinto nombre
     self.cols -> forAll(p,q| p.name = q.name implies p = q)
```

Self en este caso se refiere a una instancia de Table. Esta invariante debe cumplirse para cada una de las instancias de Table, es decir, para cada una de las tablas definidas en el modelo relacional.
Este tipo de invariantes complementa la definición de la sintaxis de los lenguajes de modelado, agregando restricciones que no pueden ser representadas gráficamente. Por ejemplo, una tabla con dos columnas de igual nombre es considerada sintácticamente incorrecta, dado que no satisface la regla de buena formación.

Reglas de buena formación a nivel meta-metamodelo:
Finalmente, también es necesario escribir reglas de buena formación sobre los meta-meta-elementos o elementos de MOF. Las reglas de buena formación a nivel meta-metamodelo son restricciones que aseguran que un metamodelo está bien formado. Un ejemplo para esta clase de reglas es que un paquete no contenga metaclases con el mismo

nombre o que una metaclase no tenga meta-atributos repetidos. Estas reglas restringen la sintaxis de MOF. Por ejemplo, la siguiente regla, definida sobre el Ecore, valida que todas las metaclases pertenecientes a un paquete tengan distinto nombre:

```
context EPackage
   inv WFR_1_EPackage:
    —[2]The EClasses must have different names.
    self.eClassifiers -> select(c | c.oclIsKindOf(EClass) )
        -> forAll (c, c2 | c.name = c2.name implies c = c2)
```

Entonces, de acuerdo con esta regla, un paquete conteniendo dos clases de igual nombre no constituye una instancia bien formada del lenguaje MOF.

3.5 Resumen

En este capítulo hemos considerado los mecanismos para definir la sintaxis de los lenguajes de modelado. En particular hemos comenzado mencionando a la técnica clásica consistente en utilizar Backus Naur Form (BNF). Este método es útil para lenguajes textuales, pero dado que los lenguajes de modelado actuales en general no están basados en texto, sino en gráficos, es conveniente recurrir a un mecanismo diferente. En los últimos años se desarrolló una técnica específica para facilitar la definición de los lenguajes gráficos, llamada "*metamodelado*". El metamodelo describe la sintaxis abstracta del lenguaje y constituye la base para el procesamiento automatizado de los modelos.
Esta técnica se basa en la idea de que un lenguaje de modelado también puede ser definido mediante un modelo conocido como metamodelo. Esta definición recursiva da lugar a una arquitectura de capas, en la cual cada nivel se instancia a partir del nivel inmediato superior. La arquitectura de cuatro capas de modelado es la propuesta del OMG orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos. Los cuatro niveles definidos en esta arquitectura se denominan: M3, M2, M1, M0.
Finalmente, como complemento al metamodelado, enfatizamos la necesidad de contar con un lenguaje formal, por ejemplo OCL. Este lenguaje formal nos permite agregar mayor precisión tanto en los modelos como en los metamodelos, lo cual constituye un requisito para el tratamiento automatizado de los artefactos en MDD.

CAPÍTULO 4

4. Herramientas de soporte a la creación de modelos de software

La ingeniería de software dirigida por modelos no tendría sentido sin la disponibilidad de herramientas que brinden el soporte apropiado para la manipulación de los distintos tipos de modelos. Estas herramientas deben facilitarle al usuario la tarea de definir nuevos lenguajes de modelado, permitiendo definir la sintaxis del lenguaje tanto la abstracta como la concreta, para facilitar la creación de modelos.
En este capítulo presentaremos algunas herramientas que pueden ser usadas en el contexto de un proyecto MDD para definir lenguajes de modelado.

4.1 Herramientas de modelado vs. herramientas de metamodelado

La mayoría de las herramientas de modelado que se utilizan actualmente, tanto las herramientas que soportan al lenguaje UML (por ejemplo ArgoUML [ArgoUML] y Rational Modeler [Rational Modeler]) como las herramientas que brindan soporte a otros lenguajes de modelado (por ejemplo diagramas de entidad relación [ER]), están basadas en una arquitectura de dos niveles (figura 4-1): los modelos son almacenados en archivos o en un repositorio cuyo esquema está programado y compilado dentro de la herramienta. Esto determina la clase de modelos que se pueden hacer y la manera en que se procesan y no puede ser modificado. Solamente la empresa proveedora de la herramienta puede modificar el lenguaje de modelado, ya que está definido en el código.

La tecnología basada en metamodelos elimina esta limitación y permite lenguajes de modelado flexibles. Esto se lleva a cabo adoptando la arquitectura de capas definida por el OMG que describimos en el capítulo anterior, lo cual agrega un nivel sobre el nivel de los lenguajes de modelado, como se ve en la figura 4-1. El nivel más bajo, el nivel del modelo, es similar al de las herramientas de modelado. El nivel intermedio contiene el modelo del lenguaje, es decir, el metamodelo. A diferencia de las demás herramientas, una herramienta basada en metamodelo permite al usuario acceder y modificar las especificaciones del lenguaje. Esto es posible ya que se tiene un nivel superior (correspondiente a M3 en la arquitectura del OMG) que incluye al lenguaje de metamodelado para especificar lenguajes de modelado. Este nivel es la parte fija de la herramienta. Actualmente la mayoría de las herramientas utilizan alguna variante de MOF como lenguaje de metamodelado en su capa superior.

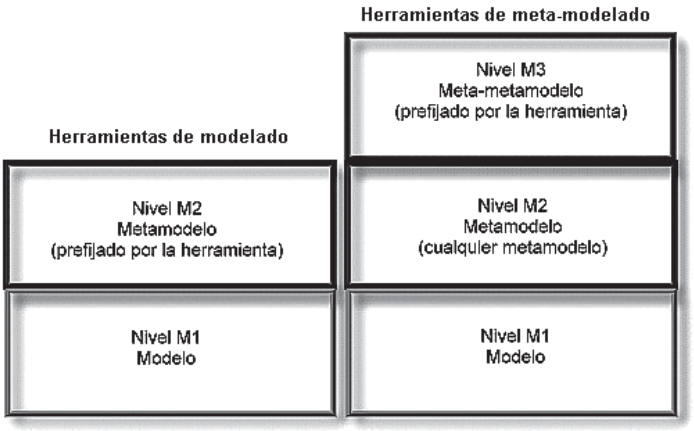

Figura 4-1. Herramientas de modelado en la Arquitectura 4 capas

4.2 ¿Qué debe proveer una herramienta de soporte al metamodelado?

Antes de mostrar las herramientas para dar soporte a un lenguaje de modelado es útil discutir la funcionalidad necesaria que deben proveer:

- Para poder definir un nuevo lenguaje de modelado, es necesario que la herramienta nos permita especificar la sintaxis abstracta del lenguaje, es decir que nos permita definir su metamodelo indicando cuá-

les son los elementos del lenguaje y sus relaciones, así como también las propiedades para cada elemento.

- Generalmente, cuando se define un nuevo lenguaje también se definen restricciones que se deben verificar para que las instancias de este nuevo lenguaje estén correctamente formadas. Por lo tanto, es preciso que la herramienta permita especificar reglas básicas, como por ejemplo, que objetos se pueden conectar a través de cuál relación.
- Para que este lenguaje sea más amigable al usuario, sería deseable que se pudiera instanciar de manera gráfica. Para esto, debe permitirse que en su definición sea posible especificar símbolos gráficos para los elementos, de manera gráfica, declarativa o en código.
- Debe ofrecer al menos la funcionalidad básica esperada de cualquier herramienta de edición. Estas funciones incluyen almacenar y recuperar un modelo del disco, deshacer y rehacer de muchos niveles, cortar, copiar, pegar, borrar. También deberían permitir manipular directamente los elementos a editar, imprimir y exportar en varios formatos, distribuir los elementos en el diagrama de manera automática, visualizar en distinta escala (zoom).
- En algunas herramientas se requiere que el metamodelador especifique los iconos, el tipo de paleta y a veces los menús en forma manual. Incluso en algunas herramientas se pedía al metamodelador que cree iconos de diferentes tamaños para diferentes usos, plataformas o configuraciones de la pantalla. Para incrementa la productividad del metamodelador es necesario automatizar estas tareas, por ejemplo brindarle la posibilidad de que un icono escale automáticamente a una variedad de tamaños. Generar todos estos elementos automáticamente asegura que todas las partes de la herramienta de modelado permanezcan sincronizadas con la definición del lenguaje de modelado. Por lo tanto, sería deseable que la herramienta proveyera una definición automática de los iconos, la paleta, los menús y los diálogos.
- Actualmente la probabilidad de trabajar con una sola herramienta en el desarrollo de un software es baja. Por lo tanto debe ser posible el intercambio de metamodelos y modelos, es decir la importación y exportación de modelos y metamodelos para el intercambio de información entre otras instancias de la herramienta y de otras herramientas.
- Es muy común que a medida que se use un lenguaje se noten deficiencias que hacen necesario cambiar su definición. Por lo tanto, es muy útil que la herramienta permita modificar los metamodelos cuando existen instancias de estos, actualizando sus instancias automáticamente. Esto permite que el metamodelador trabaje para perfeccionar la definición de su metamodelo, mientras construye el

lenguaje, y que aún antes de haber terminado los modeladores ya puedan utilizarlo.

Lo más importante, es que estas herramientas reducen el trabajo requerido para desarrollar una herramienta que soporte un nuevo lenguaje de modelado. La tarea es actualmente mucho más sencilla. Simplemente hay que definir la sintaxis del lenguaje y a partir de ella, la herramienta generará automáticamente el editor para crear instancias de ese lenguaje, sin que se tenga que escribir ni una sola línea de código.
Como mencionamos anteriormente, las herramientas de modelado basadas en metamodelo han adoptado una arquitectura de 4 capas, sin embargo no todas ellas han aceptado al lenguaje MOF como el estándar para metamodelado y en lugar de ello utilizan sus propias notaciones. Esto origina diferencias difíciles de conciliar entre las propuestas actuales dificultando la interoperabilidad entre los modelos creados utilizando distintas herramientas.
En las siguientes secciones describiremos tres de los principales enfoques: la propuesta de Eclipse que adopta una versión simplificada de MOF llamada Ecore, la propuesta de Metacase que usa un lenguaje de metamodelado llamado GOPPRR y finalmente la propuesta de Microsoft a través de sus DSL Tools.

4.3 La propuesta de Eclipse

Eclipse es principalmente una comunidad que fomenta el código abierto. Los proyectos en los que trabaja se enfocan principalmente en construir una plataforma de desarrollo abierta. Esta plataforma puede ejecutarse en múltiples sistemas operativos, lo que convierte a Eclipse en una multi-plataforma. Está compuesta por frameworks extensibles, y otras herramientas que permiten construir y administrar software. Permite que los usuarios colaboren para mejorar la plataforma existente y extiendan su funcionalidad a través de la creación de plugins.
Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas VisualAge. Inicialmente fue un entorno de desarrollo para Java, escrito en Java. Pero luego fue extendiendo el soporte a otros lenguajes de programación. Eclipse ganó popularidad debido a que la plataforma Eclipse proporciona el código fuente de la plataforma y esta transparencia generó confianza en los usuarios.
Hace algunos años se creó la Fundación Eclipse, una organización independiente sin fines de lucro que fomenta una comunidad de código abierto. La misma trabaja sobre un conjunto de productos complemen-

tarios, capacidades y servicios que respalda y se encarga del desarrollo continuo de la plataforma. La comunidad Eclipse trabaja actualmente en 60 proyectos. En este capítulo solamente pondremos foco en el proyecto dedicado a promover tecnologías de desarrollo basadas en modelos llamado Eclipse Modeling Project (http://www.eclipse.org/modeling/). Este proyecto está compuesto por otros subproyectos relacionados. En la figura 4-2 se listan los subproyectos que forman Eclipse Modeling Project.

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

Figura 4-2. Subproyectos de Eclipse Modeling Project

En este capítulo presentamos los plugins para crear la sintaxis abstracta y concreta de un lenguaje. Para la sintaxis abstracta se presentarán los plugins EMF y OCL, mientras que para la sintaxis concreta se presentarán GMF y TMF. Tanto EMF como GMF son proyectos maduros y sus implementaciones han ido evolucionando los últimos años. La primera versión de EMF se lanzó en el año 2003 y la primera versión de GMF en el año 2006. En contraste, TMF es un proyecto nuevo, que aún se encuentra en las primeras etapas.
En las secciones siguientes se presentan estos proyectos con más detalle.

4.3.1 Eclipse Modeling Framework (EMF)

El proyecto EMF es un framework para modelado, que permite la generación automática de código para construir herramientas y otras aplicaciones a partir de modelos de datos estructurados. La información referida a este proyecto, así como también la descarga del plugin pueden encontrarse en [EMF].
EMF comenzó como una implementación del metalenguaje MOF. En los últimos años se usó para implementar una gran cantidad de herramientas lo que permitió mejorar la eficiencia del código generado. Actualmente el uso de EMF para desarrollar herramientas está muy extendido.

Se usa, por ejemplo, para implementar XML Schema Infoset Modelo (XSD), Servicio de Data Objects (SDO), UML2, y Web Tools Platform (WTP) para los proyectos Eclipse. Además EMF se utiliza en productos comerciales, como Omondo, EclipseUML, IBM Rational y productos WebSphere.
EMF permite usar un modelo como el punto de partida para la generación de código, e iterativamente refinar el modelo y regenerar el código, hasta obtener el código requerido. Aunque también prevé la posibilidad de que el programador necesite modificar ese código, es decir, se contempla la posibilidad de que el usuario edite las clases generadas, para agregar o editar métodos y variables de instancia. Siempre se puede regenerar desde el modelo cuando se necesite, y las partes agregadas serán preservadas durante la regeneración.

Descripción

Como se dijo anteriormente, la generación de código es posible a partir de la especificación de un modelo. De esta manera, permite que el desarrollador se concentre en el modelo y delegue en el framework los detalles de la implementación.
El código generado incluye clases Java para manipular instancias de ese modelo como así también clases adaptadoras para visualizar y editar las propiedades de las instancias desde la vista “propiedades” de Eclipse. Además se provee un editor básico en forma de árbol para crear instancias del modelo. Y por último, incluye un conjunto de casos de prueba para permitir verificar propiedades. En la figura 4-3 pueden verse dos pantallas. La de la izquierda muestra los cuatro plugins generados con EMF. Dentro de los plugins pueden verse los paquetes y las clases correspondientes. La pantalla de la derecha, muestra el editor en forma de árbol.
El código generado por EMF es eficiente, correcto y fácilmente modificable. El mismo provee un mecanismo de notificación de cambios de los elementos, una implementación propia de operaciones reflexivas y persistencia de instancias del modelo. Además provee un soporte básico para rehacer y deshacer las acciones realizadas. Por último, establece un soporte para interoperabilidad con otras herramientas en el framework de Eclipse, incluyendo facilidades para generar editores basados en Eclipse y RCP.

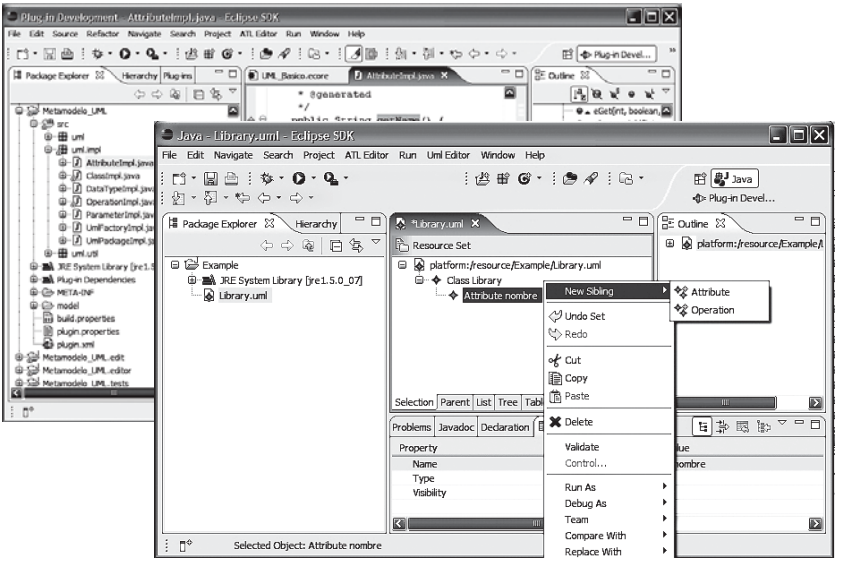

Figura 4-3. Plugins generados con EMF

Meta- metamodelo

En EMF los modelos se especifican usando un meta metamodelo llamado Ecore. Ecore es una implementación de eMOF (Essential MOF). Ecore en sí, es un modelo EMF y su propio metamodelo. Existen algunas diferencias entre Ecore y EMOF, pero aún así, EMF puede leer y escribir serializaciones de EMOF haciendo posible un intercambio de datos entre herramientas. La figura 4-4 muestra las clases más importantes del meta metamodelo Ecore.

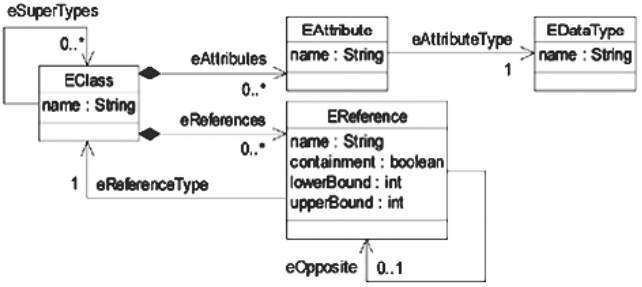

Figura 4-4. Parte del meta metamodelo Ecore

Ecore respeta las metaclases definidas por EMOF: todas las metaclases mantienen el nombre del elemento que implementan y agregan como prefijo la letra “E”, indicando que pertenecen al metamodelo Ecore. Por ejemplo, la metaclase EClass implementa la metaclase Class de EMOF. La principal diferencia está en el tratamiento de las relaciones entre las clases. MOF tiene a la asociación como concepto primario, definiéndola como una relación binaria entre clases. Tiene finales de asociaciones, con la propiedad de navegabilidad. En cambio, Ecore define solamente EReferences, como un rol de una asociación, sin finales de asociación ni Association como metaclases. Dos EReferences pueden definirse como opuestas para establecer una relación navegable para ambos sentidos. Existen ventajas y desventajas para esta implementación. Como ventaja puede verse que las relaciones simétricas, como por ejemplo “esposoDe”, implementadas con Association, son difíciles de mantener ya que debe hacerse consistentemente. En cambio con Ecore, al ser sólo una referencia, ella misma es su opuesto, es decir, se captura mejor la semántica de las asociaciones simétricas, y no es necesario mantener la consistencia en el otro sentido. Los modelos son entonces, instancias del metamodelo Ecore. Estos modelos son guardados en formato XMI (XML Metadata Interchange), que es la forma canónica para especificar un modelo.

Pasos para generar código a partir de un modelo

Para generar el código a partir de la definición de un modelo deben seguirse los siguientes pasos:

A. Definición del metamodelo

Un metamodelo puede especificarse con un editor gráfico para metamodelos Ecore, o de cualquiera de las siguientes formas: como un documento XML, como un diagrama de clases UML o como interfaces de Java con Anotaciones. EMF provee asistentes para interpretar ese metamodelo y convertirlo en un modelo EMF, es decir, en una instancia del meta-metamodelo Ecore, que es el metamodelo usado por EMF (figura 4-5).

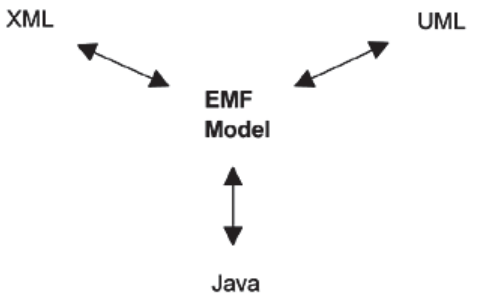

Figura 4-5. Puntos de partida para obtener un modelo EMF

La ventaja de partir de un modelo UML es que es un lenguaje conocido, por lo tanto no se requiere entrenamiento para usarlo, y que las herramientas de UML que suelen ser más amigables y proveen una mayor funcionalidad. Sólo hay que tener en cuenta que la herramienta permita exportar un modelo core que pueda ser usado como entrada para el framework EMF.
La opción más reciente para la especificación de un metamodelo es utilizar el editor gráfico para Ecore. En la figura 4-6 puede verse el metamodelo del lenguaje relacional, hecho con esta herramienta.

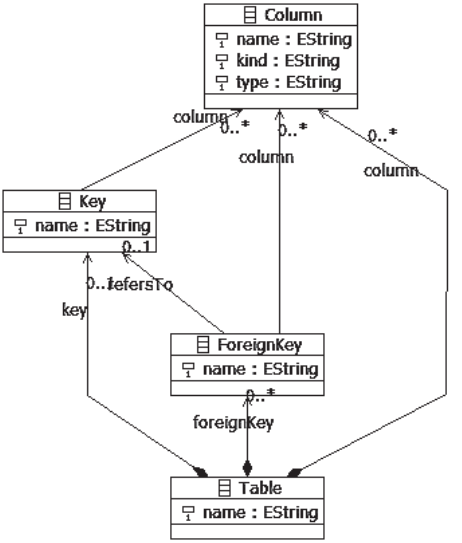

Figura 4-6. Metamodelo del lenguaje relacional

Una caracteristica importante de EMF es que permite crear instancias dinámicas de un modelo desde su editor, sin tener que generar ningun código. El framework reflexivo de EMF permite instanciar los elementos del modelo mediante la opción del menú contextual “Create Dynamic Instance” sobre el elemento a instanciar. La instancia creada se guardará en un archivo XMI dentro del espacio de trabajo.

B. La generación de código

A partir de un modelo representado como instancias de Ecore, EMF puede generar el código para ese modelo. EMF provee un menú contextual con cinco opciones:

- Generate Model Code
- Generate Edit Code
- Generate Editor Code
- Generate Test Code
- Generate All

EMF permite generar cuatro plugins. Con la primer opción se genera un plugin que contiene el código Java de la implementación del modelo, es decir, un conjunto de clases Java que permiten crear instancias de ese modelo, hacer consultas, actualizar, persistir, validar y controlar los cambios producidos en esas instancias. Por cada clase del modelo, se genera dos elementos en Java: una interface y la clase que la implementa. Todas las interfaces generadas extienden directa o indirectamente a la interface EObject. La interface EObject es el equivalente de EMF a Java.lang.Object, es decir, la base de todos los objetos en EMF. EObject y su correspondiente implementación EObjectImpl proveen la base para participar en los mecanismos de notificación y persistencia. Con segunda opción se genera un plugin con las clases necesarias por el editor. Contiene un conjunto de clases adaptadoras que permitiran visualizar y editar propiedades de las instancias en la vista “propiedades” de Eclipse. Estas clases permiten tener una vista estructurada y permiten la edición de los objetos de modelo a través de comandos. Con tercera opción se genera un editor para el modelo. Este plugin define además la implementación de un asistente para la creación de instancias del modelo.

Finalmente, con la cuarta opción se generan casos de prueba, que son esqueletos para verificar propiedades de los elementos. La última opción permite generar todo el código anteriormente mencionado en un solo paso.

En resumen, el código generado es limpio, simple, y eficiente. La idea es que el código que se genera sea lo más parecido posible al que el

usuario hubiera escrito, si lo hubiera hecho a mano. Pero por ser generado, se puede tener la confianza que es correcto. EMF establece un soporte para interoperabilidad con otras herramientas en el framework de Eclipse ya que genera código base para el desarrollo de editores basados en Eclipse y RCP. Como se mencionó, el generador de código de EMF produce archivos que pretenden que sean una combinación entre las partes generadas y las partes modificadas por el programador. Se espera que el usuario edite las clases generadas, para agregar o editar métodos, variables de instancia. Siempre se puede regenerar desde el modelo cuando se necesite, y las partes agregadas serán preservadas durante la regeneración. EMF usa los marcadores @generated en los comentarios Javadoc de las interfaces, clases y métodos generados para identificar las partes generadas. Cualquier método que no tenga ese marcador se mantendrá sin cambios luego de una regeneración de código. Si hay un método en una clase que está en conflicto con un método generado, la versión existente tendrá prioridad.

C. Anatomía del editor básico generado

En la figura 4-7 puede verse el editor generado por EMF. A la derecha se encuentra la vista outline, que muestra el contenido del modelo que se

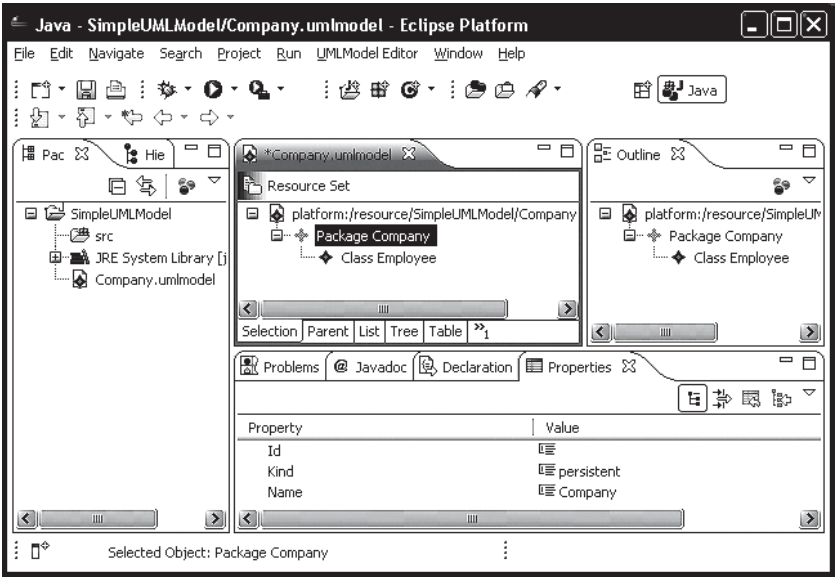

Figura 4-7. Editor generado con EMF

está editando con una vista de árbol. Cuando se selecciona un elemento en el outline se muestra también seleccionado en la primera página del editor, que tiene una forma de árbol.
Independientemente de la vista donde se seleccione el elemento, al seleccionarlo se muestran sus propiedades en la vista de propiedades. La vista de propiedades permite editar los atributos del elemento y las referencias a otros elementos del modelo (permite seleccionar de una lista de posibles). Se pueden arrastrar con el mouse los elementos para moverlos, cortarlos, copiarlos y pegarlos, borrarlos, y estas acciones soportan deshacer y rehacer. Las otras páginas del editor, Parent, List, Tree, Table, TableTree permiten otras visualizaciones de los elementos, como en forma de tabla y en forma de lista.

4.3.2 EMF y OCL

El plugin OCL permite la definición de restricciones sobre el modelo Ecore para luego evaluarlas y determinar si el modelo está correctamente formado. Además provee facilidades para definir los cuerpos de métodos agregados al modelo Ecore y reglas para calcular valores derivados.
OCL es un lenguaje libre de efectos laterales. Esto significa que ninguna expresión OCL puede modificar elementos del modelo. Ni siquiera los objetos temporarios que se crean al resolver la expresión (como strings, colleciones, tuplas, etc).
El uso de OCL dentro de EMF se realiza mediante la inclusión de anotaciones en el metamodelo. En las mismas se indica, además de la implementación, el tipo de regla OCL a definir. Estas anotaciones complementan el sistema de anotaciones de EMF. Este último sólo provee el esqueleto de los métodos en que se validan las restricciones.
Para poder utilizar el mecanismo de validaciones que provee el plugin de OCL es necesario incluir además un conjunto de plantillas JET [JET] para guiar el proceso de generación de código. En general, el código generado no es una implementación en Java de las reglas definidas, sino que contiene una invocación que luego será interpretada por el plugin OCL.
Dentro de un modelo Ecore es posible definir tres tipos de expresiones OCL: invariantes, propiedades derivadas y cuerpos para las operaciones.

Invariantes

Este tipo de expresiones OCL definen restricciones sobre el modelo. Las mismas son almacenadas en los metadatos de EMF. Es decir, no se tra-

ducen a código Java. Esto tiene como ventaja que se permite cambiar la definición de la restricción y volver a evaluar el modelo sin tener que regenerar el código.
Al definir una restricción en el modelo Ecore y generar el código, se crea para ella un método de validación. Dentro de este método se reconoce la restricción y se construye una consulta para chequear si el modelo la satisface o no.

Propiedades derivadas

EMF implementa las propiedades derivadas como atributos los cuales son marcados como transient, es decir, no persistentes y volatile, es decir, que no es necesario asignar espacio para almacenarlos.
Al definir una propiedad derivada en el modelo Ecore y generar el código, se crea para dicha restricción un método parecido al explicado para la validación de restricciones salvo por los siguientes detalles:

- Como se trata de una propiedad derivada, el contexto en el cual se maneja OCL es el atributo estructural y no la clase.
- Al no ser una restricción, se reconoce la sentencia OCL como una consulta pero no se requiere que tenga un valor booleano.
- Finalmente se evalúa la expresión en lugar de chequearse como en el otro caso.

Operaciones

OCL se usa frecuentemente para especificar pre y post condiciones para las operaciones. Una tercera clase de expresiones definidas sobre las operaciones es la expresión body, la cual define el valor de la operación en términos de sus parámetros y las propiedades disponibles dentro del contexto.
Una vez más, al generar el código, se crea el método definido en la clase el cual tiene definida una implementación. El contexto de la expresión OCL es una operación, lo cual asegura que los nombres y tipos de los parámetros son visibles.

Figura 4-8 Expresiones OCL

En la figura 4-8 puede verse la especificación del invariante validName que establece que todos los elementos del modelo tengan nombre asignado. Además puede verse la definición de la operación allAttributes, la cual devuelve todos los atributos de la entidad Classifiers.

4.3.3 Graphical Modeling Framework (GMF)

Hasta ahora se presentaron los plugins referentes a la definicion de la sintaxis abstracta del lenguaje. A continuación se presentan los plugins para la definicion de la sintaxis concreta, tanto gráfica como textual.
GMF [GMF] es un framework de código abierto que permite construir editores gráficos, también desarrollado para el entorno Eclipse. Está basado en los plugins EMF y GEF. Algunos ejemplos de editores generados con GMF son los editores UML, de Ecore, de procesos de negocio y de flujo.

Descripción

Los editores gráficos generados con GMF están completamente integrados a Eclipse y comparten las mismas características con otros edi-

tores, como vista overview, la posibilidad de exportar el diagrama como imagen, cambiar el color y la fuente de los elementos del diagrama, hacer zoom animado del diagrama, imprimirlo. Estas funcionalidades están disponibles desde la barra de herramientas, como en cualquier otro editor.

En la figura 4-9 pueden verse los principales componentes y modelos usados durante el desarrollo basado en GMF. El primer paso es la definición del metamodelo del dominio (archivo con extensión .ecore), donde se define el metamodelo en términos del meta-metamodelo Ecore. A partir de allí, se derivan otros modelos necesarios para la generación del editor: el modelo de definición gráfica específica las figuras que pueden ser dibujadas en el editor y el modelo de definición de tooling contiene información de los elementos en la paleta del editor, los menues, etc.

Una vez definidos estos modelos, habrá que combinar sus elementos. Esto se hace a través de un modelo que los relaciona, es decir, relaciona los elementos del modelo de dominio con representaciones del modelo gráfico y elementos del modelo de tooling. Estas relaciones definen el modelo de mapping (archivo con extensión gmfmap).

Por último, una vez definidos todos los modelos, y relacionados a través del archivo mapping, GMF provee un generador de modelo que permite definir los detalles de implementación anteriores a la fase de generación de código (archivo con extensión gmfgen). La generación de código producirá un plugin editor que interrelaciona la notación con el modelo de dominio. También provee persistencia y sincronización de ambos.

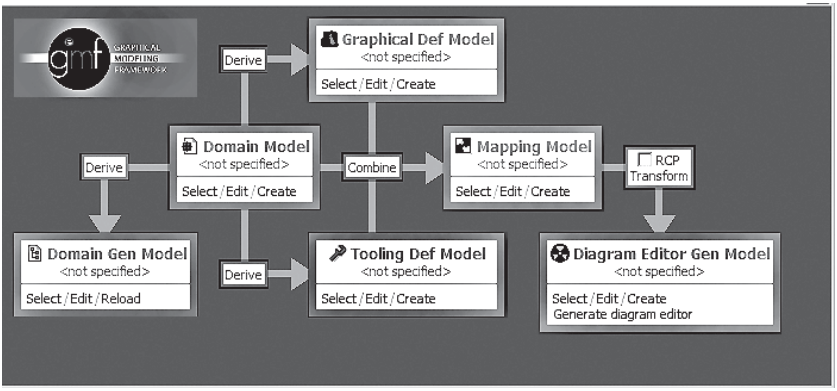

Figura 4-9. Componentes y modelos en GMF

Pasos para definir un editor gráfico

En esta sección se detallan los modelos necesarios para generar el código del editor gráfico. Cada uno de los modelos se define en un archivo separado, con formato XMI. GMF provee un editor para hacer más amigable cada una de estas definiciones.

A. Modelo de dominio
Se debe especificar el metamodelo que se quiere instanciar usando el editor y generar el código necesario para manipularlo usando EMF. No hay necesidad de crear el editor ni los casos de test.

B. Modelo de definición gráfica
El modelo de definición gráfica se usa para definir figuras, nodos, links, compartimientos y demás elementos que se mostrarán en el diagrama. Este modelo está definido en un archivo con extensión gmfgraph. En la figura 4-10 puede verse el editor.

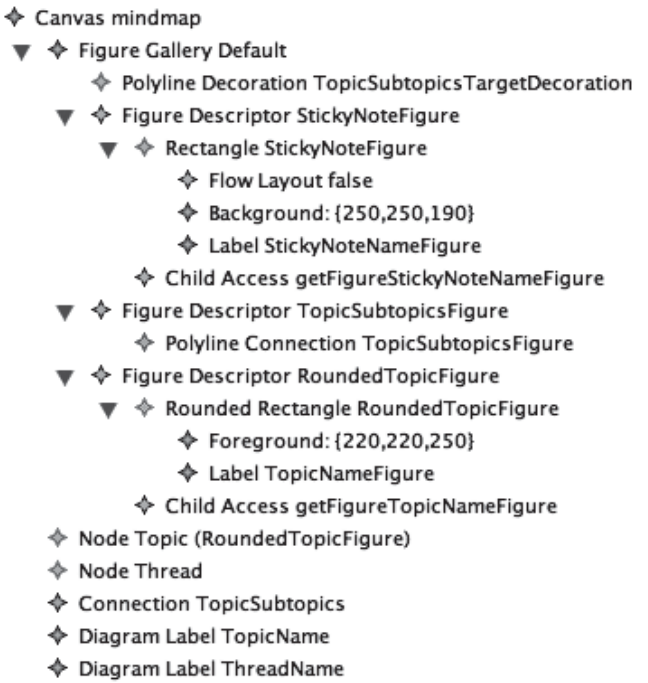

Figura 4-10. Editor del modelo de definición gráfica

C. Definición de herramientas

El modelo de definición de herramienta se usa para especificar la paleta, las herramientas de creación, las acciones que se desencadenan detrás de un elemento de la paleta, las imágenes de los botones, etc. En la figura 4-11 puede verse el editor para la creación de este modelo.

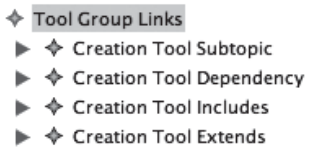

Figura 4-11. Editor del modelo de definición de herramientas

D. Definición de las relaciones entre los elementos

El modelo de definición de relaciones entre elemento afecta a los tres modelos: el modelo de dominio, la definición gráfica y la definición de herramientas. Es un modelo clave para GMF y a partir de este modelo se obtiene el modelo generador, que es el que permite la generación de código.

En la figura 4-12 puede verse el editor para este modelo. En este modelo se especifica que elemento del metamodelo va a ser instanciado con que herramienta o con que entrada de la paleta, y que vista le corresponde en el editor.

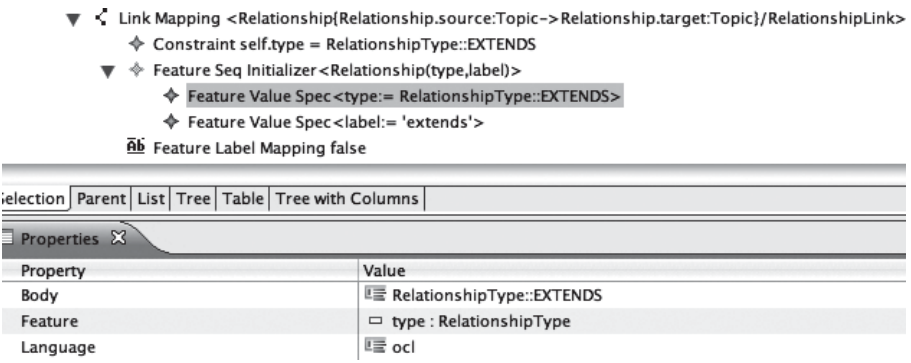

Figura 4-12. Editor del modelo de definición de relaciones

E. Generación de código para el editor gráfico

Ya definidos los elementos gráficos y de mapeo, se puede comenzar a generar el código necesario. Para llevar a cabo la generación de código, primero es necesario crear el modelo generador para poder setear las propiedades propias de la generación de código. Este archivo es muy similar al archivo genmodel definido en EMF. Este paso se hace por medio de un menú contextual, sobre el archivo de mapeo, con la opción 'Create generator model...'

El modelo generador permite configurar los últimos detalles antes de la generación de código, como por ejemplo, las extensiones del diagrama y del dominio, que por defecto es el nombre del metamodelo del lenguaje, o cambiar el nombre del paquete para el plugin que se quiere generar. También permite configurar si el diagrama se guardará o no junto con la información del dominio en un único archivo.

Anatomía del editor gráfico

En la figura 4-13 puede verse el editor gráfico cuyo código es completamente generado por GMF. A la izquierda se encuentra el explorador

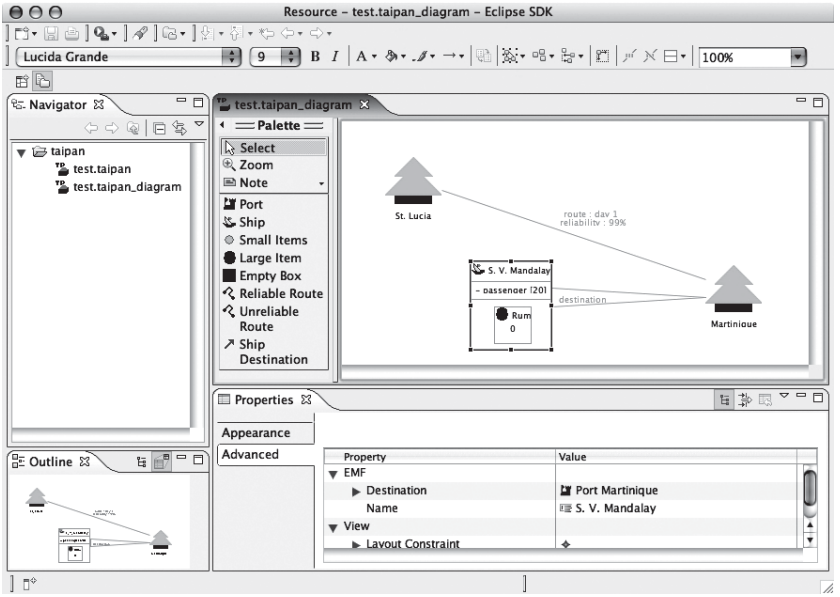

Figura 4-13. Anatomía del Editor gráfico

de paquetes, que muestra los recursos que se encuentran en cada proyecto. Por defecto, cada modelo creado por el editor está formado de dos archivos: un archivo conteniendo las instancias del metamodelo y otro conteniendo la información gráfica. Ambos archivos deben mantenerse sincronizados para tener un modelo consistente. Abajo a la izquierda se encuentra la vista outline, que es una vista con escala reducida que permite ver la totalidad del diagrama. En la parte inferior de la pantalla se ve la vista de propiedades, desde la cual se editan las propiedades de los elementos. La parte central de la pantalla muestra el editor, con su paleta de elementos. Como se dijo, este editor está integrado al entorno Eclipse, ya que muchas de las funcionalidades del editor están disponibles por medio de menús contextuales, y también desde la barra de herramientas de eclipse, como por ejemplo, el zoom, tipos de fuente, etc.

4.3.4 TMF

Además de las facilidades para crear editores visuales, en Eclipse existen plugins que permiten definir lenguajes textuales. A continuación se listan algunos de ellos.
TMF es el acrónimo inglés de Textual Modeling Framework. TMF es un subproyecto de Eclipse que contiene los plugins para definir la sintaxis concreta en forma de texto. A continuación se presentan los plugins de ese subproyecto, Xtext y TCS.

Xtext

Xtext permite desarrollar una gramática de un lenguaje usando una sintaxis similar a BNF. Permite generar un metamodelo basado en Ecore, un editor de texto para Eclipse y su correspondiente parser. La intención de Xtext es comenzar con la gramática y producir el modelo Ecore, en lugar de partir de un modelo existente y derivar una gramática que lo soporte. Sin embargo, permite realizar transformaciones desde y hacia modelos Ecore, con lo cual se maneja interoperabilidad con otras herramientas basadas en EMF, como pueden ser QVT y MOFScript. Y si es necesario, permite en la especificación de la gramática usar un metamodelo existente por medio de un mecanismo de importación.
Provee facilidades para trabajar con el lenguaje generado: genera un editor de texto completamente funcional, con resaltador de sintaxis, asistentes para completar el código, vista de outline, etc.

Actualmente XText es parte de la versión reciente de Eclipse llamada Galileo.

TCS

TCS es el acrónimo inglés de Textual Concrete Syntax. Es un componente de Eclipse que permite especificar una sintaxis concreta en forma de texto agregando información sintáctica a un metamodelo.
TCS provee:

1. Medios para especificar sintaxis textuales para metamodelos.
2. Capacidades de traducción desde sentencias textuales del lenguaje a su equivalente en el modelo y viceversa. Permitir obtener la definición textual para un modelo dado.
3. Un editor de texto enriquecido para Eclipse, con resaltado de sintaxis y ayudas al usuario, como las sugerencias de código, los hiperlinks y la vista de Outline. Todos basados en la especificación sintáctica.

TCL provee medios para asociar elementos sintácticos a elementos del metamodelo con facilidad. Las traducciones de modelo a texto y de texto a modelo se realizan usando una única especificación TCS. Y a partir de esta se genera una gramática que se encarga de realizar ambas traducciones. El metamodelo del lenguaje debe estar especificado en KM3, ya que TCL se basa en ese lenguaje para especificar sus reglas.

4.3.5 Editores textuales y gráficos

Cuando se habla de modelado, existe un conflicto acerca que notación elegir, si la gráfica o la textual. Como habiamos mencionado, ambos formalismos poseen igual poder expresivo. Ambas notaciones tienen sus ventajas. Para los editores en forma de texto, existen herramientas que se pueden usar para comparar, mezclar y versionar los modelos. Por otro lado, las herramientas gráficas tienen la ventaja de tener una visualizacion más amigable. Entonces ¿por qué no combinar las ventajas de ambos mundos? Eclipse permite combinar sus componentes, EMF para la sintaxis abstracta, GMF para la sintaxis concreta en forma gráfica y XText o TCS para la sintaxis concreta en forma de texto. Asi un modelo tendrá las dos vistas, y podrá ser editable desde ambas, lo que permite contar con las ventajas de ambas notaciones.

4.3.6 Otros plugins para la especificación de la sintaxis concreta de un lenguaje

A continuación se presentan dos plugins adicionales, el primero para especificar la sintaxis concreta en forma gráfica, y el segundo para hacerlo en forma textual: EuGENia y EMFText.

EuGENia

Como se vio en la sección anterior, GMF es un plugin para generar editores visuales. Estos editores están basados en tres modelos, el gfmgraph, gmftool y gmfmap, cada uno con su propia sintaxis, lo que requiere un manejo adecuado de los mismos.
EuGENia es un plugin que ayuda a la construcción de esos modelos. A partir de un modelo Ecore se generan automáticamente los modelos de los archivos gmfgraph, gmftool y gmfmap. Para hacerlo, se debe marcar al modelo Ecore con anotaciones especiales definidas por EuGENia, que al reconocerlas generará estos tres modelos de manera automática. Luego, esos modelos generados por EuGENia pueden ser utilizados normalmente por GMF para generar el editor gráfico.
Ejemplos de esas anotaciones son:

- @gmf.diagram: el cual debe marcar el elemento raíz del modelo. A esta anotación se pueden agregar atributos como onefile=true, que especifica que el diagrama estará guardado junto con el modelo en un solo archivo, o model.extension, que indica la extensión del archivo del modelo.
- @gmf.node: que indica que el elemento aparecerá en los diagramas como un nodo. Esta anotación acepta los siguientes detalles:

    - label: debe indicar el nombre del atributo del elemento que aparecerá como label del nodo,
    - figure: indica la figura que representa al elemento. Puede ser un rectángulo, una elipse, un rectángulo con las puntas redondeadas, o la clase Java que implementa la figura.

- @gmf.link: indica que el elemento aparecerá como un link en los diagramas. A esta anotación se le pueden agregar detalles como source, target, sourceDecoration y targetDecoration. Los valores que pueden tomar los dos últimos son none, flecha, rombo.

EMFText

Desarrollado por un equipo de la Universidad Técnica de Dresden, EMFText es una herramienta que permite a los usuarios definir una sintaxis textual para un modelo Ecore. Este plugin genera los componentes para cargar, editar y guardar instancias del modelo. La fuente para la generación es un archivo con extensión .cs, el cual contiene la especificación de la sintaxis concreta del lenguaje. Combinando esta información con el metamodelo escrito en Ecore, EMFText deriva una gramática libre de contexto y la exporta como una especificación de un parser, la cual contiene las acciones semánticas que cubren la instanciación del metamodelo. Dentro de las facilidades que tiene el editor mencionado, se pueden indicar colores para los diferentes elementos, y soporta sugerencias para completar el código, con lo cual el editor propone palabras para utilizar como siguiente elemento.
Aunque la implementación generada usualmente cubre las conversiones necesarias (de texto a modelo y de modelo a texto), se permite que el usuario la modifique para adecuarla a su metamodelo. Y estos cambios permanecerán en las siguientes generaciones, ya que estas clases modificadas no son reemplazadas.

4.4 La propuesta de Metacase (Jyväskylä)

Metacase es un proveedor de ambientes para modelado específico de dominio que tiene una gran trayectoria, ya que desde el año 1991 trabaja con lenguajes de modelado y generadores. Actualmente ofrece una herramienta llamada MetaEdit+ para trabajar con lenguajes específicos de dominio. La versión descripta en este capítulo es la 4.5.
MetaEdit+ [MetaEdit+] es una herramienta comercial, basada en repositorios, que usa una arquitectura cliente/servidor. El ambiente está implementado en VisualWorks. Permite diseñar un lenguaje específico de dominio, para luego construir herramientas de modelado y generadores de código para ese lenguaje. Para la definición de metamodelos, MetaEdit+ usa un lenguaje de meta-metamodelado llamado GOPPRR (Graph-Object-Property-Port-Role-Relationship), que se explica a continuación.

4.4.1 Descripción

La idea general puede verse en la figura 4-14. Con esta herramienta el experto define un lenguaje específico de dominio a través de un

metamodelo, que contiene los conceptos del dominio, y las reglas que deben cumplir. Además del metamodelo, el experto debe especificar su notación gráfica y el código que se quiere generar a partir de las instancias de ese metamodelo. Esta definición se almacena en el repositorio, así el resto del equipo de desarrollo puede usarlo para hacer modelos con el lenguaje definido por el experto y generar automáticamente el código para esos modelos.

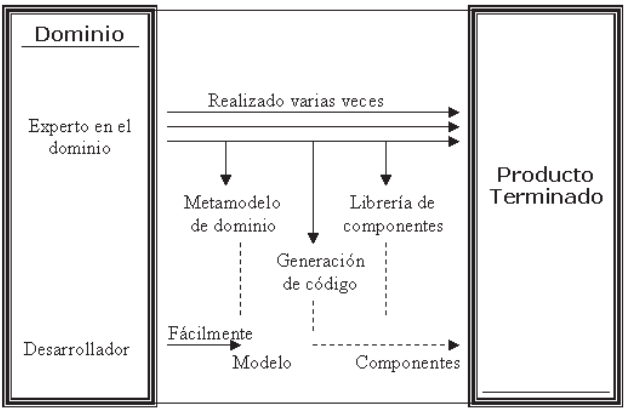

Figura 4-14. Overview de MetaEdit+

La figura 4-15 nos da una vista general de los pasos necesarios para poder definir un nuevo lenguaje, y utilizarlo. Al final de la sección se explican más detalladamente estos pasos.

4.4.2 Arquitectura

La arquitectura de MetaEdit+ puede verse en la figura 4-16. Es una arquitectura cliente/servidor basada en repositorios. Los modelos se pueden acceder o modificar usando los cuatro estilos de representación alternativos: diagramas, matrices, tablas o vistas de árboles. Para cada formato, se proveen distintos editores y un conjunto de browsers.
El repositorio permite a múltiples usuarios acceder y compartir datos concurrentemente, por lo tanto tiene soporte multiusuario / multiobjeto. El ambiente puede ejecutarse en la mayoría de las plataformas, incluyendo UNIX, Windows, NT y Macintosh.

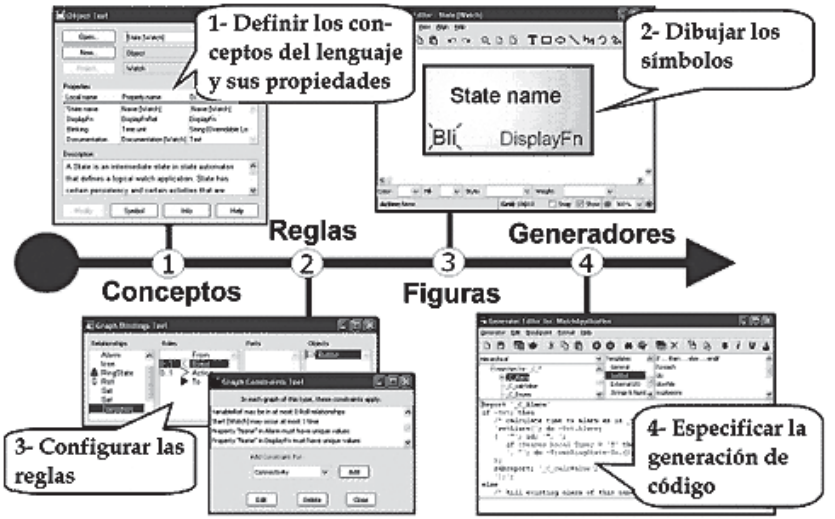

Figura 4-15. Acciones para definir un nuevo lenguaje

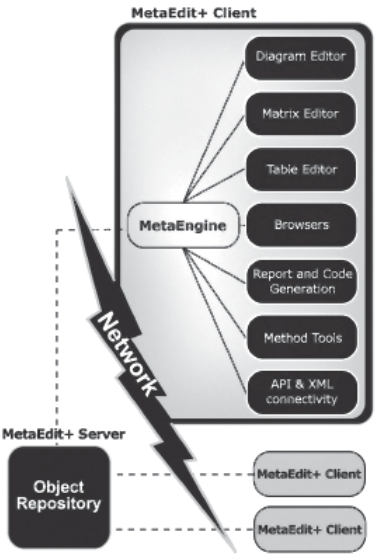

Figura 4-16. Arquitectura de MetaEdit+

4.4.3 Meta-metamodelo

El meta-metamodelo usado por MetaEdit+ es GOPPRR. GOPPRR (se pronuncia gop-ruh) es un acrónimo de Graph-Object-Property-Port-Role-Relationship. Cada uno de esos elementos también se lo llama metatipo. Sus principales elementos son los siguientes:

- Grafo (Graph): un grafo es una colección de objetos, relaciones, roles y bindings mostrando que objetos se relacionan con que roles. Ejemplo de un grafo es un diagrama de clases UML.
- Objeto (Object): un objeto es un elemento que puede ser colocado en un gráfico. Ejemplos de objetos son botones (Button), un estado (State) y una acción (Action). Todas las instancias de objetos pueden ser reutilizadas en otros gráficos utilizando la función Añadir existentes.
- Relación (Relationship): una relación es una conexión explícita entre dos o más objetos. Las. relaciones se conectan a los objetos a través de roles. Un ejemplo es una relación de herencia o una asociación de UML.
- Rol (Role): un rol especifica de qué forma participa un objeto en una relación. Como ejemplo en las relaciones de herencia son los antepasados y los descendientes.
- Puerto (Port): un puerto es una especificación opcional de una parte específica de un objeto al cual se puede conectar un rol. Normalmente, los roles conectan directamente a dos objetos y la semántica de la conexión es provista por el tipo del rol.
- Propiedad (Property): una propiedad es una descripción de una característica asociada con los otros tipos, como nombre, identificador o una descripción.

La figura 4-17 presenta el antecesor de GOPPRR, el meta-metamodelo GOPRR expresado en el mismo lenguaje.

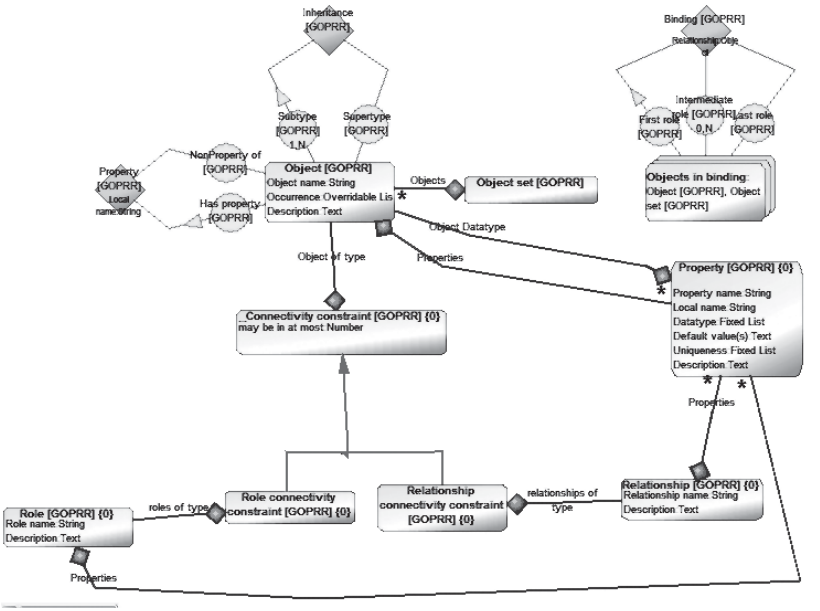

Figura 4-17. Parte del Meta-metamodelo GOPRR de MetaEdit+

4.4.4 Pasos para definir un lenguaje de modelado

Los pasos para definir un lenguaje de modelado usando MetaEdit+ son los siguientes:

A. Definición del metalenguaje o los conceptos del dominio
Un lenguaje de modelado específico de un dominio debe aplicar conceptos que se corresponden exactamente con la semántica del dominio. Usando las herramientas de metamodelado, se debe definir cada concepto, sus propiedades, descripciones y demás características.

B. Definición de reglas del dominio
El lenguaje de modelado específico debe respetar las reglas que existen en el dominio. Una vez definido el lenguaje, se debe garantizar que todos los desarrolladores usen y sigan las mismas reglas de dominio. Estas reglas son de diferentes clases y típicamente están relacionadas a conexiones entre conceptos y re-uso de diseños. Un ejemplo de estas reglas sería que sólo un rol From debe salir de cada objeto Start.

C. Definición de símbolos para la notación

Un lenguaje de modelado visual requiere la definición de símbolos o figuras. La notación debe mostrar tanto como sea posible la visualización natural del correspondiente concepto del dominio. Los usuarios finales son frecuentemente los más indicados para crear esos símbolos. Cada objeto, propiedad, puerto, relación y rol pueden tener su símbolo asociado que será usado como representación gráfica de ese concepto. Los símbolos son editados como vectores gráficos usando un editor de símbolos integrado (figura 4-18). También es posible exportar e importar definiciones.

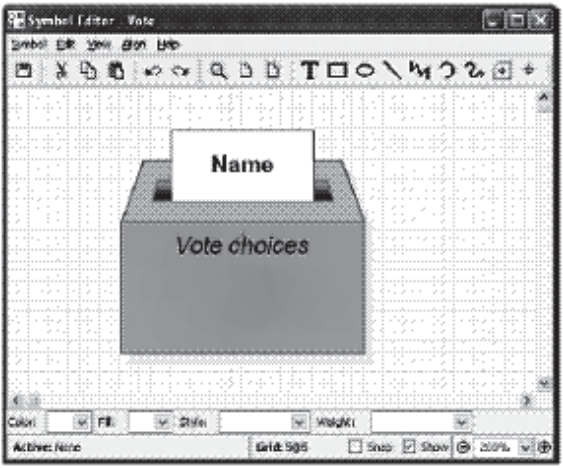

Figura 4-18. Editor de símbolos

D. Configuración del generador de código

Los generadores alejan a los modeladores de los aspectos de implementación, de la arquitectura, del uso de componentes, la optimización y demás. Se pueden configurar para generar el cien por ciento del código requerido en Java. Debido a que quien especifica el generador es el experto, el resultado serán productos con una mejor calidad que los que se hubieran producido mediante un desarrollo tradicional. La figura 4-19 muestra el Editor para configurar un generador de código.

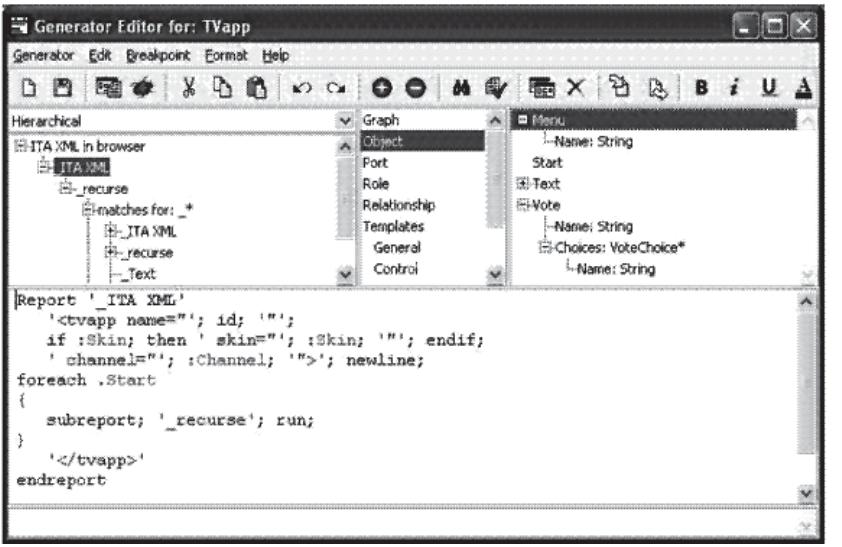

Figura 4-19. Editor del Generador de código

4.4.5 Anatomía del editor gráfico

La figura 4-20 muestra el editor gráfico generado por la herramienta. En la parte superior de la pantalla puede verse la paleta de elementos definidos en el lenguaje. Para crear instancias de estos elementos, se deben arrastrar hasta el editor. Los gráficos en el editor son los definidos con el editor de símbolos.

Una característica importante de esta herramienta es el soporte para la evolución del lenguaje. Es decir, está permitido modificar el lenguaje aún si ya se han creados instancias. Si el cambio está en las metaclases, por ejemplo, si se elimina un metatipo del lenguaje, las instancias existentes de ese metatipo no se eliminarán, para no perder información, pero no se permitirá crear nuevas instancias. Si el cambio está en las vistas, por ejemplo en los iconos definidos para algún elemento, los modelos existentes se actualizarán inmediatamente.

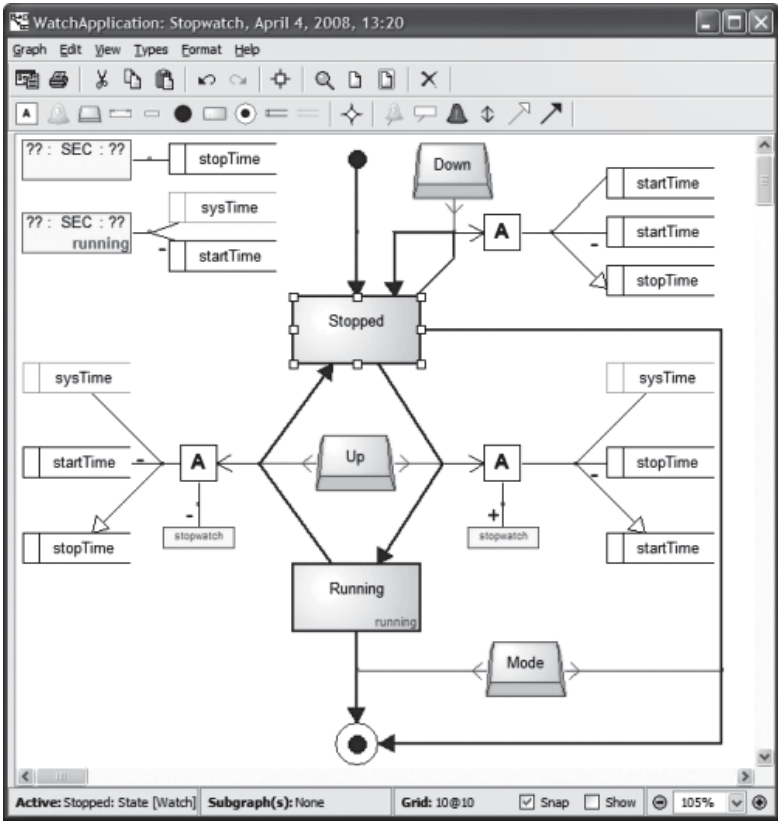

Figura 4-20. Editor gráfico generado

4.5 La propuesta de Microsoft (DSL Tools)

Visual Studio es un producto de Microsoft que contiene herramientas para definir e implementar tanto lenguajes gráficos específicos del dominio (DSLs), como generadores de código asociados. Como se mencionó en el primer capitulo, un DSL es un lenguaje diseñado para realizar tareas específicas para un dominio concreto. Las herramientas provistas en Visual Studio son denominadas colectivamente las “DSL Tools” y utilizan técnicas de desarrollo específicas del dominio para crear e implementar DSLs para Visual Studio.

4.5.1 Descripción

En el ambiente de las DSLs se definen tres roles básicos: el “autor del DSL”, que hace referencia al creador de la DSL, el “usuario del DSL”, el cual hace referencia al usuario que modela soluciones usando la DSL y por último el “usuario de la aplicación” que se refiere al usuario final de una aplicación generada con las DSLs tools.
La figura 4-21 muestra una vista general del proceso. La primera pantalla muestra el diseñador de modelos de dominio, es decir, la vista que utilizan los autores para crear un nuevo DSL. A partir de esta definición, se generará el código para las herramientas que editan y procesan instancias del DSL. La pantalla en el medio muestra la vista del usuario del DSL, que instancia los elementos definidos por el autor del DSL para generar una aplicación final que será utilizada por el usuario de la aplicación.

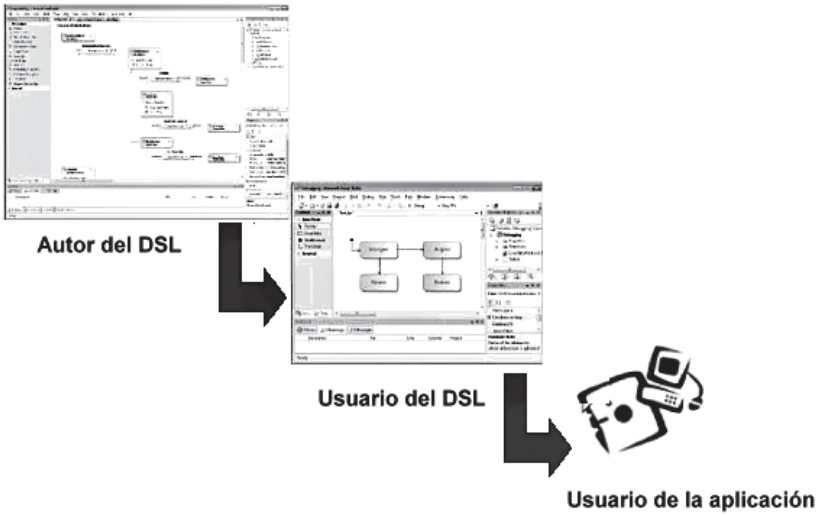

Figura 4-21. Roles definidos en DSL

4.5.2 Meta-metamodelo

El equivalente de un metamodelo en el mundo de Microsoft se llama modelo de dominio. Este modelo de dominio está compuesto por una jerarquía de clases y relaciones. La figura 4-22 muestra un metamodelo simplificado del meta-metamodelo usado para las DSL extraído del trabajo de Jean Bezivin [BHJ+ 05].

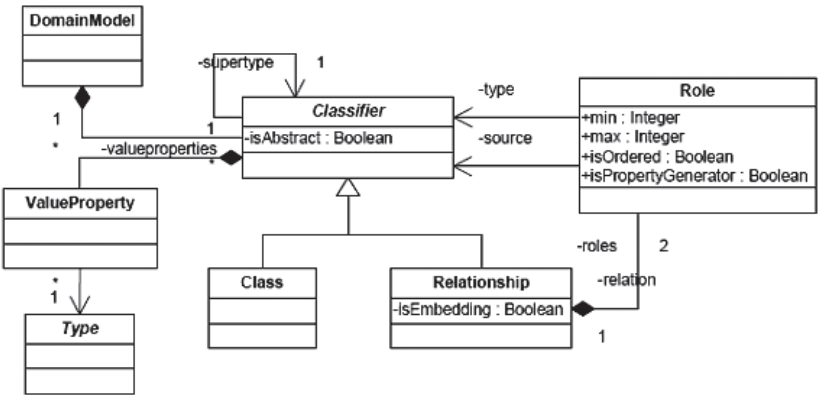

Figura 4-22. Visión simplificada del lenguaje de metamodelado de las DSL Tools expresada en MOF.

4.5.3 Pasos a seguir para definir un lenguaje

Implementar un DSL desde cero representaría una cantidad significativa de trabajo. En las DSL Tools, podemos encontrar un conjunto de frameworks para que sea más fácil crear un DSL dentro de Visual Studio. En la creación de un DSL hay 3 actividades principales:

A- Definir el Modelo de Dominio (Domain Model) constituido por clases y relaciones. Se refiere al metamodelo o sintaxis abstracta del DSL.
B- Definir la notación, es decir las formas (Shapes) que representan a los elementos del dominio (rectángulos, círculos, etc.). Existen 5 diferentes formas: formas geométricas, compartimentos, imágenes, puertos y carriles.
C- Definir la conexión entre los elementos del modelo de dominio con las formas (Shapes).

El proceso de autoría comienza ejecutando el template de proyecto Domain-Specific Designer. Las DSL Tools actualmente ofrecen varios templates, por ejemplo templates de diagramas de Actividades, Casos de Uso y Clases. De este modo podemos elegir alguno de estos templates y extenderlos para obtener nuestro propio Modelo de Dominio con la representación gráfica que deseemos.

Los templates cuentan con un asistente para la creación de DSLs donde se ingresa el nombre del DSL, el espacio de nombres que se utilizará para el código y la extensión que se usará para los archivos que contengan modelos de ese DSL. Al finalizar el asistente se crea una solución DSL conteniendo dos proyectos, Dsl y DslPackage. El proyecto Dsl contiene la definición de la DSL, los editores que se utilizarán y las herramientas de generación de código, y el proyecto DslPackage contendrá la información acerca de cómo se integrará nuestra Dsl dentro de Visual Studio.
En el proyecto Dsl se encuentra el archivo llamado DslDefinition.dsl del cual se derivan todos archivos generados. Para personalizar nuestro DSL, debemos editar el archivo de definición de DSL y luego regenerar el código. Editamos la definición del DSL a través del Designer, que es en sí mismo un diseñador gráfico construido utilizando las DSL Tools.
La figura 4-23 proporciona una vista de la solución; puede verse parte de la definición básica del DSL. A la izquierda se encuentra la definición del modelo de dominio, es decir el template o metamodelo que debemos adaptar para lograr una solución particular y, a la derecha, la definición de los diagramas utilizados en la sintaxis concreta del DSL.

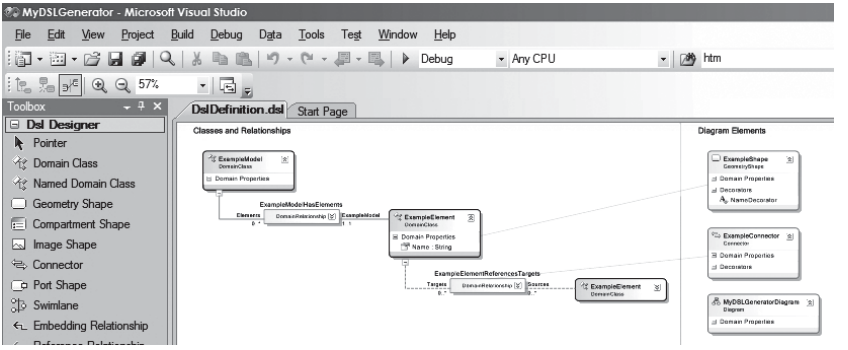

Figura 4-23. Vista de la solución

4.5.4 Generación de código

En cualquier desarrollo que involucre generación de código desde modelos es importante asegurar que sea posible explorar el código y personalizarlo cuando se llegue a los límites de lo que puede ser definido en el modelo. Por esta razón, el equipo DSL Tools tuvo la precaución de diseñar generadores de código de manera que podamos personali-

zar aún más el código resultante, por ejemplo, usando clases parciales. Además, la Definición de DSL puede incluir marcas, las cuales proporcionan a los generadores permisos para insertar código personalizado.

4.5.5 Validaciones y restricciones sobre los modelos

Para expresar las restricciones se puede escribir código que prevenga o impida a los usuarios crear modelos no válidos, distinguiéndose así dos tipos de restricciones dependiendo de la interacción con el usuario:

- Usando el sistema de validaciones (soft-constraints): estas validaciones se realizarán cuando se disparan determinados eventos, que seleccionamos para cada una de estas validaciones (al salvar, al abrirse un documento, al seleccionarlo en el menú emergente cuando hacemos doble-click en el diagrama, o al hacerlo manualmente desde código). Cuando una restricción no se cumple, aparecerá un mensaje en la pestaña de errores, y se permitirá que el usuario siga trabajando con el modelo aunque no se verifiquen todas las restricciones.
- Usando el sistema gráfico en tiempo de ejecución (hard-constraints): estas validaciones se realizan constantemente, en tiempo de ejecución (por ejemplo, cuando se intenta conectar dos Shapes mediante un Connector).

El elegir una forma u otra de restringir nuestro lenguaje depende de las necesidades de cada proyecto. Las soft-constraints, por lo general, permiten trabajar de una forma más rápida, y al final se puede pasar considerar las restricciones, ya que con este sistema de validación podemos seguir trabajando aunque haya partes de nuestro modelo que estén incorrectas.
Las hard-constraints no permiten modelos incorrectos, y las restricciones se solucionan conforme vamos dibujando el diagrama. Estas restricciones deberían ser más sencillas, para que se puedan computar de una forma rápida y no demoren la ejecución del programa. Un excesivo número de hard-constraints también puede llegar a resultar frustrante para el diseñador, por tener que estar deteniéndose a cada paso para solucionar los problemas.
Las validaciones se implementan con los métodos de validación, que pueden definirse sobre cualquier elemento incluyendo las clases del modelo de dominio, las relaciones del dominio, los shapes y los conectores. Se pueden invocar cuando se abre un archivo con el modelo, cuando se guarda, y/o cuando el usuario lo solicita expresamente.

Los mensajes de error y avisos aparecen en la ventana de la lista de errores de Visual Studio. Cada método de validación se define en una clase parcial del modelo de dominio y puede implementar una o más restricciones, mostrando en este último caso varios errores o avisos si estos existiesen al realizar el chequeo (validación). El framework de validación, es el soporte ofrecido para la validación, automáticamente aplica el método a cada instancia de la clase las veces que se requiera.

4.5.6 Anatomía del editor gráfico

La figura 4-24 muestra los principales componentes de un editor de DSLs. A la izquierda se encuentra la barra de herramientas, y a la derecha el explorador de modelos y la vista de propiedades. En la parte central se encuentra el editor gráfico definido.

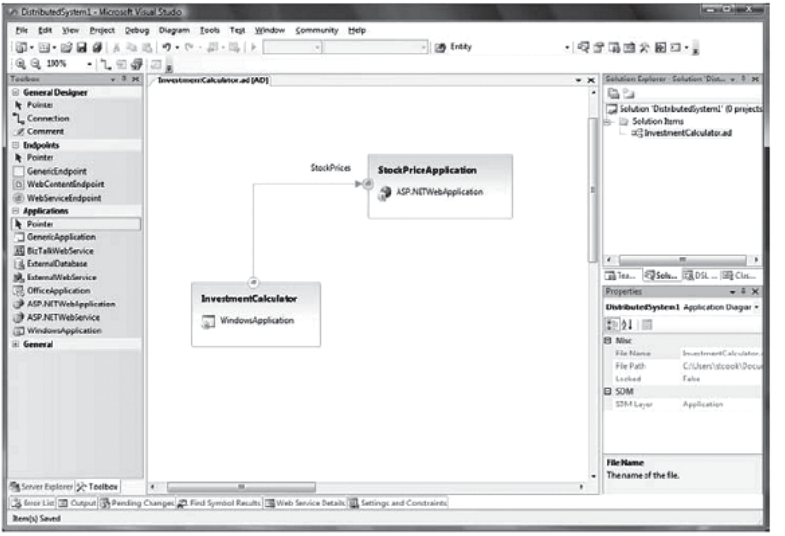

Figura 4-24. Componentes de un Editor DSL en Visual Studio.

4.6 Conclusiones

En este capítulo hemos presentado algunas herramientas que pueden ser usadas en el contexto de un proyecto MDD para definir e instanciar lenguajes de modelado. Hemos distinguido dos tipos de herramientas.

Por un lado, las herramientas que confinan la clase de modelos que se pueden hacer y la manera en que se procesan. Por otro lado, las herramientas que utilizan tecnología basada en metamodelos las cuales permiten al usuario acceder y modificar las especificaciones del lenguaje.
Hemos presentado tres de los principales enfoques: la propuesta de Eclipse, la propuesta de Metacase, y la propuesta de Microsoft. Considerando algunas de las características funcionales deseables en este tipo de herramientas, que también hemos discutido en este capítulo, concluimos resumiendo las fortalezas y debilidades de cada enfoque.
La propuesta de Eclipse presenta las siguientes ventajas: son herramientas de código abierto; la generación de código es posible a partir de la especificación de un modelo; los modelos se especifican usando Ecore, versión simplificada del lenguaje de metamodelado MOF, lo cual establece un soporte para interoperabilidad con otras herramientas; los plugins para la especificación de la sintaxis concreta mantienen integración con EMF facilitando la construcción de editores gráficos y/o textuales; hay una gran disponibilidad de asistentes y tutoriales sobre EMF y GMF.
Como contraparte tiene las siguientes debilidades: hay una gran cantidad de plugins y es difícil elegir cual es mas conveniente para una tarea en particular. Tambien resulta un poco complicado combinar las versiones apropiadas de cada plugin.
La propuesta de Metacase tiene las siguientes fortalezas: cuenta con buena documentación para su utilización, con muchos ejemplos de uso; facilita la propagación de cambios del metamodelo hacia sus instancias; posee un editor integrado para la definición de la sintaxis concreta del lenguaje; el generador de código está integrado en la herramienta.
Y sus puntos débiles son: no es una herramienta de código abierto; los editores generados necesitan de MetaEdit+ para funcionar; se dificulta la portabilidad de proyectos debido a que usa un lenguaje de metamodelado propio que no cumple con el estándar.
La propuesta de Microsoft presenta los siguientes aspectos favorables: soporta plantillas predefinidas para asistir a las distintas actividades durante el proceso de creación del lenguaje; cuenta con buena documentación para su utilización; posee un robusto soporte para especificar y evaluar restricciones sobre los modelos; genera mensajes de errores útiles y comprensibles; cuenta con mapeo visual muy amigable para definir la conexión entre la sintaxis abstracta y la concreta.
Y finalmente, las debilidades de esta propuesta son: no es una herramienta de código abierto; los editores generados necesitan de Visual Studio para funcionar; se dificulta la portabilidad de proyectos debido a que usa un lenguaje de metamodelado propio que no cumple con el estándar.

CAPÍTULO 5

5. Aplicando MDD al Desarrollo de un Sistema de Venta de Libros por Internet

En este capítulo presentamos un ejemplo concreto del proceso MDD. Con el objetivo de mostrar el poder del paradigma MDD, el sistema que desarrollaremos no es trivial, sino un ejemplo tomado de la vida real. Mostraremos como el modelo PIM de ese sistema es automáticamente transformado en un PSM considerablemente complejo y luego veremos como finalmente se llega al código ejecutable. La complejidad del ejemplo es considerable, sin embargo nos enfocaremos sólo en una parte reducida pero suficientemente completa como para apreciar los detalles del proceso. En las siguientes secciones introduciremos los requisitos del sistema que tomamos como ejemplo y describiremos los modelos y transformaciones involucrados en su desarrollo.

5.1 El Sistema de Venta de Libros por Internet

El ejemplo que exploraremos en este libro es el Sistema de Venta de Libros por Internet (On-line Bookstore System), al estilo Amazon, que permite vender y comprar libros a través de Internet. Hemos instalado el ejemplo completo en el sitio http://www.lifia.info.unlp.edu.ar/bookstore. Los requisitos generales del sistema son los siguientes:

1. El bookstore será un sistema basado en la web para la venta de libros.
2. El bookstore tiene libros de los que se conoce el nombre, descripción, los autores, el precio. Los libros están organizados por categorías (category). Un ejemplo para categorías puede ser novelas, cuentos, historia. Un mismo libro puede tener varias versiones, algunas

impresas y otras digitales. De los impresos se conoce el tamaño físico, la presentación, el stock y el peso para el envío. De los digitales su tamaño en bytes y el formato del archivo que lo contiene. La presentación impresa puede ser de tapa dura, edición de lujo, etc.

3. El usuario debe poder agregar libros en un carrito de compras online (cart), previo a realizar la compra. Al momento de finalizar la compra, se generará una orden (order) que incluye los ítems (order item) agregados al carrito de compras. Una vez confirmada la orden, se le enviará un mail al usuario informándole los datos de la misma. Similarmente, el usuario debe poder sacar ítems de su carrito o actualizar las cantidades de un ítem pedido.
4. El usuario debe ser capaz de mantener una lista (wish lists) con los libros que desea comprar más tarde.
5. El usuario debe poder cancelar órdenes antes de que hayan sido enviadas.
6. El usuario debe poder pagar con tarjeta de crédito (credit card) o transferencia bancaria (bank transfer).
7. El usuario podría devolver libros que ha comprado.
8. El usuario debe poder crear una cuenta (client account), de forma tal que el sistema recuerde sus datos (nombre de usuario, nombre y apellido, dirección, email, detalles de su tarjeta de crédito) en el momento de registrarse (login).
   a. El sistema debe mantener una lista con las cuentas de los usuarios en su base de datos central.
   b. Cuando un usuario se registra, su nombre de usuario y la palabra clave (password) debe coincidir con el nombre de usuario y la palabra clave que está almacenada en la base de datos.
   c. Se debe permitir que el usuario modifique su password cuando lo requiera. De la misma manera, se debe permitir que modifique cualquier otro dato de su cuenta.
9. El usuario debe ser capaz de buscar libros usando distintos métodos de búsqueda –título, autor, palabra clave o categoría– y luego podrá ver los detalles (book Spec) de los resultados de su búsqueda.
10. El usuario debe poder escribir una opinión acerca de sus libros favoritos. Las opiniones (review comments) aparecerán junto con los detalles del libro.

5.2 Aplicando MDD

Identificaremos las partes del proceso de desarrollo de software que tienen especial significado dentro del framework MDD. Para comenzar debemos identificar el modelo PIM de nuestro sistema y luego debemos identificar cuáles son los modelos PSM y el código que se espera como resultado del proceso de desarrollo. También es necesario que definamos cuáles son las transformaciones que usaremos para generar el PSM y el código a partir del modelo PIM. Todos estos elementos del framework MDD que aparecerán en nuestro ejemplo son mostrados en la figura 5-1. Los modelos se muestran mediante rectángulos y las transformaciones mediante flechas.

Entonces, necesitamos construir un sistema basado en la web que satisfaga los requisitos planteados. Para este ejemplo, implementaremos el sistema Bookstore usando Java, y aplicaremos el framework Spring, que es un contenedor liviano de J2EE que se ha tornado muy popular. Más específicamente, usaremos una parte del Framework llamada Spring Web MVC. Como su nombre lo sugiere, esta parte de Spring nos permite crear aplicaciones web usando una arquitectura Model-View-Controller (MVC). Para nuestro front-end (la parte "View" del MVC), usaremos JavaServer Pages (JSP). Para el almacenamiento de datos (back-end) conectado a nuestro modelo, usaremos iBATIS. iBATIS está constituido por dos frameworks independientes que generalmente se usan juntos: DAO y sqlMaps. El primero simplifica la implementación del patrón de diseño Data Access Object (DAO) y el segundo simplifica la persistencia de objetos en bases de datos relacionales permitiendo asociar objetos del modelo con sentencias SQL o procedimientos almacenados mediante ficheros descriptores XML. Dado que este desarrollo tiene fines didácticos, usaremos HSQL para la base de datos. HSQL es una base de datos personal, que no resulta adecuada para aplicaciones web multiusuario de gran escala, pero que ciertamente es apropiada para el desarrollo rápido y sencillo de prototipos.

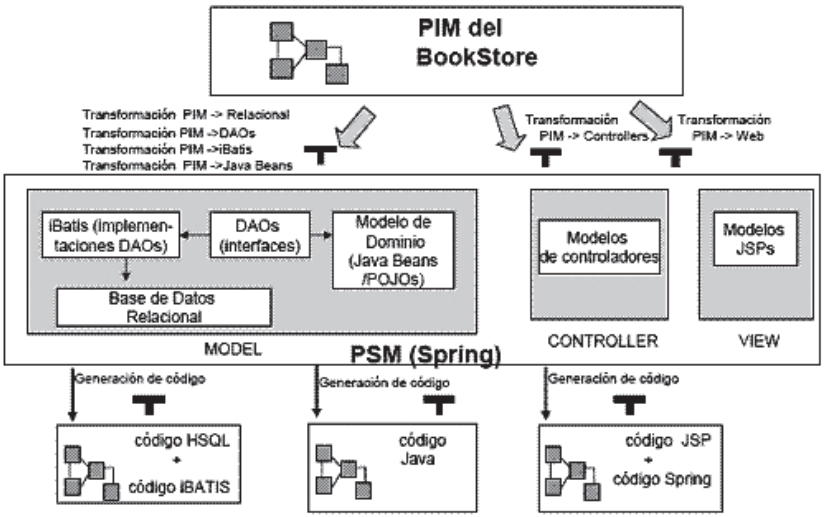

Figura 5-1. Modelos y transformaciones para el sistema de venta de libros.

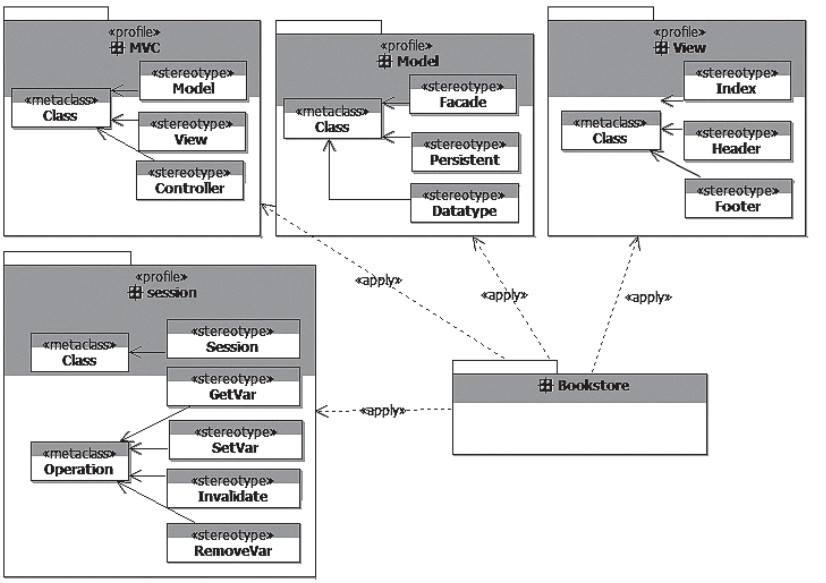

Figura 5-2. Perfiles usados para la construcción del PIM

5.2.1 El PIM y el PSM

El primer paso en el proceso MDD consiste en la construcción de un modelo independiente de la plataforma (PIM) que describa (represente o modele) al sistema de venta de libros. Hemos elegido el lenguaje UML extendido mediante perfiles para expresar nuestro PIM. Se aplicaron los siguientes perfiles: MVC, Model, View y Session. En la figura 5-2 puede verse la definición de cada uno de ellos.
Este modelo PIM puede construirse completamente "a mano" o bien puede derivarse parcialmente desde los requisitos mostrados en la sección anterior. Luego, los modelos PSM y el código son generados automáticamente a partir del PIM. La construcción del modelo PIM inicial es la etapa más "artesanal" del proceso MDD. Si bien existen ciertas heurísticas, tal como las propuestas presentadas en [Larman 04] [KG 06] y [RS 07], la automatización del proceso de creación del PIM es un tema que aún está siendo activamente investigado y no existen herramientas de soporte adecuadas disponibles para su uso.
Dado que hemos decidido adoptar una arquitectura de capas y dado que cada una de estas capas requiere una tecnología diferente, necesitaremos un modelo PSM conformado por distintos sub-modelos, uno para cada capa.

- El primer PSM especifica la base de datos relacional, y lo describiremos usando un modelo relacional.
- El PSM de la siguiente capa representa a los objetos del dominio y a los mecanismos para acceder a los datos. En este PSM estarán presentes conceptos propios de los framework Spring e iBATIS, en particular los conceptos de DAO y sqlMaps. Para crear este PSM nos resulta suficiente utilizar al lenguaje UML con algunos estereotipos.
- El PSM para los controladores muestra el comportamiento del sistema ante los estímulos externos. Aquí aparecen conceptos propios del framework tales como DispatcherServlet, ModelAndView. También usaremos UML con estereotipos para crear este modelo.
- Finalmente, el PSM para la vista Web no usaremos una extensión de UML sino el metamodelo para JSP.

5.2.2 La transformación de PIM a PSM

El PSM tiene una arquitectura Model-View-Controller de tres capas, por lo tanto definiremos transformaciones separadas desde el PIM a cada parte del PSM.

- Una transformación de PIM a la capa Model del MVC. La capa model a su vez está estructurada en distintos niveles, ya que distingue a los objetos del dominio en sí mismos (POJOs), a los mecanismos de acceso a los datos (DAOs) y a los mecanismos de persistencia de los datos (base de datos). Definiremos entonces dos transformaciones, una transformación que toma como entrada un modelo escrito en UML y produce modelos escritos en términos de un modelo relacional y otra transformación que toma el PIM y genera POJOs y DAOs.
- Una transformación de PIM a la capa Vew del MVC. Esta transformación toma como entrada un modelo escrito en UML y produce un modelo escrito en términos del modelo de JSP.
- Una transformación de PIM a la capa Controller del MVC. Esta transformación toma como entrada un modelo escrito en UML y produce un modelo escrito en una extensión de UML específico del framework Spring que describe las reacciones del sistema a los estímulos externos.

5.2.3 La transformación de PSM a Código

El siguiente paso consistirá en generar código ejecutable para cada PSM. Dado que en el framework MDD el código también es considerado un modelo, podemos hablar de modelos de código escritos en algún lenguaje de programación. Para el sistema de venta de libros por Internet tenemos varios modelos de código, escritos en HSQL, XML, Java y JSP. Por lo tanto necesitamos escribir al menos tres transformaciones de PSM a código:

- Una transformación de modelos relacionales a HSQL: una transformación que toma como entrada un modelo relacional y produce el script necesario para crear las tablas en HSQL.
- Una transformación de modelos Java a código Java: una transformación que toma como entrada un modelo escrito en UML con estereotipos para Java y produce un modelo escrito en Java.
- Una transformación de modelos Web a código JSP: una transformación que toma como entrada un modelo escrito en términos del modelo JSP para la web y produce un modelo escrito en JSP.

5.2.4 Tres niveles de abstracción

Todos los modelos en ese ejemplo describen al mismo sistema, pero desde diferentes niveles de abstracción.

- En el nivel de abstracción más alto encontramos al PIM. Este modelo define los conceptos sin incluir ningún detalle específico de la tecnología que se usará para implementar el sistema.
- En el siguiente nivel de abstracción encontramos a los PSMs. Estos modelos, si bien son específicos de sus respectivas plataformas tecnológicas, se abstraen de los detalles concernientes al código ejecutable.
- En el nivel de abstracción más bajo están los modelos de código. Estos modelos son específicos de la plataforma e incluyen todos los detalles tecnológicos necesarios para ejecutarse.

La figura 5-1 muestra los diferentes modelos ubicados en los tres niveles de abstracción y las transformaciones entre ellos. Nótese que las tres capas de la arquitectura del sistema y los tres niveles de abstracción son ortogonales. Los niveles de abstracción se muestran de arriba hacia abajo, mientras que las capas arquitecturales se muestran de derecha a izquierda.
En la siguiente sección describiremos el modelo PIM. Luego nos ocuparemos de definir las transformaciones y las tecnologías necesarias para generar los PSMs y finalmente llegar al código.

5.3 El PIM en detalle

Las figuras 5-3, 5-4 y 5-5a-c muestran el PIM del Sistema de Venta de Libros por Internet. Como mencionamos anteriormente, el PIM es el modelo que requiere mayor intervención humana a través de un proceso creativo. Aquí asumimos que el PIM ya ha sido creado, ya sea mediante un proceso semi-automático basado en la aplicación de heurísticas y patrones o bien utilizando un proceso completamente manual. Los modelos que mostramos en esta sección son el resultado final de dicho proceso.

5.3.1 Modelo de casos de uso

La figura 5-3 presenta el diagrama de casos de uso del sistema. No es el diagrama de casos de uso completo, ya que mostrarlo agregaría una complejidad innecesaria. Nuestro sistema tiene un actor llamado usuario que podrá ingresar al sistema (Login), registrarse (Register), salir del sistema (Logout), ver un listado con las categorías de la librería (View Bookstore Categories), ver los detalles de un libro en particular (View

Book Details), agregar un libro al carrito de compras (Add Item to Cart) y hacer efectiva la compra de los ítems cargados en el carrito (Checkout) entre otras cosas.

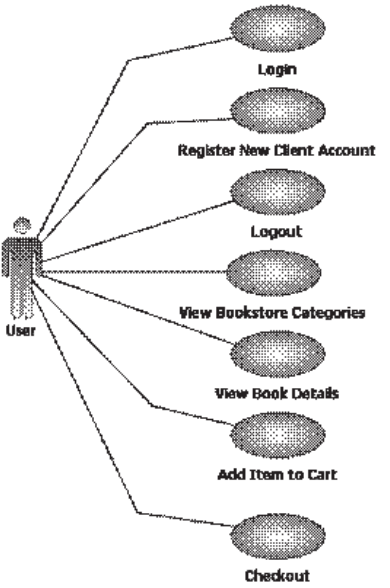

Figura 5-3. PIM del Sistema de Venta de Libros: Modelo de Casos de Uso

Luego, siguiendo la metodología propuesta por Craig Larman en [Larman 04], se escriben las interacciones con el sistema o conversaciones para los casos de uso mencionados.

1- Nombre del caso de uso: Login
Actor: Usuario
Descripción: El usuario de la librería ingresa al sistema
Precondición:
Postcondición: Se considera al usuario como logueado en el sistema.

Descripción extendida:

Acciones del actor	Acciones del sistema
1- El usuario pide ingresar al sistema e ingresa el nombre de usuario y una password	
	2- El sistema verifica que los datos ingresados correspondan a un usuario existente
	3- El sistema recupera los datos del usuario y los carga en la sesión
	4- El sistema muestra la ventana de inicio

2.1 Los datos no corresponden a un usuario existente -> informar el error y pedir el reingreso de datos

2- Nombre del caso de uso: Register New Client Account
Actor: Usuario
Descripción: El usuario de la librería se registra como cliente en el sistema
Precondición: No hay ningún usuario logueado en el sistema
Postcondición: El sistema tiene cliente nuevo.

Descripción extendida:

Acciones del actor	Acciones del sistema
1- El usuario pide registrarse, ingresando un nombre de usuario, password, nombre real, email, teléfono y dirección.	
	2- El sistema valida que los datos pedidos no se dejen incompletos
	3- El sistema crea un nuevo cliente y lo guarda en el sistema
	4- El sistema agrega al cliente a la sesión de usuario considerándolo logueado en el sistema

3.1 El nombre de usuario ya existe en el sistema -> informar el error y pedir que ingrese otro nombre de usuario

3- Nombre del caso de uso: Logout
Actor: Usuario
Descripción: El usuario de la librería pide salir del sistema
Precondición: el usuario está logueado en el sistema
Postcondición: se saca al usuario de la sesión

Descripción extendida:

Acciones del actor	Acciones del sistema
1- El usuario pide salir del sistema.	
	2- El sistema invalida la sesión del usuario
	3- El sistema muestra la ventana de inicio

4- Nombre del caso de uso: View Bookstore Categories
Actor: Usuario
Descripción: El usuario de la librería pide la lista de categorías de la librería
Precondición:
Postcondición:

Descripción extendida:

Acciones del actor	Acciones del sistema
1- El usuario solicita el listado de categorías.	
	2- El sistema recupera las categorías del sistema
	3- Mostrar una ventana con las categorías

5- Nombre del caso de uso: View Book Details
Actor: Usuario
Descripción: El Usuario pide los detalles de una edición de un libro
Precondición:
Postcondición:

Descripción extendida:

Acciones del actor	Acciones del sistema
1- El usuario solicita la información detallada de una edición de un libro indicando el libro y la versión buscada	
	2- El sistema busca la información de la edición del libro solicitado
	3- El sistema lista los datos de esa edición junto con su precio

6- Nombre del caso de uso: Add Item to Cart

Actor: Usuario

Descripción: El usuario de la librería pide agregar un libro al carrito de compras

Precondición:

Postcondición: El carrito de compras contiene un ítem más.

Descripción extendida:

Acciones del actor	Acciones del sistema
1- El usuario pide agregar un libro a su carrito de compras	
	2- El sistema recupera el carrito de compras de la sesión del usuario
	3- El sistema agrega el libro al carrito
	4- El sistema calcula el total a pagar por el usuario
	5- El sistema muestra los libros del carrito junto con el total a pagar

3.1 El libro ya se encuentra en el carrito -> modifica la cantidad pedida de ese libro

7- Nombre del caso de uso: Checkout

Actor: Usuario

Descripción: El usuario de la librería pide concretar la compra

Precondición: el carrito de compras tiene al menos un elemento

Postcondición: Se genera una nueva orden de compra y se eliminan los elementos del carrito de compras.

Descripción extendida:

Acciones del actor	Acciones del sistema
1- El usuario solicita concretar la compra de los elementos contenidos en el carrito	
	2- El sistema recupera el carrito de compras de la sesión de usuario
	3- El sistema crea una orden de compra
	4- El sistema agrega los elementos del carrito a la orden de compra
	5- El sistema muestra la orden de compra, junto con el precio total a pagar

	6- El sistema solicita al Cliente el ingreso de la dirección de envío y los datos de la tarjeta de crédito.
7- El Cliente ingresa la dirección de envío, los datos de su tarjeta de crédito y confirma la compra	
	8- El sistema registra los datos ingresados, y almacena la orden de compra

3.1 Si el usuario no está en el sistema, se pide primero que se registre o que ingrese y luego continúa con el paso 3

5.3.2 Modelo estructural

La figura 5-4 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.
En la parte inferior de las figuras pueden verse definiciones escritas en OCL. Las primeras por ejemplo, son las definiciones de dos operaciones en el contexto del carrito: getNumItems retorna la cantidad de elementos del carrito y getSubTotal retorna el total gastado.

5.3.3 Modelos del comportamiento

Para cada caso de uso se diseña un diagrama de secuencia mostrando la interacción de los elementos del sistema para llevar a cabo la funcionalidad pedida.
Las figuras 5-5a-c muestran los diagramas que especifican el comportamiento del sistema bookstore para los casos de uso Login, Logout y View Book Details presentados anteriormente.

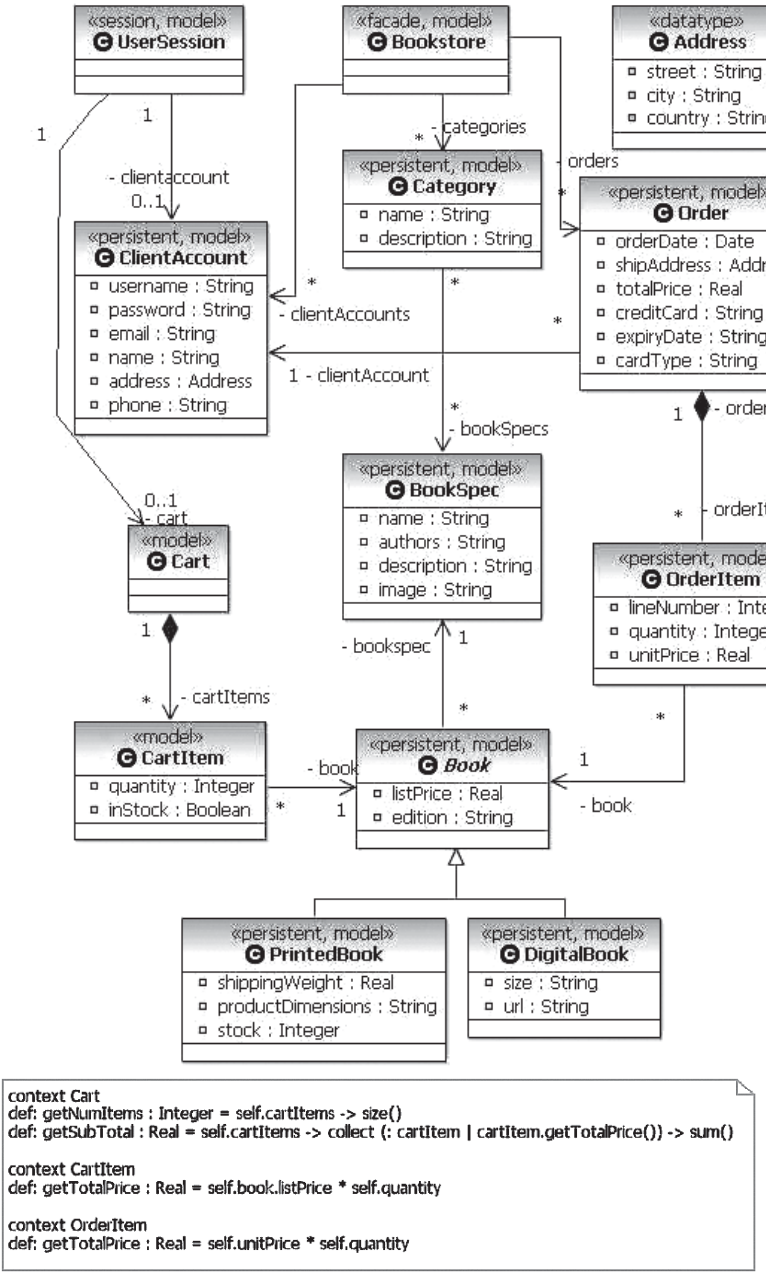

Figura 5-4. PIM del Sistema de Venta de Libros: Modelo Estructural. Contenido del Paquete Bookstore.

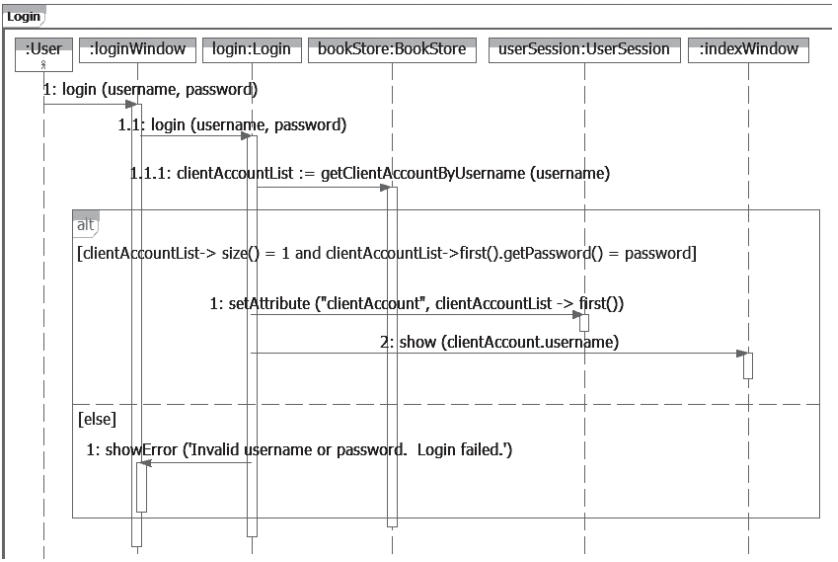

Figura 5-5.a. PIM del Sistema de Venta de Libros: Modelo dinámico para el Caso de Uso Login

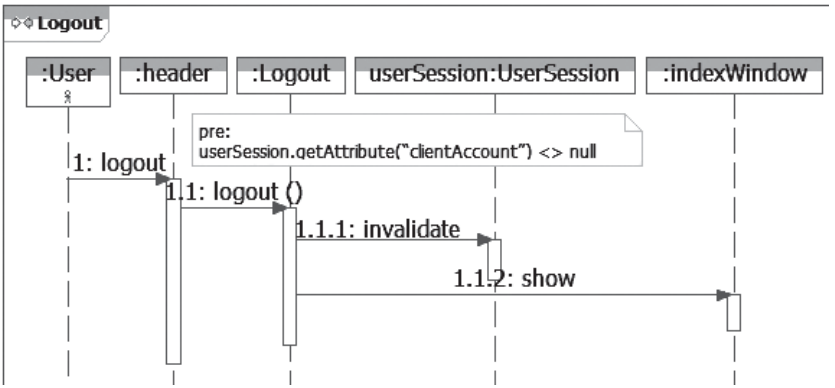

Figura 5-5.b. PIM del Sistema de Venta de Libros: Modelo dinámico para el Caso de Uso Logout

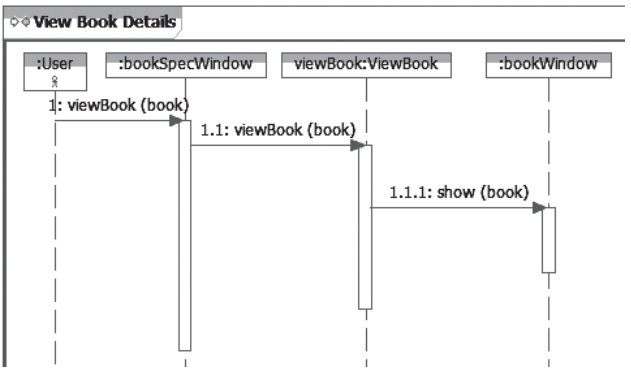

Figura 5-5.c. PIM del Sistema de Venta de Libros: Modelo dinámico para el Caso de Uso View Book Details

La figura 5-6 muestra el diagrama de vistas resultante luego de diseñar los diagramas de secuencia para los casos de uso pedidos. Nuevamente para no complicar con detalles del ejemplo, se muestran las vistas sólo de los casos de uso de los cuales se mostró el modelo dinámico.

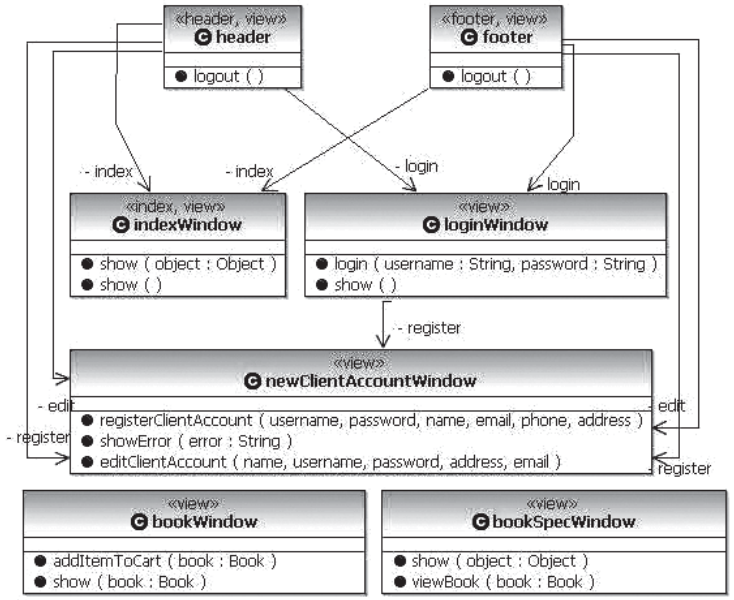

Figura 5-6. PIM del Sistema de Venta de Libros: Modelo de vistas

Como se mencionó anteriormente, además del perfil MVC se utilizó el perfil para vistas. En este perfil se definen tres estereotipos: header, footer e index. Las vistas estereotipadas con header y footer representan el encabezado y el pie de página respectivamente y son comunes a todas las ventanas. La vista estereotipada con index representa la ventana inicial. Las asociaciones entre ventanas representan la posibilidad de mostrar desde la ventana origen de la asociación, la ventana destino.
Por último, la figura 5-7 muestra una parte del modelo de controladores, los involucrados en los diagramas de secuencia. Pueden verse algunas de las clases que cumplirán la función de controladores, es decir, que ejecutarán la lógica de control para responder a las peticiones que el usuario realice sobre las vistas.

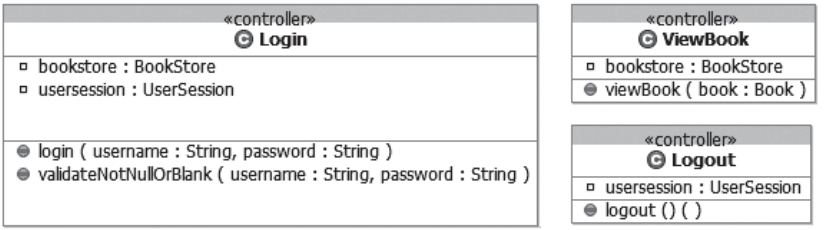

Figura 5-7. PIM del Sistema de Venta de Libros: Modelo de controladores

Los modelos que hemos mostrado en esta sección describen al sistema de venta de libros por Internet de manera independiente de cualquier plataforma tecnológica. Como nuestro objetivo es desarrollar un sistema que corra sobre una plataforma específica, en la siguiente sección explicaremos como transformar estos PIMs en un PSM particular.

5.4 Yendo del PIM al PSM

Esta sección describe informalmente las distintas transformaciones que se realizan sobre el PIM para obtener los PSMs del Sistema de Venta de Libros por Internet.

5.4.1 Transformación del PIM al modelo relacional

Las reglas de transformación para generar el modelo de base de datos relacional a partir del modelo de objetos son bastante evidentes y bien co-

nocidas [Blaha, 1998]. Sin embargo, ejecutarlas manualmente puede tornarse un trabajo complicado ya que pequeños cambios en el PIM pueden acarrear grandes modificaciones sobre el modelo relacional. Por ejemplo, cambiar el tipo de un atributo en el PIM de tipo de dato simple a clase significa introducir una clave foránea en la tabla correspondiente; el tipo de dato simple se torna una columna en la tabla, pero si el tipo de dato es una clase, esta clase se transforma en una tabla completa. La columna deberá contener una referencia (clave foránea) a un valor clave en la otra tabla.

¿Qué reglas debemos usar para generar el modelo relacional? Recordemos que debemos formular reglas que sean aplicables sobre cualquier modelo UML, no sólo sobre el PIM de nuestro Bookstore.

Como primer paso debemos traducir los paquetes de nuestro modelo a esquemas de la base de datos. En nuestro ejemplo el único paquete que contiene a todas las clases del dominio dará origen al esquema Bookstore. Luego debemos ocuparnos de transformar el contenido de dichos paquetes. Con este objetivo, primero debemos decidir como traducir los tipos de datos básicos. Esto es una tarea bastante sencilla; basta con que busquemos tipos de datos correspondientes en el modelo relacional. Entonces, los tipos de datos pueden transformarse de acuerdo a las siguientes reglas:

- Un String de UML se traduce a un VARCHAR[255] del modelo relacional.
- Un Integer de UML se traduce a un INTEGER del modelo relacional.
- Un Date de UML se traduce a un DATE del modelo relacional.
- Un Real de UML se traduce a un DECIMAL(10,2) del modelo relacional.

Adicionalmente, en el PIM tenemos tipos de datos que no son básicos, pero que tampoco son clases. Por ejemplo la dirección (address) de los clientes. Un tipo de dato es una estructura cuya identidad no es relevante y que define uno o más atributos. Tenemos dos opciones: crear una tabla separada para cada tipo de dato, o agregar columnas para el tipo de dato en la tabla que posee el atributo de dicho tipo. Aquí elegimos la segunda opción, ya que simplifica la conexión con el modelo de Java Beans. Entonces, para tipos de datos tenemos la siguiente regla:

- Un tipo de dato UML será transformado en columnas, una por cada uno de sus atributos. El nombre de la columna estará formado por el nombre del rol seguido por el nombre del campo correspondiente del tipo de dato. Por ejemplo el tipo de dato Address genera tres columnas en la tabla ClientAccount, llamadas address_street, address_city y address_country.

En segundo lugar, debemos transformar las clases persistentes a tablas. Aquí hay que distinguir si la clase persistente pertenece a una jerarquía o no. Si la clase no pertenece a una jerarquía debe ser transformada en una tabla, donde todos los atributos son columnas en la tabla (reglas ClassToTable y AttrToCol). Si la clase pertenece a una jerarquía de clases, tenemos tres opciones: transformar a tabla cada una de las clases, o transformar sólo las hojas, o transformar sólo las clases padre. Aquí elegimos la segunda opción ya que asumimos que todas las jerarquías son completas (todo objeto puede considerarse instancia de alguna clase específica). Las tablas correspondientes tendrán que agregar las columnas que representan a los atributos heredados de sus superclases.

Por lo tanto, para las clases tenemos la siguiente regla:

- Una clase UML persistente será transformada en una tabla con el mismo nombre de la clase, y con una columna por cada atributo simple. Por ejemplo la clase ClientAccount da origen a la tabla ClientAccount. Cada tabla tiene una clave primaria con el nombre de la clase concatenado con 'ID', por ejemplo ClientAccountID.
- Para las clases persistentes que forman una jerarquía serán transformadas en tablas sólo las clases hojas de la jerarquía donde se agregarán las columnas correspondientes a los atributos definidos en las superclases. Por ejemplo, la clase abstracta Book no genera ninguna tabla, mientras que sus subclases PrintedBook y DigitalBook dan origen a las tablas PrintedBook y DigitalBook respectivamente.

En el tercer paso las asociaciones en el modelo UML serán transformadas en relaciones de claves foráneas en el modelo de la base de datos, posiblemente introduciendo una tabla nueva. Notemos que existen varias posibilidades para las multiplicidades de una asociación entre las clases A y B en un modelo UML: la multiplicidad en A es cero-o-uno, la multiplicidad en A es uno o la multiplicidad en A es más de uno. Y lo mismo ocurre con la multiplicidad en B. Esto nos deja con nueve combinaciones diferentes de multiplicidades en ambos extremos de la asociación, sin embargo la transformación no es tan compleja y se expresa mediante las siguientes reglas:

- Si la asociación entre A y B tiene una multiplicidad en ambos extremos mayor a uno, entonces se debe crear una tabla representando la asociación, con dos claves foráneas, la primera hace referencia a la tabla que representa a la clase A, y la segunda hace referencia a la tabla que representa a la clase B. Por ejemplo, la asociación entre

Category y BookSpec da origen a la tabla CategoryBookSpec que contiene 2 claves foráneas: CategoryID y BookSpecID.

- Si la asociación entre A y B tiene multiplicidad 1 en el extremo A y multiplicidad * en el extremo B, entonces da origen a una clave foránea en la tabla que representa a la clase B. Esta clave hace referencia a la tabla que representa a la clase A. El nombre de la clave foránea se obtiene agregando el postfijo 'ID' al nombre de rol de la clase A. Por ejemplo la asociación entre Order y OrderItem genera una clave foránea en la tabla OrderItem llamada OrderID.
- Si ambos extremos de la asociación tiene multiplicidad 1, entonces se generará una clave foránea en cada una de las tablas que referencie a la otra tabla. Este caso no se presenta en nuestro modelo.

La figura 5-8 muestra el modelo de base de dato resultante en forma de diagrama relacional. Podemos apreciar el cumplimiento de las reglas de transformación que hemos enumerado más arriba. Las llaves representan las claves primarias mientras que las cruces representan las claves foráneas.

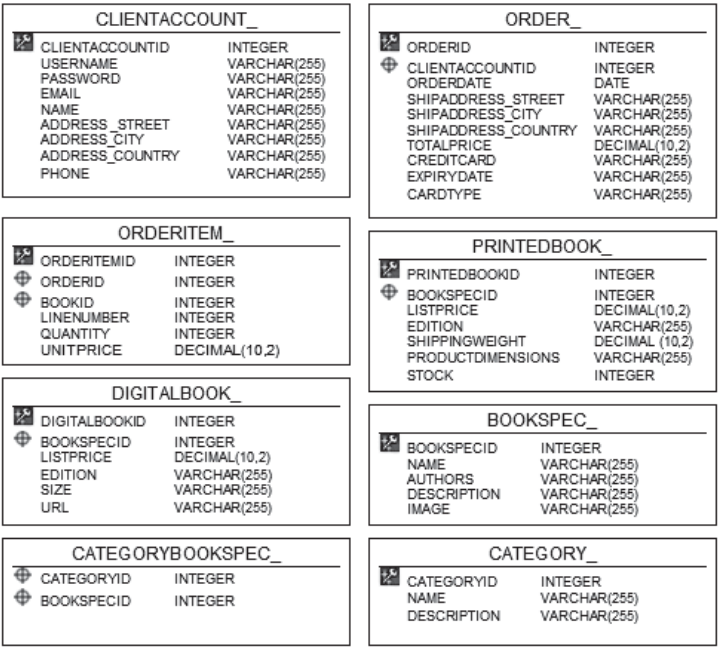

Figura 5-8. PSM relacional del Sistema de Venta de Libros: contenido del Esquema Bookstore

5.4.2 Transformación del PIM al modelo del dominio: Java Beans y DAOs

Como mencionamos anteriormente, hemos decidido utilizar el Framework Spring. El aspecto más destacable de Spring es su uso del patrón de diseño *Inversión de Control* (IoC). En resumen, IoC permite que una clase sea escrita como un JavaBean "puro" con sus propias propiedades, pero sin ninguna dependencia respecto al *framework* en el que está corriendo. Utilizando el patrón de diseño IoC, los objetos (o "beans") no buscan activamente a los datos. En vez de esto, los objetos dependientes son dados al *bean* a través del *framework*. Una de las razones por las que elegimos Spring para el ejemplo del Bookstore en Internet es que permite crear un modelo de objetos persistente usando directamente JavaBeans, con métodos getters y setters simples para cada propiedad.

Estos JavaBeans son nuestras clases del dominio; hay (en general) una correspondencia directa entre éstos y las clases en el modelo del dominio. Spring también proporciona un soporte excelente para DAOs (Data Access Objects). Permite definir DAOs como interfaces simples con métodos para acceder a los objetos utilizando distintos criterios de búsqueda, como por ejemplo el método findBooksByTitle () que nos permitirá acceder a los libros utilizando su título como clave de búsqueda.

En tiempo de ejecución, estos DAOs son mapeados en clases concretas que utilizan la tecnología de persistencia de objetos que uno elija (por ejemplo JDBC, JDO, o Hibernate).

Relacionado a los DAOs, comúnmente se plantea una discusión respecto a si representan objetos simples o colecciones de objetos. Cuando modelamos el dominio, es útil incluir un objeto Collection para representar muchas instancias de la misma clase. Por ejemplo, una clase BookSpec del dominio puede asociarse con una BookSpecCollection. La BookSpecCollection es análoga a una tabla de base de datos, mientras que la BookSpec es análoga a una fila en la tabla. Frecuentemente, esta representación se traslada perfectamente bien al modelo estático detallado. Sin embargo, en otras ocasiones, la analogía podría no ajustarse a la implementación y detalles de diseño. Por ejemplo, con DAOs, una clase DAO es realmente una fuente de colecciones (donde una colección en este caso se deriva desde la interface Java.util.Collection, significando un conjunto de objetos). Es decir el DAO no es una colección sí mismo, sino que es una fuente de colecciones. El DAO retornará diversos objetos Collection (ej., colecciones de BookSpec) de diversos tamaños y conteniendo diferentes subconjuntos de libros dependiendo del método DAO llamado.

Entonces, para completar nuestro sistema Bookstore tenemos que generar un PSM Java Beans. Haremos algunas elecciones arquitectónicas en el modo de usar Java Beans. Estas elecciones son específicas para cada sistema. Según las exigencias del proyecto y las posibilidades ofrecidas por las herramientas disponibles, tendremos que hacer nuestras propias elecciones en cada situación.
El modelo Java Beans para el Bookstore se estructura en una manera bastante similar que el PIM. Por ser un sistema medianamente pequeño, simplemente generaremos un componente (clase Java Beans) por cada clase del PIM y adicionaremos las clases necesarias para representar las interfaces (DAOs) del sistema.
Por cuestiones de legibilidad, podemos agrupar a las reglas de transformación para obtener el modelo PSM Java Beans en dos conjuntos: las reglas para generar las clases del dominio, que definen cómo generar las clases que representan los conceptos del dominio del problema (figura 5-9), y las reglas para generar las clases de acceso a los datos, que especifican como construir las interfaces que facilitan el acceso a los objetos del sistema (figura 5-10). Estas reglas son las siguientes:

Reglas para generar las clases del dominio

Para todas las clases del PIM que representan conceptos del dominio (marcadas con estereotipo <<model>>), excepto la clase con estereotipo <<facade>>, se cumplen las siguientes reglas de transformación:

1. Por cada clase, se genera una clase destino con el mismo nombre. Por ejemplo: La clase Category del PIM da origen a la clase Category en el modelo destino.
2. Cada atributo y cada operación de cada clase, estarán en la clase destino, con los tipos de datos correspondientes.
   Un String de UML se traduce a un String de Java.
   Un Integer de UML se traduce a un int de Java.
   Un Date de UML se traduce a un Date de Java.
   Un Real de UML se traduce a un float de Java.
   Cualquier otro tipo de dato se traduce a una clase con el mismo nombre, los mismos atributos y métodos para acceder a cada uno de esos atributos.
3. En el caso de los atributos, pasan a tener visibilidad “private”, por lo que se genera también un método setter y uno getter por cada uno. Por ejemplo: En la clase Order, el atributo Card Type da origen al atributo Card Type en la clase destino, manteniendo el mismo tipo (String) y con visibilidad privada. Además, se agregan los

métodos setCard Type (Card Type:String) y getCard Type (): String, para poder acceder a dicho atributo.

4. En el caso de las operaciones, se define el correspondiente tipo de retorno, los parámetros con sus tipos (según la regla de pasaje de tipos de datos) y se mantiene el tipo de pasaje. Además, si estuvieran especificados en el PIM, se mantienen el *body*, las precondiciones y las post-condiciones. Si hubiera definidas operaciones adicionales en OCL, se agregan en la clase donde están definidas y se define como body el cuerpo de la definición OCL. Por ejemplo: en la clase Cart se agregan las operaciones getNumItems () y getSubTotal. La primera retorna un int y la segunda un float. En ambas se define como body la definición de la operación adicional.
5. Por cada asociación en el PIM entre estas clases, se genera una asociación en el modelo destino, con los mismos nombres en los finales de asociación.
6. Por cada final de asociación navegable con cardinalidad uno (1), se agrega un método getter y un setter en la clase origen de la asociación, del modelo destino. Por 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.
7. Por cada final de asociación navegable con cardinalidad muchos (*), se agrega un método getter para recuperar la colección en la clase origen de la asociación, del modelo destino. Además, para poder modificarla, se agrega también un método add y un método remove (concatenado con el nombre del final de la asociación). Por ejemplo: En el PIM, la clase Category se asocia con la clase BookSpec con cardinalidad uno a muchos a través del final de asociación *bookSpecs*, que representa la lista de libros de la categoría. Entonces en la clase Category del modelo destino, se genera el método getBookSpecs(): List para poder acceder a dicha lista y los métodos addBookSpec (bookSpec:BookSpec) y removeBookSpec (bookSpec:BookSpec) para poder modificarla.
8. Si la clase es persistente (estereotipo <<persistent>>), se le agrega a la clase destino un atributo identificador de tipo Integer. Por ejemplo: En el PIM, la clase Category está marcada como persistente, entonces la clase destino contendrá el atributo CategoryID de tipo Integer.

Reglas de generación de las interfaces de acceso a los datos (DAOs interfaces)

9. Si la clase es persistente (estereotipo <<persistent>>) y además se cumple que no es parte de otra clase (es decir no participa en una relación de composición como "parte") y no es abstracta (esta condición se agrega para ser mantener consistencia con las decisiones tomadas respecto a las clases abstractas en la transformación de PIM a modelo relacional ya descripta), entonces se genera en el modelo destino una clase DAO con el mismo nombre concatenado con "Dao" y con estereotipo <<interface>>. Por ejemplo: La clase Order da origen a la clase OrderDao, mientras que la clase OrderItem, por ser "parte" en una relación de composición y la clase Book por ser abstracta, no originan clase Dao.

Cada clase Dao generada en el modelo destino contendrá:

Métodos para recuperar y buscar objetos.

10. Un método getter con el identificador como parámetro, que retorna el objeto con dicho identificador, perteneciente a la clase que dio origen al Dao. Por ejemplo: en la clase CategoryDao, el método getCategoryByCategoryID (CategoryID: Integer): Category retorna la categoría correspondiente al CategoryID.
11. Un método getter por cada final de asociación en el PIM. El método tendrá al identificador de la clase de ese final de asociación como parámetro y retorna la lista de objetos, de la clase que dio origen al Dao, asociados con el objeto correspondiente al identificador. Por ejemplo: en el PIM la clase Order está asociada con la clase ClientAccount, entonces en la clase OrderDao se genera el método getOrderListByClientAccountID (clientAccountID: Integer): List que dado un identificador de cuenta de cliente, retorna la lista de órdenes de compra de la cuenta correspondiente a dicho identificador.
12. Un método getter que retorna todos los objetos de la clase que dio origen al Dao. Por ejemplo: en la clase CategoryDao, el método getCategoryList(): List retorna todas las categorías existentes en el Bookstore.
13. Un método getter con una keyword como parámetro, que retorna la lista de objetos de la clase que dio origen al Dao que contengan dicha keyword en cualquiera de sus atributos. Por ejemplo: en la

clase CategoryDao, el método getCategoryListByKeyword (keyword: String): List retorna la lista de categorías que contienen a la keyword en cualquiera de sus atributos.

Métodos para modificar objetos.

14. Un método update para modificar un objeto de la clase que dio origen al Dao. Por ejemplo: en la clase CategoryDao, el método: updateCategory (category: Category) actualiza la información de la categoría dada como parámetro.
15. Un método insert para agregar un objeto de la clase que dio origen al Dao. Por ejemplo: en la clase CategoryDao, el método: insertCategory (category: Category) agrega al Bookstore la categoría dada como parámetro.

Y finalmente, a partir de la clase del PIM con estereotipo <<facade>>, se generan dos clases en el modelo destino:

16. Primero, una clase con el mismo nombre concatenado con "Impl". Esta clase contiene e implementa todos los métodos contenidos en las clases Dao del sistema. Por cada clase DAO, se genera una asociación navegable desde la clase "Impl" hasta la clase DAO. La clase "Impl" contendrá además, por cada asociación con una clase Dao, un método setter y uno getter con el nombre de la clase destino concatenado con "Dao". Por ejemplo: en la clase BookstoreImpl, por estar asociada a la clase ClientAccount Dao, se generan los métodos: getClientAccount Dao():Client AccountDao y setClientAccountDao (clientAccount Dao: Client AccountDao)
17. Y se genera también una clase destino con el mismo nombre concatenado con "Facade" y con estereotipo <<interface>>. Esta interface contendrá todos los métodos contenidos en las clases Dao del sistema, que son implementados por la mencionada clase "Impl" (a través de una relación de dependencia).

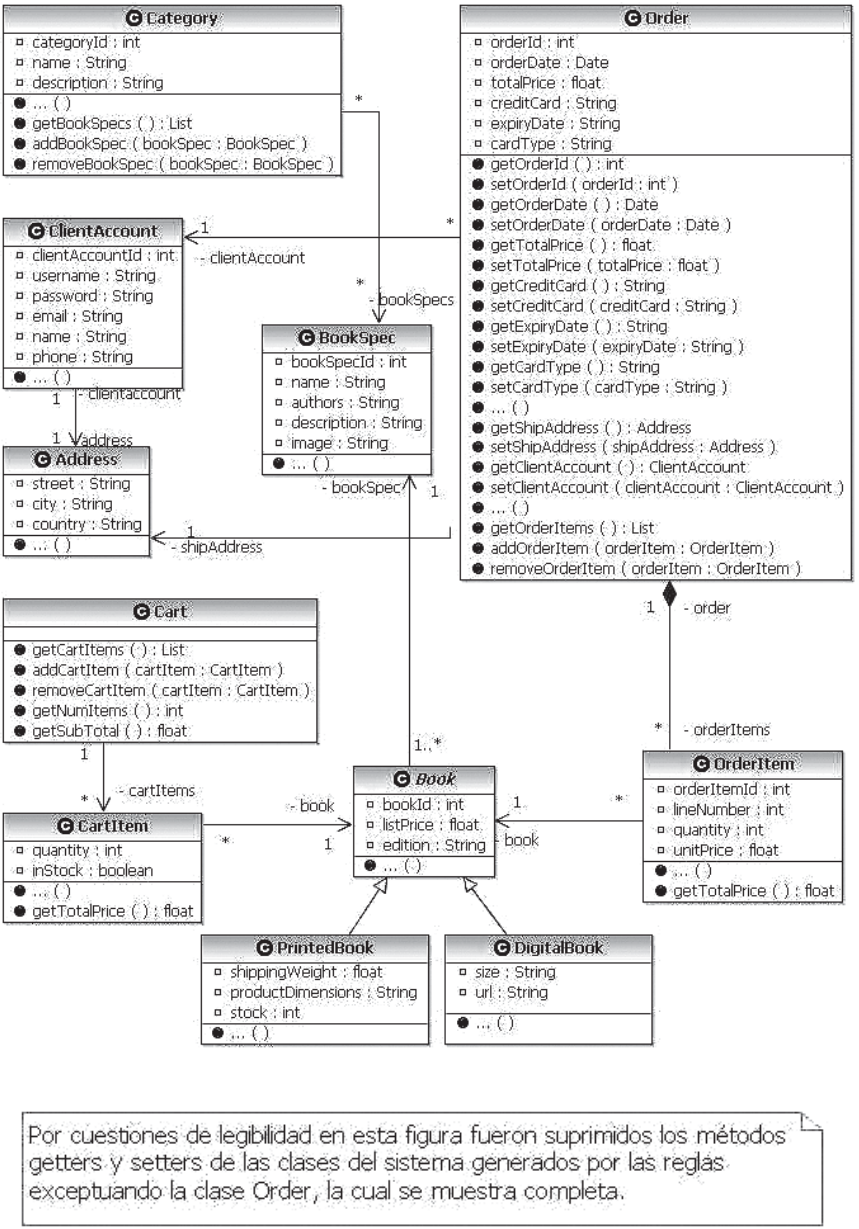

Figura 5-9. PSM Java: entidades del dominio.

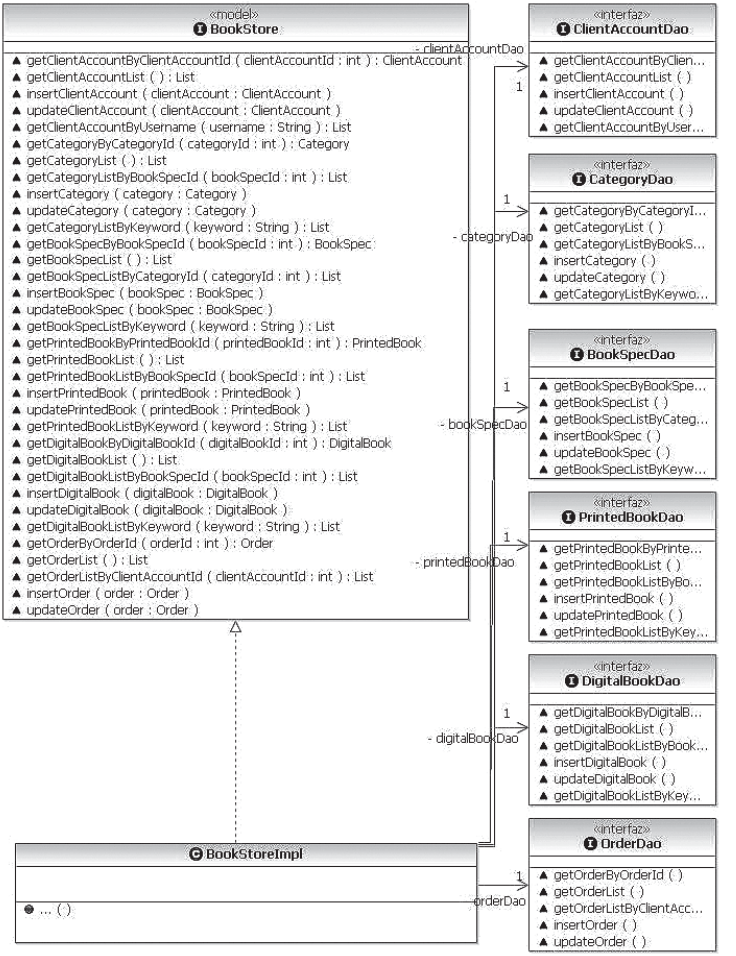

Figura 5-10. PSM Java: interfaces de acceso a los datos.

5.4.3 Creando el puente entre el modelo relacional y el modelo del dominio

iBATIS es un framework de código abierto basado en capas que se ocupa de la capa de persistencia (se sitúa entre la capa de negocio y la capa de acceso a datos). Proporciona un modo simple y flexible de mover los

datos entre los objetos Java y la base de datos relacional, asociando los objetos de modelo (JavaBeans) con sentencias SQL o procedimientos almacenados mediante ficheros descriptores XML, simplificando la utilización de la base de datos. Específicamente, dicha conexión se materializará a través de la creación de implementaciones para cada una de las interfaces DAO definidas en el PSM, tal como queda expresado en la siguiente regla de transformación:

18. Por cada clase persistente y concreta del PIM, se genera una clase en el modelo destino que extiende a la clase del framework iBATIS llamada SQLMapClientDaoSupport, y que implementa la interfaz DAO correspondiente (es decir, implementa todos los métodos definidos en la interfaz). Por ejemplo, para la clase Category se genera otra clase en el PSM con nombre SqlMapCategoryDao, que extiende a la clase del framework SqlMapClientDaoSupport y que implementa a la interfaz CategoryDAO generada en el modelo de dominio (Java Beans).

Las figuras 5-11 y 5-12 exhiben el modelo resultante de aplicar las reglas. La primera muestra las extensiones del framework, mientras que la segunda ilustra las clases que implementan a las interfaces del PSM.

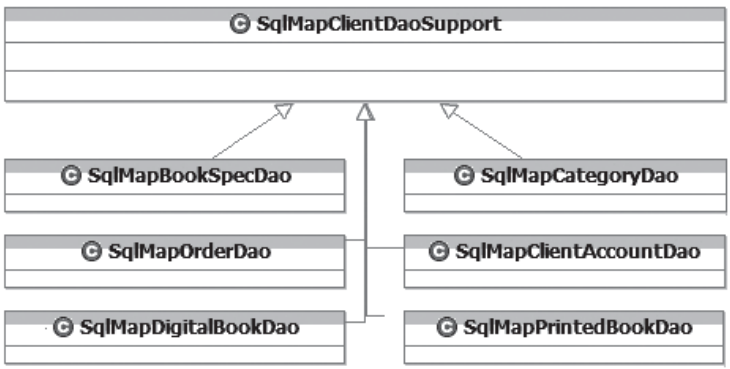

Figura 5-11. PSM Java: puente entre el modelo relacional y el modelo de dominio. Jerarquía de clases.

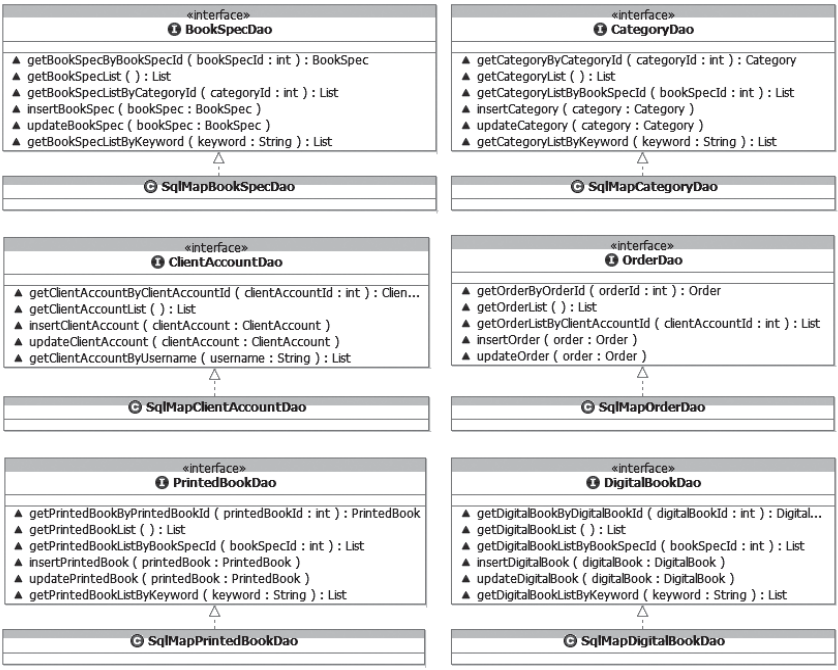

Figura 5-12. PSM Java: puente entre el modelo relacional y el modelo de dominio. Interfaces.

5.4.4 Transformación del PIM al modelo de los controladores

Spring define varios tipos de controladores, cada uno especializado en una tarea en particular. Para poder crear un controlador es necesario implementar el Controlador deseado (definido como una interface) e implementar los métodos necesarios para procesar una petición, en el caso más simple. En otros casos es necesario extender la jerarquía de controladores provista y redefinir unos pocos métodos.
Todos estos tipos de controladores trabajan de manera similar: todos retornan un objeto tipo ModelAndView, es decir, modelo y vista, que como el nombre lo indica, retorna el modelo resultante de ejecutar las tareas requeridas que será desplegado en la vista especificada.
El objeto ModelAndView contiene el nombre lógico de la vista, y el *framework* se encargará de convertir ese nombre lógico al nombre físico que es el que buscará el servidor web.

Además de los controladores, para que la aplicación funcione son necesarios varios archivos de configuración, como el archivo web.xml, applicationContext.xml, etc., los cuales también pueden derivarse automáticamente a partir de los modelos PIM.

Reglas de generación para el modelo de controladores

Para todas las clases del PIM que representan conceptos de los controladores (marcadas con estereotipo <<controller>>), se cumplen las siguientes reglas de transformación:

1. Cada clase genera otra clase Java con el mismo nombre a la que se le agrega el sufijo "Controller".
   Por ejemplo: La clase Login del PIM da origen a la clase Login Controller en el modelo destino.

El tipo de controlador que se necesita para satisfacer una petición depende de la petición. Existen tres casos básicos que dan lugar a las siguientes reglas:

2. Si el usuario no interactúa para completar la tarea requerida, es decir, sólo solicita información, el controlador será un controlador simple. En este caso, la clase del PIM genera una clase en el modelo destino que implementa la interface "Controller" definida en el paquete org.springframework.web.servlet.mvc.
   Por ejemplo, la clase Logout genera el controlador LogoutController que implementa la interface Controller.

3. Si es necesaria la interacción con el usuario, como por ejemplo, la entrada de datos, el controlador debe hacerse cargo de las tareas de desplegar el formulario, validación de los campos ingresados y el envío de los datos desde el formulario. En este caso, la clase del PIM genera una clase en el modelo destino que extiende de la clase SimpleFormController definida en el paquete org.springframework.web.servlet.mvc.
   Por ejemplo: La clase Login del PIM da origen a la clase Login Controller que extiende a la clase org.springframework.web.servlet.mvc.SimpleFormController

4. Por último, hay que analizar el caso donde la aplicación necesita una interacción del usuario que involucra más de una pantalla.

Esta secuencia de pantallas se llama asistente o wizard en inglés. Aquí, el controlador debe hacerse cargo de las tareas de desplegar las pantallas, validar los campos de las mismas, mantener los datos ingresados por el usuario en cada una de las pantallas, y del envío de datos. En este caso la clase del PIM genera una clase en el modelo destino que extiende la clase AbstractWizard FormController definida en el paquete org.springframework. web.servlet.mvc

5. Por lo tanto, el tipo de controlador depende fuertemente del tipo de petición requerida. Además, cuando la aplicación necesita una interacción con el usuario que implica la entrada de datos, es necesario una validación de esos datos, que puede ser simplemente validar que los campos no sean nulos o alguna validación más precisa. Para esto se considera un método especial utilizado en el PIM llamado validateNotNullOrBlank que valida precisamente que los campos contengan alguna información. La existencia de este método en el PIM genera una clase llamada igual que la clase que contiene el método a la que se le agrega el sufijo "Validator", que implementa la interface Validator definida en Spring, la cual contendrá un método validate donde se harán las validaciones requeridas.
   Por ejemplo: la clase Login del PIM posee un método llamado validateNotNullOrBlank. Esto genera una clase asociada a la clase LoginController llamada LoginValidator con un método validate que validará que los parámetros correspondientes no sean blancos ni valores nulos.

6. Por último, las peticiones que involucran la entrada de datos necesitan una clase encargada de mantener los datos ingresados por el usuario. Estas clases son formularios asociados al controlador, que poseen un atributo por cada parámetro ingresado, y define para cada parámetro los métodos de acceso.
   Por ejemplo: La clase Login de modelo de controladores recibe la petición login (username, password) con datos ingresados por el usuario. Por lo tanto se genera, una clase LoginForm dentro del paquete forms, que tiene como atributos los strings username y password. Además, posee los métodos de acceso a esos atributos.

7. En el PIM hay una clase estereotipada session, con sus métodos también estereotipados. Esta clase se corresponde con una existente para aplicaciones web: Javax.servlet.http.HttpSession, ac-

cesible por todos los controles a través del request. Por lo tanto, los mensajes destinados la clase estereotipada session en el modelo origen generará dos mensajes en el modelo destino, el primero es una petición al request para pedir la session y el segundo es el correspondiente mensaje a ese objeto session.
Por ejemplo: la clase Logout en su método logout tenía un mensaje a la clase UserSession. Por lo tanto, se genera en el modelo destino el mensaje request.getSession() y luego otro mensaje con el nombre del mensaje originario a la session.

En este modelo se debe mantener la interacción entre los objetos definida por los diagramas de secuencia. Así, por cada diagrama de secuencia en el PIM se generará un diagrama de secuencia en el PSM, dependiendo del tipo de controlador indicado por las reglas 2, 3 y 4, de la siguiente forma:

8. Si el controlador es un controlador simple, se define un método llamado handleRequest(HttpServletRequest request, HttpServlet Response response).
   Por ejemplo, para el método logout de la clase Logout se generará un método en la clase LogoutController especificado por la figura 5.14.b

9. Si el controlador es un formulario simple, se define un método llamado onSubmit(HttpServletRequest request, HttpServletResponse response,Object command, BindException errors). Por ejemplo, para el método login (username, password) de la clase Login se generará un método en la clase LoginController especificado por la figura 5.14.a. Nótese que este controlador trabaja junto con el formulario LoginForm para llevar a cabo su tarea. Este formulario es el encargado de mantener los datos ingresados por el usuario. Así, desde el método del controlador se le piden estos parámetros al formulario.

10. Todos los mensajes show a las páginas se transforman en un mensaje de creación a la clase ModelAndView con el nombre de la página como parámetro. Si el mensaje tiene parámetros que no están cargados en la sesión, se le pasan estos parámetros a la instancia de ModelAndView, detallando el nombre del parámetro y el contenido, luego del nombre de la página antes mencionado. Algo parecido pasa con los mensajes de error. Se transforman en mensajes de creación a la clase ModelAndView

donde se le pasa el mensaje de error. Por ejemplo, en el diagrama de secuencia del mensaje login de la clase Login, se encontraba el mensaje show(clientAccount.username) a la clase index. Esto genera que en el diagrama de secuencia del mensaje onSubmit de la clase LoginController exista un mensaje de creación a la clase ModelAndView donde se le especifica el nombre de la página, en este caso index.

Las figuras 5-13, 5-14a, 5-14b y 5-14c muestran el PSM resultante de la aplicación de las reglas descritas.

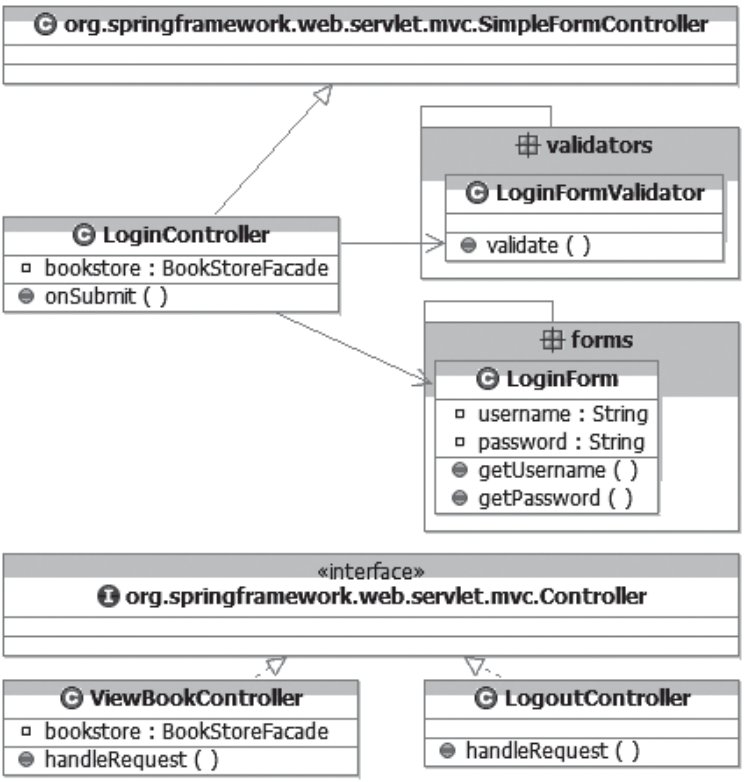

Figura 5-13. PSM Controladores: Los controladores y el framework Spring

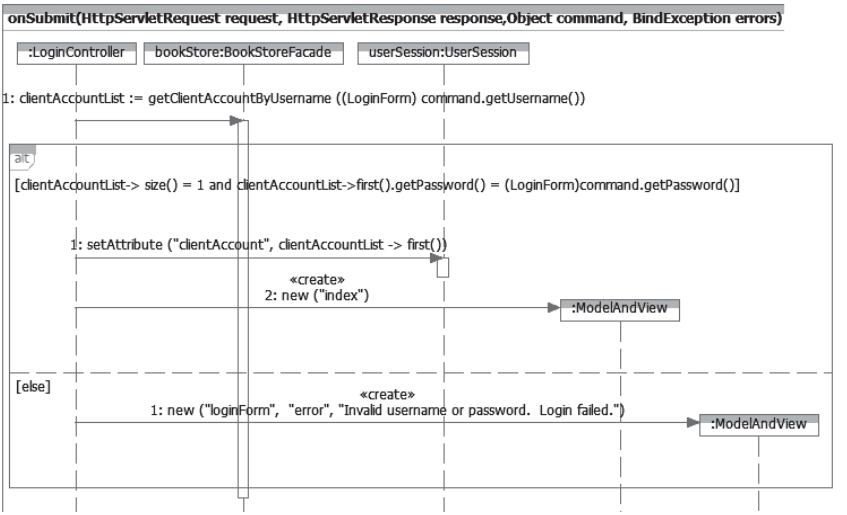

Figura 5-14.a. PSM Controladores: Diagrama de secuencia para la operación login

Figura 5-14.b PSM Controladores: Diagrama de secuencia para la operación logout

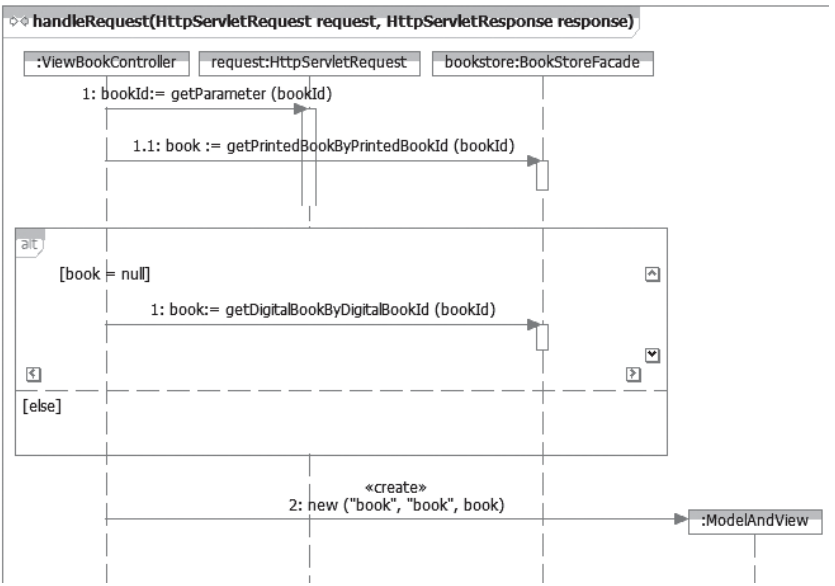

Figura 5-14.c PSM Controladores: Diagrama de secuencia para la operación View Book Details

5.4.5 Transformación del PIM al modelo de las vistas

Java Server Pages (JSP) es una tecnología Java que permite generar contenido dinámico para web, en forma de documentos HTML, XML o de otro tipo. Una de las ventajas de utilizar JSP es que se hereda la portabilidad de Java, y es posible ejecutar las aplicaciones en múltiples plataformas sin cambios. Es común incluso que los desarrolladores trabajen en una plataforma y que la aplicación termine siendo ejecutada en otra.
Una página JSP puede procesar formularios Web, acceder a bases de datos y redireccionar a otras páginas. Las páginas JSP son transformadas a un servlet y después compiladas.

Una página JSP es básicamente una página Web con HTML tradicional y código Java. La extensión de fichero de una página JSP es ".jsp" en vez de ".html" o ".htm", y eso le indica al servidor que esta página requiere un tratamiento especial que se conseguirá con una extensión del servidor o un plug-in. Entonces para ejecutar las páginas JSP, se necesita un servidor Web con un contenedor Web que cumpla con las especifica-

ciones de JSP y de Servlet. Así, cuando un cliente solicita una página JSP, se ejecuta en el servidor el código JSP de la página, dando como resultado una página HTML que se fusiona con el HTML original, generando una página HTML de respuesta que será enviada al cliente.
Las páginas JSP incluyen ciertas variables privilegiadas sin necesidad de declararlas ni configurarlas, como por ejemplo request, response, session.

Reglas de generación para el modelo de vistas

Para todas las clases del PIM que representan conceptos de las vistas (marcadas con estereotipo <<view>>), se cumplen las siguientes reglas de transformación:

1- Por cada clase se genera una página JSP que tiene como nombre el nombre de la clase a la que se le saca el sufijo “Window” si lo tiene, y se le agrega el sufijo “.jsp”
Por ejemplo: La clase indexWindow del PIM da origen a la página index.jsp en el modelo destino.

2- Existen dos clases en el modelo de vistas que son las estereotipadas con “header” y con “footer” que representan el encabezado y el pie de página. Para todas las demás clases se genera una directiva JSP para incluir estas dos páginas al principio y al final respectivamente.
Por ejemplo: en la página index.jsp se generará una directiva para incluir el header al principio de la página. Esto se realiza por medio de una directiva JSP con un atributo con nombre file con valor simple con el nombre de la página correspondiente a la clase estereotipada <<header>>. Análogamente pasará lo mismo para la clase estereotipada como footer.

3- Las asociaciones entre las vistas representan formas de navegar entre ellas. Como consecuencia, cada asociación navegable entre vistas en el modelo origen genera un link navegable entre las páginas correspondientes.
Por ejemplo: La asociación entre header y loginWindow da origen a un link en la página header.jsp que tiene como destino la página login.jsp.

4- El objetivo de las vistas es mostrar información al usuario. Para poder especificar esa información se considero un método espe-

cial “show (parameter [0..*])” que imprime en pantalla lo indicado por los parámetros del mensaje. Por lo tanto, por cada parámetro se genera dentro de la página JSP correspondiente los elementos JSP necesarios para mostrar la información requerida.
Por ejemplo: El mensaje show (clientAccount.username) de la clase indexWindow da origen a este código en la página index.jsp necesario para mostrar el valor de la variable username de clientAccount en pantalla.

Para el resto de los mensajes de las clases estereotipadas con <<view>> se tiene en cuenta si tienen o no parámetros y por el tipo de los parámetros. Las tres reglas siguientes contemplan cada uno de estos casos:

5- En el caso de un mensaje sin parámetros en el modelo origen, se crea un link con nombre igual al mensaje que recibe la vista, y como destino una url con el nombre del mensaje que le envía la vista al controlador al que se le agrega el sufijo “.do”.
Por ejemplo: El mensaje logout () de la clase header genera un link en la página header.jsp con nombre logout y con destino logout.do

Si el mensaje tenía pre condiciones, las mismas se mantienen para mostrar o no el link correspondiente, por ejemplo, logout tiene como pre condición “userSession.getAttribute(“clientAccount”) <> null”, lo que genera en la correspondiente página JSP el código necesario para testear la pre condición, y en caso que sea válida, se muestra el link correspondiente.

6- Si el método en el modelo origen tiene parámetros, hay que tener en cuenta el tipo de los parámetros. Si los parámetros son de tipo simple, es decir, String o números, se puede esperar que sea el usuario el que ingrese esos valores. Por lo tanto, se genera en la correspondiente página, un formulario donde a cada parámetro se le hace corresponder un campo de entrada con el correspondiente tipo.
Por ejemplo: Para el método login (String username, String password) de la clase loginWindow se genera en la página login.jsp un formulario con dos campos de tipo texto, el primero llamado username y el segundo password. Análogamente a la regla anterior el destino del formulario está dado por el mensaje entre la vista y el controlador al que se le agrega el sufijo “.do”, en este caso será login.do

En la figura 5-15 mostramos una parte del PSM de las páginas JSP resultante de aplicar las reglas descritas.

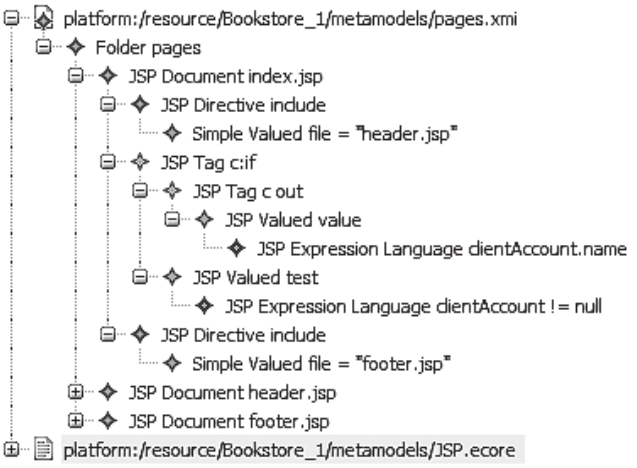

Figura 5-15. PSM de las vistas: parte de la página index.jsp.

5.5 Resumen

En este capítulo hemos desarrollado un ejemplo concreto de un desarrollo de software aplicando los conceptos de MDD. Hemos construido los modelos independientes de la plataforma que describen al sistema de manera abstracta y luego, a partir de ellos y haciendo uso de mecanismos de transformación, hemos obtenido los modelos específicos de una plataforma particular. En este capítulo tanto los modelos como las transformaciones han sido presentados informalmente (mediante diagramas y lenguaje natural), en los siguientes capítulos introduciremos el grado de formalización necesario para permitir la automatización del proceso de transformación aquí descrito.

CAPÍTULO 6

6. Lenguajes para definir las transformaciones

En este capítulo analizaremos algunos mecanismos para la definición de transformaciones modelo a modelo. Discutiremos sobre sus usos, presentaremos los requisitos necesarios que los lenguajes de estas características deberían satisfacer y finalmente introduciremos QVT, el estándar para transformaciones apoyado por el OMG.
Respecto a la distinción entre transformaciones modelo a modelo (M2M) y transformaciones modelo a texto (M2T), podemos destacar que la principal diferencia es que mientras la transformación modelo a modelo crea su modelo destino como una instancia de un metamodelo específico, el destino de una transformación modelo a texto es simplemente un documento en formato de texto.
En el próximo capítulo incluiremos la definición en QVT de las transformaciones de PIM a PSM que especificamos informalmente en el Capítulo 5, al desarrollar el sistema Bookstore. Posteriormente, un lenguaje específico para M2T será utilizado en la última etapa del desarrollo del sistema (transformaciones de PSMs a código).

6.1 Mecanismos para definir transformaciones modelo a modelo

En la categoría M2M podemos distinguir varios mecanismos para definir transformaciones. Czarnecki et al en [CH 06] propone una taxonomía para estos mecanismos de transformación. Cada mecanismo específico puede corresponder a más de una clase en la taxonomía. Las siguientes son las principales propuestas a considerar:

Manipulación directa
Con este mecanismo los usuarios tienen que implementar las reglas de transformación y de organización, también deben definir cómo manejar trazas y otras facilidades desde cero, en un lenguaje de programación de propósito general como Java.

Operacional
La propuesta operacional es similar a la manipulación directa pero ofrece soporte más orientado a transformación de modelos. Una solución típica en esta categoría es extender la sintaxis de un lenguaje de modelado mediante el agregado de facilidades para expresar las transformaciones. Por ejemplo podemos extender un lenguaje de consultas como OCL, con constructores imperativos. Esta versión de OCL ampliado y ejecutable, resulta en un sistema de programación orientado a objetos propiamente dicho. Ejemplos de sistemas en esta categoría son el lenguaje Operational Mappings de QVT, MTL [VJ 04] y Kermeta [MFJ 05]. Algunos de estos lenguajes brindan otras facilidades específicas, tales como mecanismos para rastreo de transformaciones.

Relacional
Esta categoría agrupa propuestas en las cuales el concepto principal está representado por las relaciones matemáticas. En general, las propuestas relacionales pueden verse como una forma de resolución de restricciones. Ejemplos de propuestas relacionales son el lenguaje Relations de QVT, MTF [MTF], Kent Model Transformation Language [AK 02] [AHM 05], Tefkat [GLRSW 02] [LS 05], entre otros.
Básicamente, una transformación se especifica a través de la definición de restricciones sobre los elementos del modelo fuente y el modelo destino. Sin bien son declarativas, estas transformaciones relacionales pueden tener semántica ejecutable implementada con programación lógica, usando matching basado en unificación, backtracking, etc. (véase [GLRSW 02]). Las propuestas relacionales soportan naturalmente reglas multi-direccionales.

Basada en transformaciones de grafos
Esta categoría reúne a las propuestas de transformación de modelos basadas en la teoría de transformaciones de grafos [EEKR 99]. En particular, este tipo de transformaciones actúa sobre grafos tipados, etiquetados y con atributos, que pueden pensarse como representaciones formales de modelos de clase simplificados. Las principales propuestas en esta categoría incluyen a AGG [Taentzer 03], AToM3 [deLV 02], VIATRA [VVP 02] [VP 04], GReAT [AKS 03], UMLX [Willink 03], MOLA [KBC 04],

y Fujaba [Fujaba]. AGG y AToM3 son sistemas que implementan directamente la propuesta teórica para grafos con atributos y también las transformaciones sobre estos grafos.
Las reglas de transformación son unidireccionales e *in-place*. Cada regla de transformación de grafo consta de dos patrones, el derecho y el izquierdo. La aplicación de una regla implica localizar un patrón izquierdo en el modelo y reemplazarlo por el correspondiente patrón derecho.

Híbrida
Las propuestas híbridas combinan diferentes técnicas de las categorías anteriores. Las distintas propuestas pueden ser combinadas como componentes separados o, en un modo granularmente más fino, a nivel de reglas individuales.
QVT es una propuesta de naturaleza híbrida, con tres componentes separadas, llamadas Relations, Operational mappings, y Core. La justificación de su naturaleza la veremos en detalle en la última sección de este capítulo. Ejemplos de combinación a nivel fino son ATL [ATL] y YATL [Patrascoiu 04].

Discusión sobre los mecanismos presentados
En este apartado, comentamos la aplicabilidad práctica de los diversos tipos de transformaciones de modelo. Estos comentarios se basan en nuestra experiencia y en los ejemplos de uso publicados junto con las propuestas. Debido a la carencia de experimentos controlados, estos comentarios no están completamente validados, pero esperamos que estimulen la discusión y fomenten la evaluación.
La manipulación directa es obviamente la propuesta más básica. Ofrece al desarrollador poca ayuda en la definición y ejecución de transformaciones. Esencialmente, todo el trabajo tiene que ser hecho por el desarrollador. La propuesta puede ser mejorada agregando bibliotecas especializadas y *frameworks* que implementen mecanismos tales como *pattern matching*, tipos abstractos de datos [IP 09] y trazas. Las propuestas operacionales son similares salvo que ofrecen un formalismo de metamodelado ejecutable mediante un lenguaje específico. Los mecanismos especializados son provistos a través de bibliotecas y frameworks, mejorando la asistencia para transformaciones de modelos de una manera evolutiva. Las propuestas relacionales parecen lograr un equilibrio razonable entre la flexibilidad y la expresividad declarativa. Pueden proporcionar multi-direccionalidad e incrementalidad. Por otra parte, su poder depende de la sofisticación de los mecanismos que incluyan para resolver restricciones. Consecuentemente, el funcionamiento depende en gran medida de los tipos de restricciones que

necesiten ser solucionadas, lo cual puede limitar su aplicabilidad. En todo caso, las propuestas relacionales parecen ser las más aplicables en escenarios de sincronización de modelos. Las propuestas basadas en transformaciones de Grafos, en su forma pura son declarativas e intuitivas; sin embargo, la programación habitual sobre grafos con aplicación concurrente los hace bastante difíciles de usar debido a la posible carencia de confluencia y terminación. Las teorías existentes para la detección de estos problemas no son lo suficientemente generales como para cubrir la amplia gama de transformaciones en la práctica. Como resultado, herramientas tales como GReAT, VIATRA y MOLA proporcionan mecanismos para la programación explícita. A menudo se afirma que las transformaciones de grafos son una elección natural para transformaciones de modelos, porque en general, los modelos son grafos. Asimismo, podemos considerar que una debilidad de las actuales teorías de transformación de grafos y herramientas es que no se consideran grafos ordenados (con relación de orden entre las aristas). Como consecuencia, son aplicables a los modelos que contienen predominantemente colecciones sin orden, tales como diagramas de clase, donde las clases poseen colecciones sin orden de atributos y métodos.
Finalmente, las propuestas híbridas permiten que el usuario combine diversos conceptos y paradigmas dependiendo del uso. Dado la amplia gama de escenarios prácticos, la propuesta híbrida resulta ser la más promisoria.

6.2 Requisitos para los lenguajes M2M

En esta sección discutiremos los requisitos más importantes (y no demasiado obvios) para la definición de lenguajes de transformación M2M y sus implementaciones. Esto nos permitirá comprender el fundamento de las construcciones del estándar QVT.

- Muchos escenarios reales de transformaciones M2M requieren que las reglas de transformación tengan la habilidad de determinar que elementos ya han sido calculados por otras reglas, porque cada regla usualmente abarca sólo un pequeño aspecto de toda la transformación. Una búsqueda es solamente posible si el motor tiene la habilidad de hacer algún registro de la *traza de la transformación*. Una traza de transformación puede entenderse como una huella o rastro en tiempo de ejecución de una transformación. Usualmente los lenguajes de transformación imperativos tienen un procedimiento de búsqueda más explícito,

mientras que los declarativos tienen una forma implícita de explotar la traza.

- Luego de ejecutar una transformación M2M sobre un modelo fuente para generar un modelo destino, es posible que el modelo fuente sufra modificaciones. Para propagar dichas modificaciones sobre le modelo destino la transformación deberá ejecutarse nuevamente. En este caso es importante que sólo realice los cambios requeridos al modelo destino existente (en lugar de regenerarse todo el modelo desde cero). Esto sólo es posible si hay un mecanismo de identificación sobre el modelo destino. Usualmente, la traza de la transformación contiene esta información. Este problema es conocido como *propagación de cambios*.

- Una transformación implícita o explícitamente define una relación entre sus modelos fuente y destino. En algunos escenarios, ambos modelos existen antes de que la transformación sea ejecutada y entonces, la relación entre ellos existe previamente también. En este caso, una transformación puede ser consultada para verificar si la relación existe y opcionalmente modificar el modelo destino para que la relación se mantenga satisfactoriamente. Este problema es diferente del escenario de *propagación de cambios*, porque para el primero uno puede asumir la existencia de una traza de transformación, mientras que en el último escenario la transformación puede no haberse ejecutado nunca.

- En numerosos casos el modelo fuente es muy grande. Como dijimos anteriormente, después de la primera ejecución de la transformación, suelen ocurrir algunos cambios en el modelo fuente que deben propagarse hacia el modelo destino. En este caso debería ser posible aproximar cuáles reglas de la transformación serán nuevamente ejecutadas y sobre qué subconjunto de elementos del modelo fuente se aplicarán. Puede requerirse un *análisis de impacto* sobre las reglas de transformación para implementar esto, así como disponer de información de trazas. La optimización de este punto no debe ser subestimada. Este tipo de requisito suele referirse como *actualización incremental*.

- Hay muchos escenarios de uso donde se requiere que los modelos destino, generalmente PSMs, puedan ser modificados manualmente por los modeladores, luego de generados. El motor de transformación debería respetar estos cambios en caso de pro-

ducirse una nueva transformación. Este problema se conoce como "política de conservación".

- Los elementos del modelo destino pueden ser construidos incrementalmente. Por ejemplo, en un escenario típico de transformación, las clases en el metamodelo fuente son transformadas en clases en el metamodelo destino en una primera fase. Luego, en una segunda fase, las asociaciones fuente se transforman en asociaciones relacionando a las clases que fueron generadas previamente.

- Es polémico incluir *transformaciones bidireccionales* como un requisito. Esto puede lograrse escribiendo dos o más transformaciones unidireccionales, o una transformación que pueda ejecutarse en ambas direcciones. Esto último sólo es posible en un escenario declarativo. Sin embargo, es cuestionable la utilidad de este requisito en la práctica. Una transformación bidireccional puede sólo ser definida y tener sentido cuando describe una relación de isomorfismo. En la práctica, este no es el caso habitual, ya que diferentes metamodelos usualmente describen aspectos a diferentes niveles de abstracción que se concretan en forma diferente.

- Dado que las transformaciones son elementos complejos es necesario contar con herramientas de desarrollo de transformaciones que permitan aplicar el paradigma "dividir para conquistar", es decir, que permitan definir una transformación compleja en términos de transformaciones mas simples, como fue propuesto en [PGPB 09].

Resulta bastante obvio afirmar que si bien las transformaciones M2M podrían implementarse utilizando directamente un lenguaje de programación de propósito general como Java, esto no lograría satisfacer los requisitos mencionados arriba. Por ello se ha trabajado en la definición de lenguajes específicos para la definición de transformaciones, que den soporte a estos requisitos.

6.3 El estándar QVT para expresar transformaciones de modelos

En esta sección presentaremos el lenguaje estándar para expresar transformaciones de modelos, conocido como QVT (acrónimo inglés de

"Queries, Views and Transformations"). QVT se ha convertido en una especificación voluminosa y compleja, por lo que, respecto a su descripción, mostraremos sólo una vista general de su arquitectura y propiedades mediante pequeños ejemplos. En el próximo capítulo se presentarán ejemplos más completos; en particular incluiremos la definición en QVT de las transformaciones obtenidas al desarrollar el sistema Bookstore.
El lector se preguntará porque QVT se denomina de esa forma si solamente maneja transformaciones, es decir, ¿cómo se justifican la Q y la V? Los Queries son partes intrínsecas de cualquier transformación, facilitan consultas para selección y filtrado de elementos del modelo de entrada; esto se logra con OCL. La idea de View es permitir la visualización en una forma específica de metamodelos MOF. Puede pensarse que las transformaciones de modelos proveen mecanismos para esto, pero la especificación adoptada recientemente, elude el problema de las vistas. Si las vistas necesitan ser editadas, entramos en el problema del requisito de transformaciones bidireccionales y sus consecuencias. QVT clama por estas transformaciones, porque dos de sus lenguajes pueden especificar reglas bidireccionales, pero no indica cómo se definen las vistas en estos casos.
Comenzaremos con una breve revisión de su historia, continuaremos con su descripción y finalmente incluiremos algunas conclusiones.

6.3.1 Historia

La guía MDA [MDAG] definida por el OMG, habla sobre transformaciones entre modelos a diferentes niveles de abstracción. Por ejemplo, asume que un escenario típico de desarrollo basado en MDA consiste en la transformación de un PIM a un PSM, a partir del cual se generará posteriormente el código. La guía MDA intencionalmente no dice como se supone que esto suceda, y es aquí donde QVT entra en juego.
El OMG inicialmente publicó un *Request for Proposal* (RFP) para transformaciones M2M, a comienzos de 2002 [QVTR]. Recién hacia finales de 2005 fue conocida la especificación adoptada. Esta tardanza puede explicarse por la complejidad del problema que se abarca.
Aunque mucha gente pueda tener buenas ideas acerca de cómo escribir transformaciones M2M en Java, por ejemplo, es claro que los escenarios M2M reales requieren técnicas más sofisticadas. El RFP de QVT solicitó propuestas dirigidas a este nivel de sofisticación, aún pensando que tales requisitos no estaban muy explorados ni conocidos en ese momento. Otro problema fue que la experiencia previa existente con

transformaciones M2M era prácticamente nula en los comienzos. Esto no fue una buena posición de partida desde la cual generar un estándar, que idealmente es una consolidación de tecnologías existentes. Esta situación fue agravada por la multiplicidad y diversidad de las propuestas recibidas. Por esta razón no hubo bases claras para la consolidación. Como resultado, llevó una considerable cantidad de tiempo encontrar coincidencias y aún hoy, el resultado obtenido especifica tres lenguajes QVT diferentes que sólo se conectan débilmente, justificando la naturaleza híbrida del estándar –relacional y operacional- que intenta abarcar las diferentes respuestas al RFP.

6.3.2 Descripción general de QVT

QVT es el estándar del OMG para escribir transformaciones. Los conceptos, definiciones y ejemplos que siguen, están extraídos de su documento de especificación MOF 2.0 Query/View/Transformation [QVT].
La especificación de QVT, como dijimos, tiene una naturaleza híbrida, relacional (o declarativa) y operacional (o imperativa). Comprende tres diferentes lenguajes M2M: dos lenguajes declarativos llamados *Relations* y *Core*, y un tercer lenguaje, de naturaleza imperativa, llamado *Operational Mappings*. Como anticipamos en la sección anterior, esta naturaleza híbrida fue introducida para cubrir diferentes tipos de usuarios con diferentes necesidades, requisitos y hábitos. Esta estrategia no resulta sorprendente si consideramos las numerosas propuestas iniciales generadas por el RFP, cada una con distintas expectativas de uso y funcionalidad.
Entonces, la especificación de QVT define tres paquetes principales, uno por cada lenguaje definido: QVTCore, QVTRelation y QVTOperational, como puede observarse en la figura 6-1. Estos paquetes principales se comunican entre sí y comparten otros paquetes intermedios (figura 6-2).
El paquete QVTBase define estructuras comunes para transformaciones. El paquete QVTRelation usa expresiones de patrones *template* definidas en el paquete QVTTemplateExp. El paquete QVTOperational extiende al QVTRelation, dado que usa el mismo framework para trazas. Usa también las expresiones imperativas definidas en el paquete ImperativeOCL. Todos los paquetes QVT dependen del paquete EssentialOCL de OCL 2.0, y a través de él también dependen de EMOF (EssentialMOF).

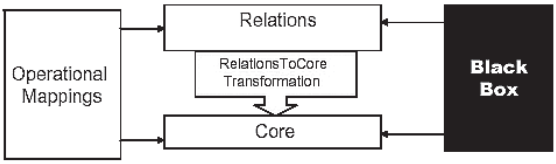

Figura 6-1. Relaciones entre metamodelos QVT

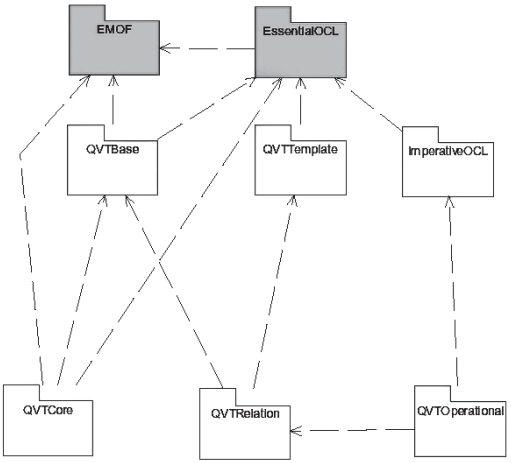

Figura 6-2. Dependencias de paquetes en la especificación QVT

En las siguientes secciones continuaremos explicando la arquitectura de las partes declarativa y operacional de QVT.

6.3.3 QVT declarativo

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

- Un lenguaje *Relations,* amigable para el usuario, que soporta *pattern matching* complejo de objetos y creación de *template* para objetos. Las trazas entre elementos del modelo involucrados en una transformación se crean implícitamente. Soporta propagación de cambios, ya que provee un mecanismo para identificar elementos del modelo destino. Describimos este lenguaje más detalladamente en el siguiente apartado.

- Un lenguaje *Core,* definido usando extensiones minimales de EMOF y OCL. Las trazas no son automáticamente generadas, se definen explícitamente como modelos MOF, y pueden crearse y borrarse como cualquier otro objeto. El lenguaje Core no soporta *pattern matching* para los elementos de modelos. Esta propuesta absolutamente minimal lleva a que el Core sea el "assembler" de los lenguajes de transformación.

Lenguaje Relations

Es una especificación declarativa de las relaciones entre modelos MOF. En este lenguaje, una transformación entre modelos se especifica como un conjunto de relaciones que deben establecerse para que la transformación sea exitosa.
Los modelos tienen nombre y los elementos que contienen deben ser de tipos correspondientes al metamodelo que referencian. Por ejemplo:

```
transformation umlRdbms(uml:SimpleUML,rdbms:SimpleRDBMS){…}
```

En esta declaración llamada "umlRdbms," hay dos modelos tipados: "uml" y "rdbms". El modelo llamado "uml" declara al paquete SimpleUML como su metamodelo y el modelo "rdbms" declara al paquete SimpleRDBMS como su metamodelo. Una transformación puede ser invocada para *chequear* consistencia entre dos modelos o para modificar un modelo *forzando* consistencia.
Cuando se fuerza consistencia, se elige el modelo destino; éste puede estar vacío o contener elementos a ser relacionados por la transformación. Los elementos que no existan serán creados para forzar el cumplimiento de la relación.
En el ejemplo que sigue, en la relación PackageToSchema, el dominio para el modelo "uml" está marcado como *checkonly* y el dominio para el modelo "rdbms" está marcado *enforce*. Estas marcas habilitan la modificación (creación/borrado) de elementos en el modelo "rdbms", pero no

en el modelo “uml”, el cual puede ser solamente leído pero no modificado.

```
relation PackageToSchema
/* transforma cada paquete UML a un esquema relacional
*/
{
      checkonly domain uml p:Package {name=pn}
      enforce domain rdbms s:Schema {name=pn}
}
```

- **Relaciones, dominios y *pattern matching***

Las relaciones en una transformación declaran restricciones que deben satisfacer los elementos de los modelos. Una relación se define por dos o más dominios y un par de predicados *when* y *where*, que deben cumplirse entre los elementos de los modelos. Un dominio es una variable con tipo que puede tener su correspondencia en un modelo de un tipo dado. Un dominio tiene un patrón, que se puede ver como un conjunto de variables y un conjunto de restricciones que los elementos del modelo asocian a aquellas variables que lo satisfacen (*pattern matching*). Un patrón del dominio es un *template* para los objetos y sus propiedades que deben ser inicializadas, modificadas, o creadas en el modelo para satisfacer la relación. En el ejemplo de abajo, en la relación PackageToSchema se declaran dos dominios que harán correspondencia entre elementos de los modelos “uml” y “rdbms” respectivamente. Cada dominio especifica un patrón simple: un paquete con un nombre, y un esquema con un nombre, en ambos la propiedad “name” asociada a la misma variable “pn” implica que deben tener el mismo valor.
A su vez, en la relación ClassToTable, se declaran también dos dominios. Cada dominio especifica patrones más complejos que en la otra relación: por un lado una clase con un espacio de nombres, de tipo persistente y con un nombre. Por otro lado, el otro dominio especifica una tabla con su esquema, un nombre y una columna cuyo nombre se formará en base al nombre de la clase UML y define a esta columna como clave primaria de la tabla. Por la propiedad “name” asociada a la misma variable “cn”, la clase y la tabla deben tener el mismo nombre. Además la cláusula *when* en la relación indica que previamente a su aplicación debe satisfacerse la relación PackageToSchema, mientras que la cláusula *where* indica que siempre que la relación ClassToTable se establezca, la relación AttributeToColumn debe establecerse también.

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

Relaciones top-level

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

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

Una relación topLevel tiene la palabra *top* antecediéndola para distinguirla sintácticamente. En el ejemplo de arriba, PackageToSchema y ClassToTable son relaciones topLevel, mientras que AttributeToColumn es una relación no topLevel, que será invocada por alguna de las otras para su ejecución.

• Claves y creación de objetos usando patrones

Como ya mencionamos, una expresión *object template* provee una *plantilla* para crear un objeto en el modelo destino. Cuando para un patrón válido en el dominio de entrada no existe el correspondiente elemento en la salida, entonces la expresión *object template* es usada como *plantilla* para crear objetos en el modelo destino.
Por ejemplo, cuando ClassToTable se ejecuta con el modelo destino "rdbms", la siguiente expresión *object template* sirve como *plantilla* para crear objetos en el modelo destino "rdbms":

```
t:Table {
       schema = s:Schema {},
       name = cn,
       column = cl:Column {name=cn+'_tid', type='NUMBER'},
       primaryKey = k:PrimaryKey {name=cn+'_pk', column=cl}
}
```

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

Sintaxis abstracta del lenguaje relations

En esta sección presentamos los principales elementos que conforman la sintaxis abstracta del lenguaje Relations. Para ver la descripción detallada de todas las metaclases, debe consultarse el manual de QVT.
La figura 6-3 muestra el Paquete QVTBase para Transformaciones y Reglas, del cual detallamos las metaclases ***Transformation, Rule y Domain.***

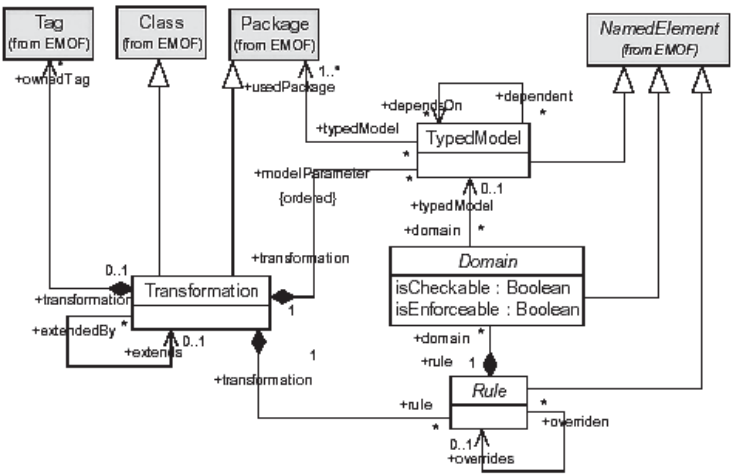

Figura 6-3. Paquete QVT Base – Transformaciones y Reglas.

Una ***transformation*** define cómo un conjunto de modelos puede ser transformado en otro. Contiene un conjunto de reglas (*rules)* que especifican su comportamiento en ejecución. Se ejecuta sobre un conjunto de modelos con tipo, especificados por un conjunto de parámetros (*typed model)* asociados con la transformación.

Superclases

`Package`
`Class`

Asociaciones

`modelParameter: TypedModel [*] {composes}`

El conjunto de modelos tipados que especifican los tipos de modelos que pueden participar en la transformación.

`rule: Rule [*] {composes}`

Las reglas contenidas en la transformación, las cuales juntas especifican el comportamiento en ejecución de la transformación.

`extends: Transformation [0..1]`

Una transformación puede ser extendida por otra.

Una ***rule*** especifica cómo están relacionados los elementos especificados por sus dominios con cada uno de los otros, y cómo los elementos de un dominio son calculados desde los elementos del otro dominio. Rule es una clase abstracta, cuyas subclases concretas son las responsables de especificar la semántica exacta de cómo se relacionan y se computan los dominios entre ellos. Una *rule* puede reescribir a otra. La *rule* que reescribe se ejecuta en lugar de la anterior.

Superclases

`NamedElement`

Asociaciones

`domain: Domain [*] {composes}`

Los dominios que componen la *rule*.

`transformation: Transformation[1]`

La transformación dueña de la *rule*.

`overrides: Rule [0..1]`

La *rule* reescrita por esta *rule*.

Un ***domain*** especifica un conjunto de elementos de un *typed model* que son de interés a una *rule*. Domain es una clase abstracta, cuyas subclases concretas son las responsables de especificar el mecanismo exacto por el cual el conjunto de elementos de un dominio puede ser especificado. Puede especificarse como un grafo de patrones, como un conjunto de variables tipadas y restricciones, o con cualquier otro mecanismo adaptable. Como ya dijimos, un dominio puede marcarse como *checkable* o e*nforceable*.

Superclases

`NamedElement`

Atributos

`isCheckable : Boolean`

Indica si el dominio es chequeable o no.

`isEnforceable : Boolean`

Indica si el dominio es forzado o no.

Asociaciones

`rule: Rule [1]`

La regla que es dueña del dominio.

`typedModel: TypedModel [0..1]`

El modelo tipado que contiene los tipos de los elementos del modelo especificado por el dominio.

Estas metaclases básicas son utilizadas en la siguiente parte del metamodelo, que mostramos en la figura 6-4. Esta parte del metamodelo se agrupa dentro del paquete QVTRelations, del cual detallamos las metaclases ***RelationalTransfomation*** y ***Relation***:

Una ***relationalTransformation*** es una especialización de *Transformation* y representa una definición de transformación escrita en el lenguaje QVTRelation.

Superclases

`Transformation (from QVTBase)`

`NamedElement`

Asociaciones

`key: Key [*] {composes}`

Las claves definidas en la transformacion.

Una ***relation*** es la unidad básica de especificación del comportamiento de la transformación en el lenguaje Relations. Es una subclase concreta de *Rule*. Especifica una relación que debe establecerse entre los elementos de un conjunto de modelos candidatos que conforman a los *typed models* de la transformación dueña de la relación. Se define por dos o más *relation domains* que especifican los elementos de modelos a relacionar. Su cláusula *when* especifica las condiciones necesarias para que la relación se establezca, y su cláusula *where* especifica la condición que debe satisfacerse por los elementos del modelo que están siendo relacionados. Una *relation* puede opcionalmente tener asociada una implementación operacional *black-box* para forzar un dominio.

Superclases

`Rule`

Atributos

`isTopLevel : Boolean`

Indica cuando la relación es top-level o cuando es una relación no top-level.

Asociaciones

`variable: Variable [*] {composes}`

El conjunto de variables en la relación. Este conjunto incluye todas las currencias de variables en sus dominios y sus cláusulas **when** y **where**.

`/domain: Domain [*] {composes} (from Rule)`

El conjunto de dominios contenidos en la relación que especifican los elementos de modelo que participan de la relación. Relation hereda esta asociación de Rule, y está restringida a contener solamente RelationDomains vía esta asociación.

`when: Pattern [0..1] {composes}`

El patrón (como un conjunto de variables y predicados) que especifica la cláusula **when** de la relación.

`where: Pattern [0..1] {composes}`

El patrón (como un conjunto de variables y predicados) que especifica la cláusula **where** de la relación.

`operationalImpl: RelationImplementation [*] {composes}`

El conjunto de implementaciones operacionales *black-box,* si las hay, que están asociadas con la relación para forzar sus dominios.

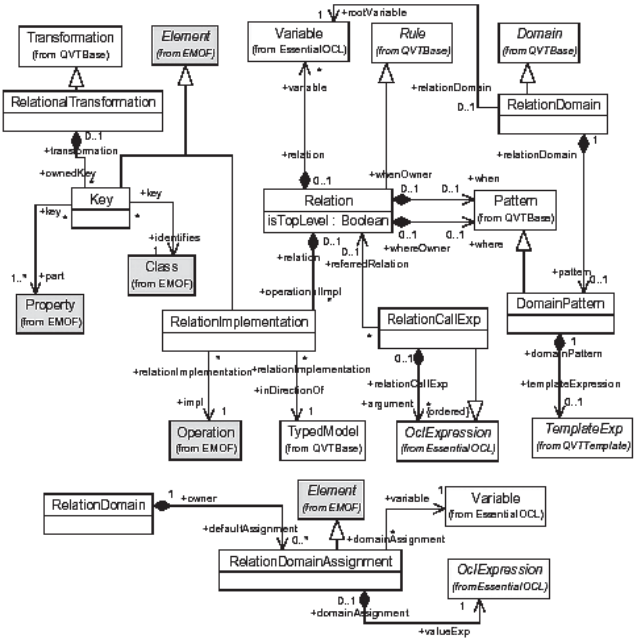

Figura 6-4. Paquete QVT Relations

Finalmente, la figura 6-5 presenta el Paquete QVT Template, del cual detallamos la metaclase ***TemplateExp***:

Una ***TemplateExp*** especifica un patrón de correspondencia (o plantilla) con elementos de modelos definidos en una transformación. Los elementos del modelo pueden ligarse a variables y estas variables pueden ser usadas en otras partes de la expresión. Una *expresión template* puede corresponder tanto a elementos simples como a una colección de ellos, dependiendo si es una expresión *object template* (metaclase ObjectTemplateExp) o una expresión *collection template* (metaclase CollectionTemplateExp), las cuales son subclases de *TemplateExp*.

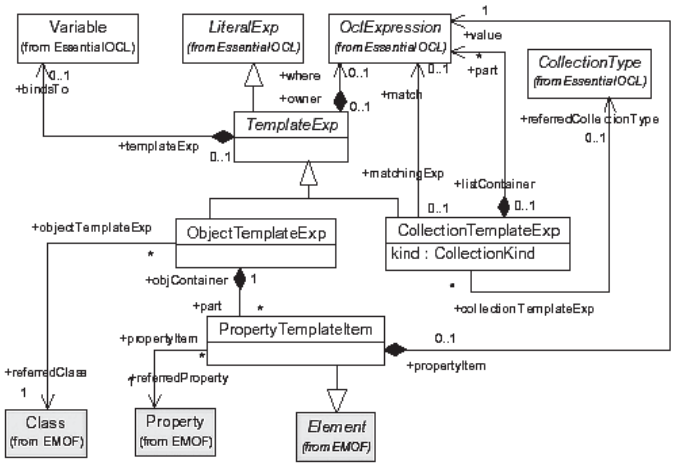

Figura 6-5. Paquete QVT Template

Superclases

`LiteralExp`

Associaciones

`bindsTo: Variable [0..1] {composes}`

La variable que refiere al elemento del modelo asociado por su expresión *template*.

`where: OclExpression [0..1] {composes}`

Expresión booleana que debe ser verdadera para que se liguen los elementos mediante la expresión *template*.

6.3.4 QVT operacional

Además de sus lenguajes declarativos, QVT proporciona dos mecanismos para implementaciones de transformaciones: un lenguaje estándar, *Operational Mappings*, e implementaciones no-estándar o *black-box*.
El mecanismo de caja negra (*black-box)*, permite implementaciones opacas, escritas en otro lenguaje, que pueden ser invocadas desde QVT. Por su parte, el lenguaje *Operational Mappings* se especificó como una forma estándar para proveer implementaciones imperativas. Proporciona una extensión del lenguaje OCL mediante el agregado de nuevas construcciones con efectos laterales que permiten un estilo más procedural, y una sintaxis que resulta familiar a los programadores.
Este lenguaje puede ser usado en dos formas diferentes. Primero, es posible especificar una transformación únicamente en el lenguaje *Operational Mappings.* Una transformación escrita usando solamente operaciones *Mapping* es llamada transformación operacional. Alternativamente, es posible trabajar en modo híbrido. El usuario tiene entonces que especificar algunos aspectos de la transformación en el lenguaje Relations (o Core), e implementar reglas individuales en lenguaje imperativo a través de operaciones *Mappings.*

Una **transformación *Operacional*** representa la definición de una transformación unidireccional, expresada imperativamente. Tiene una signatura indicando los modelos involucrados en la transformación y define una operación *entry*, llamada “main”, la cual representa el código inicial a ser ejecutado para realizar la transformación. La signatura es obligatoria, pero no así su implementación. Esto permite implementaciones de *caja negra* definidas fuera de QVT.
El ejemplo que sigue muestra la signatura y el punto de entrada de una transformación llamada *Uml2Rdbms,* que transforma diagramas de clase UML en tablas RDBMS.

```
transformation Uml2Rdbms(in uml:UML,out rdbms:RDBMS) {
//el punto de entrada para la ejecución de la
transformación.
main() {
uml.objectsOfType(Package)->map packageToSchema();
}
….
}
```

La signatura de esta transformación declara que un modelo rdbms de tipo RDBMS será derivado desde un modelo uml de tipo UML. En el ejemplo, el cuerpo de la transformación (m*ain*) especifica que en primer lugar se recupera la lista de objetos de tipo Paquete y luego se aplica la operación de mapeo (mapping operation) llamada packageToSchema() sobre cada paquete de dicha lista. Esto último se logra utilizando la operación predefinida map() que itera sobre la lista.

Una ***MappingOperation*** es una operación que implementa a una correspondencia entre uno o más elementos del modelo fuente y uno o más elementos del modelo destino. Una mappingOperation se describe sintácticamente mediante una signatura, una guarda (su cláusula *when*), el cuerpo del mapping y una post condición (su cláusula *where*). La operación puede no incluir un cuerpo (es decir, oculta su implementación) y en ese caso se trata de una operación de caja negra (black-box). Una *mapping operation* siempre refina una relación, donde cada dominio se corresponde con un parámetro del *mapping.* El cuerpo de una operación *mapping* se estructura en tres secciones opcionales. La sección de inicialización es usada para crear los elementos de salida. La intermedia, sirve para asignarle valores a los elementos de salida y la de finalización, para definir código que se ejecute antes de salir del cuerpo. La operación *mapping* que sigue define cómo un paquete UML se transforma en un esquema RDBMS.

```
mapping Package::packageToSchema() : Schema
       when { self.name.startingWith() <> "_"}
{
       name := self.name;
       table := self.ownedElement->map class2table();
}
```

La relación implícita asociada con esta operación mapping tiene la siguiente estructura:

```
relation REL_PackageToSchema {
       checkonly domain:uml (self:Package)[]
       enforce domain:rdbms (result:Schema)[]
       when { self.name.startingWith() <> "_" }
}
```

Sintaxis abstracta del lenguaje Operational Mappings

En esta sección presentamos los principales elementos que definen la sintaxis abstracta del lenguaje *Operational Mappings*. El lector podrá encontrar la descripción detallada de todas las metaclases en el documento de especificación de QVT.

La figura 6-6 muestra el Paquete QVT Operational para Transformaciones Operacionales, del cual detallamos la metaclase ***OperationalTransformation***:

Una ***OperationalTransformation*** representa la definición de una transformación unidireccional, expresada imperativamente.

Superclases

`Module`

Atributos

`/isBlackbox : Boolean (from Module)`

Indica que la transformación es opaca.

`/isAbstract : Boolean (from Class)`

Indica que la transformación sirve para la definición de otras transformaciones.

Asociaciones

`entry : EntryOperation [0..1]`

Una operación actuando como punto de entrada para la ejecución de la transformación operacional.

`modelParameter: ModelParameter [*] {composes, ordered}`

Indica la signatura de esta transformación operacional. Un parámetro de modelo indica un tipo de dirección (in/out/inout) y un tipo dado por un tipo de modelo (ver la descripción de clase ModelParameter).

`refined : Transformation [0..1]`

Indica una transformación relacional (declarativa) que es refinada por esta transformación operacional.

`relation : Relation [0..*] {composes, ordered}`

El conjunto ordenado de definiciones de relaciones que tienen que están asociadas con las operaciones *mapping* de esta transformación operacional.

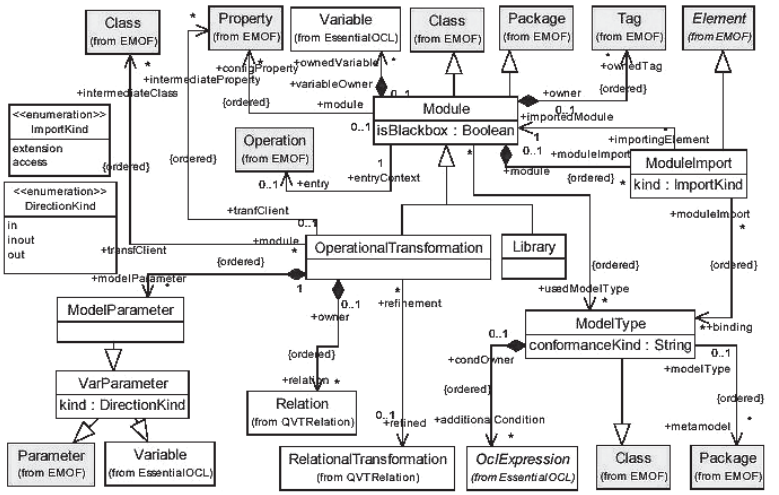

Figura 6-6. Paquete QVT Operational – Transformaciones Operacionales

La figura 6-7 presenta la jerarquía de las Operaciones Imperativas del Paquete QVT Operational, de la que detallamos la metaclase ***MappingOperation***.

Una ***MappingOperation*** es una operación implementando un *mapping* entre uno o más elementos del modelo fuente, en uno o más elementos del modelo destino.

Superclases

`ImperativeOperation`

Atributos

`isBlackbox: Boolean`

Indica si el cuerpo de la operación está disponible. Si *isBlackbox* es verdadero esto significa que la definición debería ser proporcionada externamente a fin de que la transformación pueda ser ejecutada.

Asociaciones

`inherited: MappingOperation [*]`

Indica la lista de operaciones *mappings* que son especializados.

`merged: MappingOperation [*]`

Indica la lista de operaciones *mappings* que son mezclados.

`disjunct: MappingOperation [*]`

Indica la lista de operaciones *mappings* potenciales para invocar

`refinedRelation: Relation [1]`

Indica la relación refinada. La relación define la guarda (*when*) y la poscondición para la operación *mapping*.

`when: OclExpression [0..1] {composes}`

La pre-condición o guarda de la operación.

`where: OclExpression [0..1] {composes}`

La post-condición o guarda de la operación.

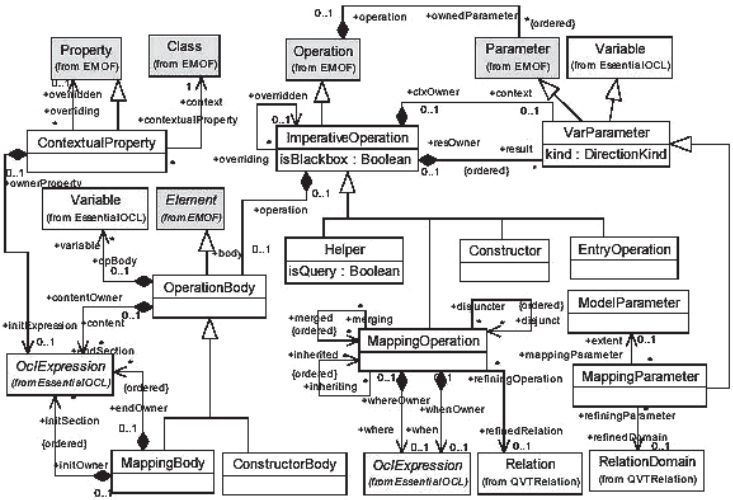

Figura 6-7. Paquete QVT Operational – Operaciones Imperativas

Finalmente, la figura 6-8 muestra el Paquete OCL Imperativo que extiende a OCL proveyendo expresiones imperativas y manteniendo las ventajas de la expresividad de OCL.

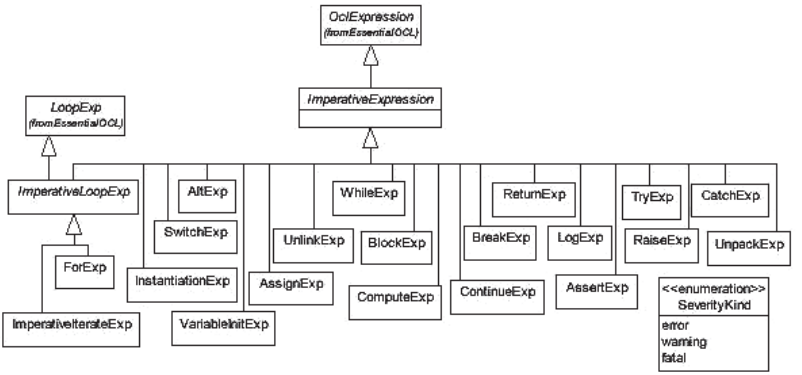

Figura 6-8. Paquete OCL Imperativo

ImperativeExpression es la raíz abstracta de esta jerarquía sirviendo como base para la definición de todas las expresiones con efectos laterales definidas en esta especificación. Tales expresiones son AssignExp, WhileExp, IfExp, entre otras. La **Superclase** de *ImperativeExpression* es OclExpression. Podemos notar que en contraste con las expresiones OCL puras, libres de efectos laterales, las expresiones imperativas en general, no son funciones. Por ejemplo, son constructores de interrupción de ejecución como break, continue, raise y return que producen un efecto en el flujo de control de las expresiones imperativas que las contienen.

6.4 Conclusiones

En este capítulo hemos analizado los principales mecanismos existentes para la definición de transformaciones modelo a modelo. A continuación, hemos introducido QVT, el estándar para transformaciones promovido por el OMG. Existen también otros lenguajes para transformaciones M2M y herramientas disponibles que no se alinean con QVT; algunas propietarias, otras de código abierto. La demora en el uso de los estándares puede radicar en que muy pocos proyectos en escala industrial han usado transformaciones M2M y por otro lado, en que MDD es un paradigma reciente con requisitos aún en proceso de consolidación.

Hemos analizado algunos requisitos relevantes para que un lenguaje de transformaciones M2M sea práctico y usable. Al desarrollar un lenguaje M2M, la dificultad está en encontrar un equilibrio entre un lenguaje usable para el desarrollador y la posibilidad de implementar el lenguaje en forma razonablemente eficiente y lógicamente integrable en un ambiente de desarrollo MDD. La posibilidad de rastrear las transformaciones ha sido señalada también como un requisito importante.

Respecto a la documentación, la especificación de QVT es un documento que fue sometido a varias y distintas revisiones, que fueron incorporadas incrementando su dimensión. El resultado es un documento complejo. QVT especifica tres lenguajes diferentes (Relations, Core y Operational Mappings). Su naturaleza híbrida –relacional y operacional- se justifica en la intención de abarcar las diferentes respuestas del RFP, satisfaciendo así las distintas preferencias en el mercado. El desarrollador de transformaciones podrá optar por definir transformaciones imperativas, transformaciones declarativas o bien transformaciones que combinen ambos estilos.

En términos de uso del lenguaje, tanto el lenguaje Relations como el Core parecen haber pagado un precio alto al soportar mappings bidireccionales, mecanismo que carece mayormente de aplicación práctica. Además, estos lenguajes declarativos no tienen mecanismos para manejar excepciones, lo que representa una omisión importante. El lenguaje operacional es bastante complejo; una transformación M2M es intrínsecamente un problema funcional que no parece integrarse fácilmente al paradigma de la orientación a objetos para su implementación. Además, el desarrollo de transformaciones M2M requiere herramientas sofisticadas provistas de editores inteligentes y facilidades de debugging avanzadas.

La definición del estándar no incluye normativas acerca de cómo se construyen herramientas para el lenguaje, pero debería definir las bases para que pueda ser implementado razonablemente. Actualmente existen herramientas para transformación de modelos que definen su propio lenguaje. También han surgido herramientas que implementan al estándar declarativo y otras al operacional. Algunas de las principales herramientas para transformaciones serán abordadas en los próximos capítulos.

Finalmente, cabe destacar que los principales esfuerzos han sido dedicados a la definición de la sintaxis de estos lenguajes, mientras que la definición formal de su semántica no ha generado el mismo interés y es aún un tema en proceso de investigación. En ese sentido se destacan las propuestas descritas en [BM 08], [BCR 06], [Poernomo 08] y [GPP 09].

CAPÍTULO 7

7. Definición formal de las transformaciones usando QVT

En este capítulo mostraremos ejemplos de transformaciones entre modelos escritas formalmente utilizando el lenguaje estándar QVT, el cual fue introducido en el capítulo anterior. Los ejemplos están basados en el caso de estudio presentado en el capítulo 5. Aquí presentamos la definición formal de las mismas transformaciones que han sido descriptas previamente para transformar el PIM del Sistema de Venta de Libros por Internet (Bookstore) en un PSM de tres capas específicas de la plataforma. En el capítulo 5 las transformaciones fueron escritas informalmente en lenguaje natural; las definiciones que elaboraremos en este capítulo serán lo suficientemente formales como para permitir que las transformaciones puedan ser ejecutadas automáticamente por una computadora. Sólo mostraremos algunas partes del código de las transformaciones, la implementación completa puede obtenerse en el siguiente sitio:
http://www.lifia.info.unlp.edu.ar/bookstore

7.1 La transformación de UML a relacional

En esta sección presentamos la definición de la transformación para convertir un PIM escrito en UML a un PSM dependiente del modelo relacional. Como hemos visto en el capítulo 3, para definir la transformación formalmente necesitamos, en primer lugar, los metamodelos tanto del lenguaje fuente como del lenguaje destino de la transformación. Luego, en el código de la transformación nos referiremos a los elementos (o instancias) de dichos metamodelos. Por lo tanto, necesitamos el metamodelo de UML y el metamodelo relacional. Usaremos una versión simplificada del metamodelo del estándar UML 2.0 [UML 03], como mostramos en la figura 7-1.

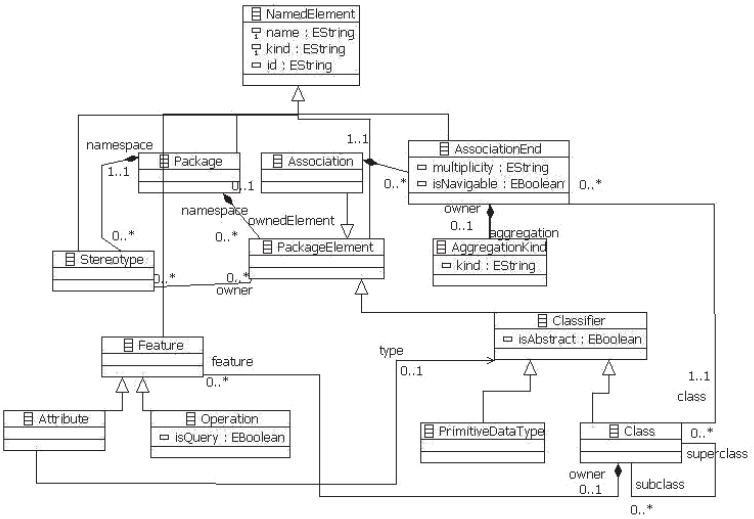

Figura 7-1. Metamodelo de UML simplificado

En la figura 7-2 presentamos el metamodelo relacional. Este metamodelo es una versión simplificada del metamodelo estándar para SQL definido en [CWM 03].

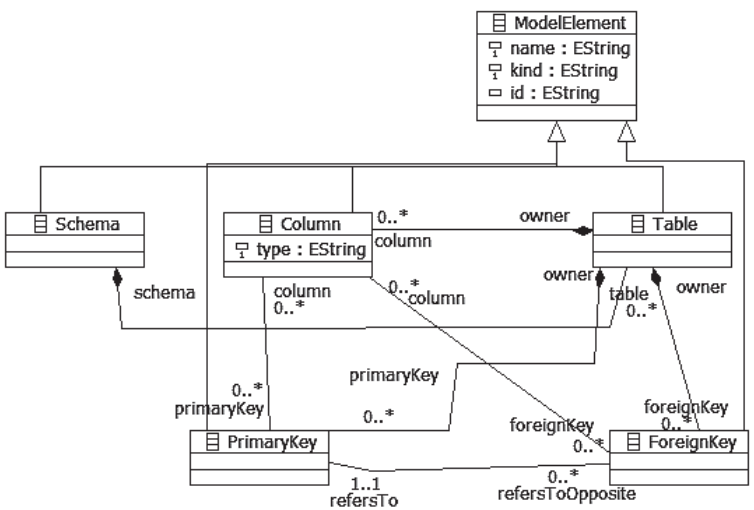

Figura 7-2. Metamodelo relacional simplificado

Basándonos en los metamodelos del lenguaje fuente y del lenguaje destino, podemos comenzar a escribir la definición de las transformaciones.

La primera regla define la transformación de un Paquete UML a un Esquema de la base de datos relacional, como ya explicamos en el capítulo 5. Esta regla se complementa con las reglas que transforman las clases en tablas, los atributos en columnas y las asociaciones en relaciones de claves foráneas, entre otras.

```
transformation PIM2PSMREL(uml:UMLModel, rdbms:RelModel) {

/*  Se genera un esquema por cada paquete */

top relation PackageToSchema {

        pn : String;

        checkonly domain uml p : UMLModel::Package {name = pn};

        enforce domain rdbms s : RelModel::Schema {name = pn};

      }

/*  Se genera una tabla por cada clase */

top relation ClassToTable {

        cn : String;

        checkonly domain uml c : UMLModel::Class {
          namespace = p : UMLModel::Package {},
          stereotype = s : UMLModel::Stereotype
{name='persistent'},
          isAbstract = false,
          name = cn
        };

        enforce domain rdbms t : RelModel::Table {
          schema = sch : RelModel::Schema {},
          name = cn,
          column = newColumn : RelModel::Column{
```

```
            name = cn.firstToLower()+'ID', type = 'INTEGER'
},
            primaryKey = k : RelModel::PrimaryKey {
            column = newColumn : RelModel::Column{}}
        };

        when { PackageToSchema(p, sch); }
        where { ClassToPrimaryKey(c, k);}
      }

/*  se generan las claves primarias */

      relation ClassToPrimaryKey {

        cn : String;

        checkonly domain uml c : UMLModel::Class {
          name = cn
        };

        enforce domain rdbms k : UMLModel::PrimaryKey {
          name = cn.firstToLower() + '_pk'
        };

      }

/*  Los atributos se transforman en columnas */

      top relation AttributeToColumn {

        aName, attributeType, columnType: String;

        checkonly domain uml attribute :
UMLModel::Attribute {
            owner = class : UMLModel::Class{},
            name = aName,
            type = attrType :UMLModel::PrimitiveDataType {
                  name = attributeType}
        };

        enforce domain rdbms column : RelModel::Column {
            owner = table :RelModel::Table{},
```

```
            name = aName,
            type = columnType
        };

        when {
            ClassToTable(class,table);
        }
        where {
            columnType =
primitiveTypeToRelType(attributeType);
        }
      }


/*  Se agrega una foreignKey por cada relación "1..*" */

      top relation AssociationToForeignKey {
        an, fkn, fcn, multNotN : String;

        checkonly domain uml a : UMLModel::Association {
          namespace = p : UMLModel::Package {},
          name = an,
          associationEnd = ae1 : UMLModel::AssociationEnd {
            multiplicity = multNotN,
            class = class1 : UMLModel::Class {
            stereotype = st1 : UMLModel::Stereotype{
              name = 'persistent' }
            }
          },
          associationEnd = ae2 : UMLModel::AssociationEnd {
            multiplicity = '*',
            class = class2 : UMLModel::Class {
            stereotype = st2 : UMLModel::Stereotype{
              name = 'persistent' }
            }
          }
        };

        enforce domain rdbms fk : RelModel::ForeignKey {
          name = fkn,
          owner = destTbl : RelModel::Table {
            schema = s : RelModel::Schema {}
```

```
        },
        column = fc : RelModel::Column {
          name = fcn,
          type = 'INTEGER',
          owner = destTbl
        },
        refersTo = pKey : RelModel::PrimaryKey {
          owner = destTbl : RelModel::Table {}
        }
      };

      when {
        ClassToPrimaryKey(class2, pKey);
        PackageToSchema(p, s);
        ClassToTable(class2, destTbl);
        multNotN <> '*';
      }

      where {
        fkn = class1.name.firstToLower()+'ID';
        fcn = fkn;
      }
    }
}
```

7.2 La transformación de UML a Java Beans

En esta sección especificamos las transformaciones para convertir un PIM escrito en UML a un modelo Java Beans. Las definiciones se basan en las reglas que presentamos en la sección 5.4.2. Tanto el lenguaje fuente como el lenguaje destino es UML, para el cual ya hemos presentado un metamodelo en la figura 7-1. El siguiente código QVT ilustra algunas de las transformaciones necesarias para generar los objetos del dominio y los mecanismos para acceder a los datos:

```
transformation PIM2PSMJava(uml:UMLModel, java:UMLModel) {

/* Transformación de paquetes UML  a paquetes Java */

    top relation PackageToPackage {
```

```
        pn : String;

        checkonly domain uml srcPackage:UMLModel::Package
{name=pn};

        enforce domain java tarPackage : UMLModel::Package
{name=pn};

      }

/*  Transformación de las clases del dominio en clases
Java */

      top relation ClassToJavaClass{

        className, stName : String;

        checkonly domain uml umlClass : UMLModel::Class{
          namespace = srcPackage : UMLModel::Package{},
          name = className,
          stereotype = s1 : UMLModel::Stereotype{name =
'model'},
          stereotype = s2 : UMLModel::Stereotype{name =
stName}
        };

        enforce domain java javaClass : UMLModel::Class{
          namespace = tarPackage : UMLModel::Package{},
          name = className };
        when{
          PackageToPackage(srcPackage,tarPackage) and
                  (stName <> 'facade');
        }
    }

/*  Transformación de atributos UML a atributos Java con
métodos de acceso     */

      top relation AttributeToJavaAttribute{

        attrName : String;
```

```
        checkonly domain uml umlAttr :
UMLModel::Attribute{
           owner = umlClass : UMLModel::Class{},
           name = attrName
        };

        enforce domain java javaAttr :
UMLModel::Attribute{
          owner = javaClass : UMLModel::Class{
            feature = getter : UMLModel::Operation{
              name = 'get' + attrName.firstToUpper()},
            feature = setter : UMLModel::Operation{
              name = 'set' + attrName.firstToUpper()}
          },
          name = attrName
        };
        when{
          ClassToJavaClass(umlClass, javaClass);
        }
    }

/* Se agregan los identificadores a las clases
persistentes del dominio */

     top relation PersistentClassToJavaIDClass{

       checkonly domain uml umlClass : UMLModel::Class{
         stereotype = s : UMLModel::Stereotype{
           name = 'persistent'
         }
       };

       enforce domain java javaClass : UMLModel::Class{
         feature = attr : UMLModel::Attribute{
           name = umlClass.name.firstToLower() + 'ID',
           type = pdt : UMLModel::PrimitiveDataType{
             namespace = javaClass.namespace,
             name = 'Integer'
           }
         }
       };
       when{
```

```
          ClassToJavaClass(umlClass, javaClass);
        }
    }

/* A partir de las clases UML se generan los DAOs */

     top relation ClassToDAOComponent{

        className, agName : String;

        checkonly domain uml uc : UMLModel::Class{
            namespace = srcPackage : UMLModel::Package{},
            name = className,
            isAbstract = false,
          stereotype = s : UMLModel::Stereotype{name =
'persistent'},
          associationEnd = assoEnd1
:UMLModel::AssociationEnd{
            association = asso : UMLModel::Association{
           associationEnd = assoEnd2
:UMLModel::AssociationEnd{
             aggregation = ak : UMLModel::AggregationKind{
               kind = agName} }
           }
          }

        };

        enforce domain java dc : UMLModel::Class{
          namespace = tarPackage : UMLModel::Package{},
          name = className + 'DAO',
          stereotype = st : UMLModel::Stereotype{
            namespace = tarPackage,
            name = 'interface'}
        };
        when{
          PackageToPackage(srcPackage,tarPackage) and
(assoEnd1 <> assoEnd2) and (agName <> 'composite');
        }
    }
```

```
/* Se generan las operaciones de acceso a datos en los
DAOs */

    top relation DAOComponentToDAOOperations{

        checkonly domain uml class : UMLModel::Class{
        };
        enforce domain java javaClass : UMLModel::Class{
          feature = oper1 : UMLModel::Operation{
            name = 'get' + class.name.firstToUpper() +
'List'
          }
        };
        enforce domain java javaClass : UMLModel::Class{
          feature = oper2 : UMLModel::Operation{
            name = 'get' + class.name.firstToUpper() +
'ByKeyword'
          }
        };
        enforce domain java javaClass : UMLModel::Class{
          feature = oper3 : UMLModel::Operation{
            name = 'update' + class.name.firstToUpper()
          }
        };
        enforce domain java javaClass : UMLModel::Class{
          feature = oper4 : UMLModel::Operation{
            name = 'insert' + class.name.firstToUpper()
          }
        };
      when{
        ClassToDAOComponent(class, javaClass);
       }
    }

/* Transformación de la clase facade a la clase que la
implementa */

    top relation FacadeToImpl{

        className : String;

        checkonly domain uml uc : UMLModel::Class{
```

```
            namespace = srcPackage : UMLModel::Package{},
            name = className,
            stereotype = s : UMLModel::Stereotype{
              name = 'facade'
            }
          };
          enforce domain java implClass : UMLModel::Class{
            namespace = tarPackage : UMLModel::Package{},
            name = className + 'Impl'
          };
          enforce domain java facadeClass : UMLModel::Class{
            namespace = tarPackage : UMLModel::Package{},
            name = className + 'Facade',
            stereotype = s : UMLModel::Stereotype{
              namespace = tarPackage : UMLModel::Package {},
              name = 'interface'
            }
          };
          when{
            PackageToPackage(srcPackage,tarPackage);
          }
      }

}
```

7.3 Resumen

La definición formal de las transformaciones que hemos presentado en este capítulo incluye la especificación de los metamodelos del lenguaje fuente y destino, y la especificación de la transformación entre instancias de estos metamodelos usando el lenguaje QVT. Esta definición formal hace posible la generación automática de los modelos específicos de la plataforma a partir de los modelos abstractos.

La aplicación de MDD resulta productiva cuando el proceso de transformación es complejo o cuando la transformación sin ser compleja involucra una gran cantidad de trabajo rutinario. El ejemplo que desarrollamos en este libro muestra ambas situaciones. Si bien la transformación del modelo UML al modelo relacional, así como la transformación del modelo UML al modelo Java Beans, no son complejas, resultan extensas y repetitivas. Automatizar estas transformaciones nos libera del trabajo monótono que implicaría realizarlas manualmente.

Por otra parte, la generación de los modelos de los controladores y las vistas muestra cierto grado de complejidad ya que involucra modelos dinámicos que describen las interacciones entre los objetos del sistema. Y además la estructura de los modelos origen es muy diferente a la estructura de los modelos destino. Estas transformaciones deben ser creadas por miembros del equipo de desarrollo con mayor conocimiento y experiencia en el dominio del problema y en la plataforma de implementación. De esta forma sus habilidades quedarán capturadas en los modelos y en las transformaciones y podrán ser re-usadas por otros desarrolladores con menor experticia.

CAPÍTULO 8

8. Lenguajes para definir transformaciones modelo a texto

Como hemos señalamos en el Capítulo 6, las transformaciones modelo a modelo crean su modelo destino como una instancia de un metamodelo específico, mientras que el destino de una transformación modelo a texto (M2T) es simplemente un documento en formato de texto.
En este capítulo presentamos la motivación para desarrollar este tipo de lenguajes y enumeramos los requisitos necesarios que los lenguajes de estas características deberían satisfacer. Luego introducimos el lenguaje estándar Mof2Text especificado por el OMG para transformaciones M2T describiendo pequeños ejemplos extraídos y adaptados del documento de especificación [Mof2Text]. A continuación mostramos una transformación M2T definida informalmente que genera código Java desde un modelo Java Beans. Finalmente, utilizando Mof2Text, incluimos la definición formal de dicha transformación y la aplicamos a parte del PSM Java Beans del Sistema Bookstore presentado en el Capítulo 5, obteniendo así el código Java correspondiente. Esta definición será lo suficientemente formal como para permitir su ejecución automatizada, mediante una herramienta que implemente al estándar.

8.1 Características de los lenguajes modelo a texto

Antes de ver la especificación de Mof2Text en detalle, nos ayudará conocer las características de los lenguajes de transformación M2T a través de un análisis de los aspectos que motivan su definición. También enumeraremos los requisitos más importantes que estos los lenguajes deberían satisfacer.

8.1.1 Motivación

No sólo dentro de la propuesta MDD, sino también en áreas tradicionales de la Ingeniería de Software como programación y definición de lenguajes, contar con lenguajes para transformaciones modelo a texto nos permite:

- Elevar el nivel de abstracción: dado que los sistemas se tornan más complejos, en la historia de los lenguajes de programación está comprobado que elevar el nivel de la abstracción tiene gran utilidad (por ejemplo el pasaje de lenguajes Assembler a lenguajes de alto nivel como Cobol).

- Automatizar el proceso de desarrollo de software en sus últimas etapas: en particular decrementar el tiempo de desarrollo, incrementar la calidad del software, poner el foco en la parte creativa del software.

- Generar automáticamente nuevos artefactos desde nuestros modelos. Por ejemplo: obtener código Java, EJB, JSP, C#, o bien Scripts de SQL, texto HTML, casos de test y documentación de modelos.

Existen varias alternativas que podríamos considerar para alcanzar estos objetivos, por ejemplo: lenguajes de programación de propósito general (como Java), lenguajes de Template/scripting (por ejemplo XSLT, Velocity Template Language, Eclipse Java Emitter Templates - JET), lenguajes de transformación M2M (por ejemplo ATL), lenguajes propietarios basados en UML y propuestas basadas en DSLs, entre otras.

Sin embargo, estas alternativas tienen ciertas desventajas: por un lado, los lenguajes de propósito general como los de scripting no permiten definir los metamodelos fuente para transformaciones (lo mismo ocurre para transformaciones M2M, no obstante, en Java esto puede hacerse usando EMF). Por otra parte, los lenguajes para transformación modelo a modelo, como ATL, están basados en metamodelo pero no están diseñados pensando en la generación de texto; podría lograrse, pero en forma más complicada y menos intuitiva a través de la definición del metamodelo de salida. Con respecto a las herramientas para UML, como vimos en el Capítulo 4, la transformación de los modelos a código está implementada de manera fija dentro de la herramienta, lo cual la torna poco adaptable. Por su parte, los DSLs proveen la flexibilidad de las herramientas basadas en metamodelo, pero deben adaptar la generación de código para cada lenguaje específico del dominio.

Estas son algunas de las razones más importantes que justifican la conveniencia de definir lenguajes específicos para soportar transformaciones M2T.

8.1.2 Requisitos

Siguiendo un proceso similar al que precedió a la aparición del estándar para transformaciones modelo a modelo QVT (ver Capítulo 6), el OMG publicó un *Request for Proposal* (RFP) para transformaciones modelo a texto a comienzos de 2004 [OMG RFP 04]. Este RFP definía requisitos específicos que debían satisfacer las propuestas enviadas. Algunos requisitos eran obligatorios y otros opcionales; los listamos a continuación:

Requisitos obligatorios:

- Generación de texto a partir de modelos MOF 2.0
- Re-uso (si es posible) de especificaciones OMG ya existentes, en particular QVT
- Las transformaciones deberán estar definidas en el metanivel del modelo fuente
- Soporte para conversión a String de los datos del modelo
- Manipulación de Strings
- Combinación de los datos del modelo con texto de salida
- Soporte para transformaciones complejas
- Varios modelos MOF de entrada (modelos fuente múltiples)

Requisitos opcionales:

- Ingeniería *round-trip* (de ida y vuelta)
- Detección/protección de cambios manuales, para permitir re-generación
- Rastreo de elementos desde texto, si es necesario para poder soportar los dos últimos requisitos

Una de las propuestas a este RFP, fue el lenguaje MOFScript [Oldevik 06]. El proceso de combinación de MOFScript con las restantes propuestas produjo como resultado al estándar llamado MOF2Text. El objetivo del OMG al diseñar este estándar fue dar soporte a la mayoría de los requisitos listados arriba.
Si bien MOFScript no fue estandarizado, tiene una gran aceptación en la industria debido a la disponibilidad de una herramienta de soporte madura y confiable.

8.2 El estándar Mof2Text para expresar transformaciones modelo a texto

El lenguaje Mof2Text (MOF Model to Text Transformation Language) es el estándar especificado por el OMG para definir transformaciones modelo a texto. Presentaremos sólo una descripción general de sus elementos; en el documento de especificación del OMG [Mof2Text] se puede ver en detalle la descripción de su sintaxis concreta y su sintaxis abstracta. Sus metaclases principales son extensiones de OCL, de QVT y de MOF.

8.2.1 Descripción general del lenguaje

En reiteradas ocasiones durante el desarrollo de este libro, hemos mencionado que las transformaciones son el motor de MDD. De la misma manera que el estándar QVT fue desarrollado para cubrir las necesidades para transformaciones M2M, el estándar Mof2Text fue creado para poder transformar un modelo a diversos artefactos textuales tales como código, especificaciones de implementación, informes, documentos, etc. Esencialmente, este estándar está pensado para transformar un modelo en una representación textual plana. En Mof2Text una transformación es considerada una plantilla (en inglés "template"), donde el texto que se genera desde los modelos se especifica como un conjunto de bloques de texto parametrizados con elementos de modelos. Estos elementos son extraídos de los modelos que van a completar los *placeholders* en la transformación. Los *placeholders* son expresiones especificadas a través de *queries*, mecanismos útiles para seleccionar y extraer valores desde los modelos. Estos valores son luego convertidos en fragmentos de texto usando un lenguaje de expresiones extendido con una librería de manipulación de Strings.
Por ejemplo, la siguiente especificación de transformación genera una definición Java para una clase UML.

```
[template public classToJava(c : Class)]
class [c.name/]
{
       // Constructor
       [c.name/]()
       {
       }
}
[/template]
```

Si aplicamos esta transformación a la clase 'PrintedBook' del sistema Bookstore (figura 8-1), se genera el siguiente texto:

```
class PrintedBook
{
	// Constructor
	PrintedBook ()
	{
	}
}
```

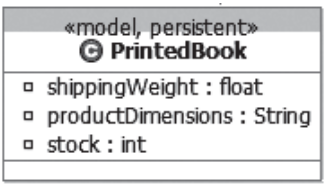

Figura 8-1. Clase 'PrintedBook' del sistema Bookstore

Como puede verse, el texto generado tiene una estructura similar a la especificación, preservando la indentación, los espacios en blanco, etc. Una transformación puede invocar otras transformaciones. La invocación a una transformación es equivalente a colocar *in situ* el texto producido por la transformación invocada.
En el siguiente fragmento de Mof2Text, la transformación *classToJava* invoca a la transformación *attributeToJava* para cada atributo de la clase y agrega ';' como separador entre los fragmentos de texto producidos por cada invocación a *attributeToJava*:

```
[template public classToJava(c : Class)]
class [c.name/]
{
	// Attribute declarations
	[attributeToJava(c.attribute)/]

	// Constructor
	[c.name/]()
	{
```

```
        }
}
[/template]

[template public attributeToJava(a : Attribute)]
[a.type.name/] [a.name/];
[/template]
```

Para la clase 'PrintedBook', se genera el siguiente texto:

```
class PrintedBook
{
        // Attribute declarations
        float shippingWeight;
        String productDimensions;
        int stock;

        // Constructor
        PrintedBook ()
        {
        }
}
```

En vez de definir dos transformaciones separadas, una transformación puede iterar sobre una colección usando el bloque *for*. El uso de este bloque mejora la legibilidad. Por ejemplo la transformación *classToJava* de arriba puede usar el bloque *for* de la siguiente manera:

```
[template public classToJava(c : Class)]
class [c.name/]
{
        // Attribute declarations
        [for(a : Attribute | c.attribute)]
        [a.type.name/] [a.name/];
        [/for]
        // Constructor
        [c.name/]()
        {
        }
}
[/template]
```

De forma similar puede utilizarse el bloque *if* como sentencia condicional, con la condición entre paréntesis. Por ejemplo, dentro de un bloque *for* que itera sobre atributos de una clase UML y los convierte a atributos en una clase C++, si el tipo del atributo es complejo, al nombre del tipo se le agrega un '*':

```
       [for(a : Attribute | c.attribute)]
       [a.type.name/][if(isComplexType (a.type)]*[/
if][a.name/];
       [/for]
```

Una transformación puede tener una guarda o condición escrita en OCL, que indica cuando puede ser invocada. Por ejemplo, la siguiente transformación *classToJava* es invocada solamente si la clase no es abstracta:

```
[template public classToJava(c : Class) ? (c.isAbstract
= false)]
class [c.name/]
{
       // Attribute declarations
       [attributeToJava(c.attribute)/]
       // Constructor
       [c.name/]()
       {
       }
}
[/template]
```

Las navegaciones complejas sobre el modelo pueden ser especificadas a través de *queries*. El siguiente ejemplo muestra el uso de un *query* llamado *allOperations()* que retorna, para una clase dada como parámetro, el conjunto de operaciones de la clase y de todas sus superclases abstractas:

```
[query public allOperations(c: Class) : Set ( Operation
) =
c.operation->union( c.superClass-
>select(sc|sc.isAbstract=true)-
>iterate(ac : Class;
       os:Set(Operation) = Set{}| os-
>union(allOperations(ac)))) /]
```

```
[template public classToJava(c : Class) ? (c.isAbstract
= false)]
class [c.name/]
{
// Attribute declarations
  [attributeToJava(c.attribute)/]
  // Constructor
  [c.name/]()
  {
  }
  [operationToJava(allOperations(c))/]
}

[/template]

[template public operationToJava(o : Operation)]
[o.type.name/] [o.name/] ([for(p:Parameter |
o.parameter)
separator(',')] [p.type/] [p.name/] [/for]);
[/template]
```

El bloque *let* puede utilizarse para chequear si un elemento es de un cierto tipo y en ese caso, declarar una variable de ese tipo, que luego sea usada en la transformación:

```
[template public classToJava(c : Class)]
[let ac : AssociationClass = c ]
class [c.name/]
{
    // Attribute declarations
    [attributeToJava(c.attribute)/]
    // Constructor
    [c.name/]()
    {
    }

    // Association class methods
    [for (t:Type | ac.endType)]
    Attach_[t.name/]([t.name/] p[t.name/])
    {
    // Code for the method here
    }
```

```
    [/for]
}
[/let]
[/template]
```

En esta *transformación*, el bloque *let* verifica si el argumento real es de tipo AssociationClass. Si es así, declara una variable ac de tipo AssociationClass que puede ser usada dentro del bloque *let*. Si la verificación falla, el bloque *let* no produce ningún texto.
Las transformaciones pueden componerse para lograr transformaciones complejas. Pueden construirse transformaciones de mayor tamaño estructuradas en módulos con partes públicas y privadas.
Un *Module* puede tener dependencias 'import' con otros módulos. Esto le permite invocar *Templates* y *Queries* exportados por los módulos importados. Una transformación puede comenzar su ejecución invocando directamente a una transformación pública. No hay noción explícita de transformación *main*. Las transformaciones pueden sobre-escribirse. Un módulo puede extender a otro a través del mecanismo de herencia. El lenguaje solamente soporta herencia simple.

8.2.2 Texto explícito vs. código explícito

En la mayoría de los casos, el estilo de especificación de las transformaciones es intuitivo; la lógica de producción de texto se especifica en una forma delimitada por corchetes ('[' y ']'). Sin embargo, puede haber casos donde esta forma de escritura delimitada por corchetes y anidada, puede tornarse compleja. En este caso es más intuitivo especificar la estructura de la transformación sin usar delimitadores especiales. Esto se logra indicando el modo de escritura, es decir, *text-explicit* o *code-explicit;* text-explicit es el modo por defecto. La sintaxis es similar a las anotaciones Java:

`@text-explicit` o `@code-explicit.` Por ejemplo:

```
@text-explicit
[template public classToJava(c : Class)]
class [c.name/]
{
    // Constructor
    [c.name/]()
    {
```

```
    }
}
[/template]

@code-explicit
template public classToJava(c : Class)
'class 'c.name '
{
    // Constructor
    'c.name'()
    {
    }
}'
/template
```

En la forma code-explicit, en vez de la plantilla, se visualiza el texto de salida. Los bloques se cierran usando una barra seguida de la palabra clave del bloque (por ejemplo, /template).

8.2.3 Traceability (rastreo de elementos desde texto)

La generación de código basado en modelos es una de las principales aplicaciones de las transformaciones M2T. Mof2Text provee soporte para rastrear elementos de modelo desde partes de texto. Un bloque *Trace* relaciona texto producido en un bloque con un conjunto de elementos de modelo provistos como parámetros. El texto a ser rastreado debe estar delimitado por palabras clave especiales. Además, las partes de texto pueden marcarse como protegidas. Tales partes se preservan y no se reemplazan por subsecuentes transformaciones M2T. El bloque *trace* puede incluir información de la transformación original. Las partes de texto deben poder relacionarse sin ambigüedad con un conjunto de elementos y deben tener una identificación única.
En el ejemplo siguiente, el bloque *trace* identifica el texto a ser rastreado relacionando el texto generado con el elemento de modelo 'c' de tipo Class. El bloque protegido identifica la parte de texto que debe preservarse durante subsecuentes transformaciones. Se producen delimitadores en el texto de salida para identificar claramente la parte protegida. Dado que los delimitadores son específicos del lenguaje destino, no están definidos en este estándar. Las herramientas de implementación serán las responsables de producir los delimitadores correctos para el lenguaje destino elegido:

```
[template public classToJava(c : Class)]
[trace(c.id()+ '_definition') ]
class [c.name/]
{
    // Constructor
    [c.name/]()
    {
    [protected('user_code')]
    ; user code
    [/protected]
    }
}
[/trace]
[/template]
```

8.2.4 Archivos de salida

El bloque *file* especifica el archivo donde se envía el texto generado. Este bloque tiene tres parámetros: una URI que denota el nombre del archivo, un booleano que indica cuando el archivo se abre en modo *append* (en este caso el parámetro tiene el valor *true*) o no (valor *false*), y un *id* único y opcional, generalmente derivado desde los identificadores de los elementos del modelo, que también se utilizan para las trazas. Por ejemplo, una herramienta de transformación puede usar el id para buscar un archivo que fue generado en una sesión previa o cuando el nombre del archivo fue modificado por el desarrollador de la transformación. El modo de apertura por defecto es *overwrite*.
El siguiente ejemplo especifica que la clase Java se guarda en un archivo cuya URI se forma con 'file:\\' concatenado con el nombre de la clase seguido por '.java'. El archivo se abre en modo 'overwrite' (2do parámetro en *false*) y se le asigna un id, que es el id de la clase concatenado con 'impl'.

```
[template public classToJava(c : Class)]
[file ('file:\\'+c.name+'.java', false, c.id + 'impl')]
class [c.name/]
{
// Constructor
[c.name/]()
{
}
```

```
}
[/file]
[/template]
```

8.3 Definición de transformaciones modelo a texto

En esta sección presentamos un ejemplo de la definición de transformaciones modelo a texto. En el primer apartado las describiremos informalmente, en lenguaje natural. A continuación, incluimos las definiciones formales en lenguaje Mof2Text, lo cual permite su automatización a través de una herramienta. Finalmente mostramos el código generado a partir de una parte del PSM Java Beans construido en el Capítulo 5.

8.3.1 La transformación del PSM Java Beans a código Java

En esta subsección incluimos las reglas que transforman al modelo Java Beans (descrito en el Capítulo 5), a código Java; consideramos la parte del PSM que modela los conceptos del dominio del sistema Bookstore. En este modelo las clases ya contienen operaciones *setter* y *getter* para sus atributos y para los finales de asociación.
Las reglas transforman clases y asociaciones a código. Al ejemplificar cada regla utilizamos las clases que aparecen en la figura 8-2.

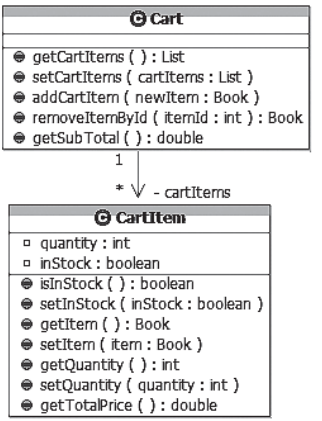

Figura 8-2. Parte del Modelo de dominio Java Beans del sistema Bookstore

Para todas las clases del PSM Java Beans se aplican las siguientes reglas de transformación:

1. Por cada clase, se genera una clase Java con el mismo nombre y se especifica un archivo (*file*) donde se guarda la clase, cuya URI se forma con 'file:\\' concatenado con el nombre de la clase seguido de '.java', en modo 'overwrite' y con identificador el nombre de la clase concatenado con 'Impl' (esto último es debido a que en el modelo del dominio algunas clases no tienen identificador; en general, es conveniente usar el identificador de la clase, en vez del nombre).

   Por ejemplo: La clase Cart del PSM da origen a la clase Cart en Java. También se abre un archivo externo cuya URI es 'file:\\Cart.java', en modo 'overwrite', con identificador 'CartImpl'.

2. Cada atributo y cada operación, estarán en la clase Java destino, manteniendo los tipos de datos correspondientes. En el caso de los atributos, pasan con visibilidad "private". En el caso de las operaciones, se mantiene el tipo de retorno (si la operación no lo especifica, se agrega **void** como tipo de retorno) y los parámetros con sus tipos.

   Por ejemplo: En la clase CartItem, el atributo quantity da origen al atributo quantity en la clase Java, manteniendo el mismo tipo (int) y con visibilidad privada:

   private int quantity;

3. Si la operación es un setter para un atributo, se genera el encabezamiento y cuerpo del método correspondiente en la clase Java. El cuerpo del método está formando por la asignación del parámetro a la variable de instancia, precedida por '**this.**'

   Por ejemplo: En la clase CartItem, la operación setQuantity(**int** quantity) da origen al método Java:

```
public void setQuantity(int quantity) {
      this.quantity = quantity;
}
```

4. Si la operación es un getter se genera el encabezamiento y cuerpo del método correspondiente en la clase Java. Esto es válido sólo

en el caso que el getter sea para recuperar un atributo; si el nombre de la operación comienza con 'get' pero el resto no coincide con el nombre de ningún atributo de la clase dueña de la operación, se la considera como una operación común y se aplica la regla 2.

Por ejemplo: En la clase CartItem, la operación getQuantity():int da origen al método Java:

```
public int getQuantity() {
      return quantity;
      }
```

Por otro lado en la clase CartItem, la operación getTotalPrice(): double, da origen al método Java (que se deriva de su especificación OCL):

```
public double getTotalPrice() {
      return book.getListPrice() * quantity;
}
```

5. Por cada final de asociación navegable con cardinalidad uno, se agrega en la clase Java (*source* de la asociación), un atributo con el mismo nombre y tipo del final de asociación.

   Por ejemplo: la clase CartItem se asocia con la clase Book con cardinalidad uno. Entonces en la clase CartItem generada, se agrega el atributo book de tipo Book:
   private Book book.

6. Por cada final de asociación navegable con cardinalidad "muchos", se agrega un atributo en la clase Java (*source* de la asociación), con el mismo nombre del final de asociación y de tipo List.

   Por ejemplo: La clase Cart se asocia con la clase CartItem con cardinalidad muchos. Entonces en la clase Cart generada, se agrega el atributo cartItems de tipo List.

8.3.2 Definición formal de la transformación usando Mof2Text

Definimos a continuación la transformación en Mof2Text que convierte el PSM Java Beans del dominio descrito en el Capítulo 5 generando

código Java. El PSM está definido usando una variante de UML extendida con estereotipos para crear modelos JB. Por lo tanto, el metamodelo fuente coincide con el metamodelo UML 2.0 simplificado del Capítulo 7 (figura 7.1).

```
[module JavaClass_gen (UMLModel)/]

[query public allClasses(p: Package) : Set ( Class ) =
      p.elements-> select(c| c.oclIsTypeOf (Class)) /]

 [query public attributes(c: Class) : Set ( Attribute )
=
      c.features-> select(a| a.oclIsTypeOf (Attribute))
/]

[query public operations(c: Class) : Set ( Operation ) =
      c.features-> select(o| o.oclIsTypeOf (Operation))
/]

[query public associationEnd (c: Class) : Set (
AssociationEnd ) =
(p.elements-> select (a|a.oclIsTypeOf (Association)
and
a.source.class= c)) -> collect (ae| ae.target) /]

[template public PackageToJava (p : Package)]
{
      // Declaracions de las clases
      [classToJava (allClasses (p))/]

}
 [/template]

[template public classToJava(c : Class)]
[file ('file:\\'+c.name+'.java', false, c.name +
'Impl')]
class [c.name/]
{
      // Declaraciones de Atributos
      [attributeToJava (attributes (c) )/]
      [associationToJava (associationEnd(c))/]
```

```
       // Declaraciones de Operaciones
      [setterToJava (operations (c)))/]
      [getterToJava (operations (c)))/]
      [operationToJava (operations (c)))/]

       // Constructor
       [c.name/]()
       {
       }
}
 [/template]

[template public attributeToJava(a : Attribute)]
private [a.type.name/] [a.name/];
[/template]

[template public associationEndToJava(ae :
AssociationEnd)]
 private [if( ae.multiplicity = '1')][ae.class.name/]
[a.name/];[/if]
         [if ( ae.multiplicity <> '1')] List [a.name/];
[/if] [/template]

[template public gettersToJava(g : Operation) ?
g.name.substring(1,3)= 'get' and (o.owner.attribute->
includes (o.name.substring (4,g.name.size())]
public [g.type.name/] [g.name/] ()
      { return
[(g.name.substring(4,g.name.size()).firstToLower()/])];
      }
[/template]

[template public settersToJava (s : Operation)?
s.name.substring(1,3)= 'set')]
public void [s.type.name/] [s.name/]([
[s.parameter.type/] [s.parameter.name/])

      { this.[(s.name.substring(4,
s.name.size()).firstToLower()/] = [s.parameter.name/]; }
[/template]
```

```
[template public operationToJava(o : Operation) ?
(o.name.substring(1,3) <> 'get' and
o.name.substring(1,3) <> 'set') or
(o.name.substring(1,3) = 'get' and not
(o.owner.attribute-> includes
(o.name.substring(4,o.name.size())]
[if( o.type.name = '')]void [/if]
[if( o.type.name <> '')][o.type.name/] [/if] [o.name/]
([for(p:Parameter | o.parameter)
   separator(',')] [p.type/] [p.name/] [/for])
   {
   // Aquí se genera código para el método
   }
 [/template]

[/file]
[/module]
```

8.3.3 La transformación del PSM a Java aplicada al ejemplo

Como ya dijimos, la figura 8-2 muestra parte del PSM Java Beans generado para las entidades del dominio del sistema Bookstore. La salida producida por la transformación **JavaClass_gen** aplicada a la clase CartItem de este PSM se envía a un archivo con URI: 'file:\\CartItem.java', que se abre en modo 'overwrite' con identificador 'CartItemImpl'. El código Java generado para dicha clase es el siguiente:

```
public class CartItem {

     private Book book;
     private int quantity;
     private boolean inStock;


     public boolean isInStock() {
          return inStock;
     }
     public void setInStock(boolean inStock) {
          this.inStock = inStock;
     }
```

```
public Book getItem() {
     return book;
}

public void setItem(Book book) {
     this.book = book;
}

public int getQuantity() {
     return quantity;
}

public void setQuantity(int quantity) {
     this.quantity = quantity;
}

public double getTotalPrice() {
     return book.getListPrice() * quantity;
}
```

8.4 Conclusiones

En este capítulo hemos presentado la motivación para definir lenguajes de transformación de modelo a texto y sus requisitos. A diferencia de las transformaciones modelo a modelo, estas transformaciones no especifican metamodelo destino, sino que el resultado es simplemente un documento en formato de texto.
Hemos introducido la sintaxis del lenguaje estándar MOF2Text para este tipo de transformaciones. MOF2Text es un lenguaje imperativo que se ajusta y reusa partes de estándares ya existentes; surge como una combinación de las diferentes propuestas presentadas en respuesta al RFP del OMG. En particular, uno de los lenguajes que tuvo mayor influencia en la definición del estándar es MOFScript. Si bien MOFScript no fue estandarizado, logró una gran aceptación en la industria debido a la disponibilidad de una herramienta de soporte madura y confiable. Por su lado, también está emergiendo desde la comunidad Eclipse, la herramienta que da soporte a MOF2Text. En el Capítulo 9 enumeraremos éstas y otras herramientas que dan soporte a lenguajes de transformación. Por último, a modo de ejemplo de uso del lenguaje MOF2Text, hemos presentado la definición de algunas reglas de transformación y las hemos aplicado al sistema Bookstore.

CAPÍTULO 9

9. Herramientas de soporte para la definición de transformaciones de modelos

Existen diversos enfoques que contribuyen brindando soluciones para implementar transformaciones de modelos: lenguajes, frameworks, motores, compiladores, etc. En su mayoría estas soluciones no son compatibles con la especificación del lenguaje estándar para transformaciones QVT. Esto se debe a que la versión 1.0 de QVT fue publicada a principios de 2008 por lo cual las soluciones en base a dicha especificación se encuentran en un estado inicial.
En este capítulo describimos las características de las herramientas más populares que soportan al proceso de definición de transformaciones de modelo a modelo y de modelo a texto.

9.1 Herramientas de transformación de modelo a modelo

En esta sección presentamos algunas de las herramientas más difundidas que permiten implementar transformaciones entre modelos.

9.1.1 ATL

ATL (Atlas Transformation Language) [ATL] consiste en un lenguaje de transformación de modelos y un *toolkit* creados por ATLAS Group (INRIA y LINA) que forma parte del proyecto GMT de Eclipse. En él se presenta un lenguaje híbrido de transformaciones declarativas y operacionales de modelos que si bien es del estilo de QVT no se ajusta a las especificaciones. Aunque la sintaxis de ATL es muy similar a la de QVT, no es

interoperable con este último. Tiene herramientas que realizan una compilación del código fuente a *bytecodes* para una máquina virtual que implementa comandos específicos para la ejecución de transformaciones. ATL forma parte del framework de gestión de modelos AMMA que se encuentra integrado en Eclipse y EMF. ATL posee un algoritmo de ejecución preciso y determinista. El toolkit es de código abierto.

9.1.2 ModelMorf

ModelMorf [ModelMorf] es un motor de transformación desarrollado por Tata Consultancy Servicies. Funciona bajo línea de comandos para ejecutar transformaciones de modelos basados en la especificación de QVT declarativo. Cumple parcialmente con la especificación de QVT del OMG pero es propietaria, no es de código abierto.

9.1.3 Medini QVT

ikv++ technologies ag

Medini QVT [Medini] es una herramienta de código abierto, construida por ikv++ technologies sobre las bases del proyecto OSLO [Oslo]. Open Source Libraries for OCL (OSLO) es un proyecto de la Universidad de Kent que provee una implementación para las clases del metamodelo de OCL. Medini incluye un conjunto de herramientas construidas para el diseño y ejecución de transformaciones declarativas de modelos. Define una estructura de clases que extienden a las librerías de OSLO para modelar transformaciones QVT. Pero además, Medini implementa un motor capaz de ejecutar especificaciones escritas en QVT declarativo.

9.1.4 SmartQVT

SmartQVT [SmartQVT] es un proyecto que nació en el departamento de I+D de France Telecom. Es un compilador de código abierto para transformaciones de modelos escritas en QVT operacional. En esencia, toma como entrada una transformación escrita en lenguaje QVT operacional y obtiene a partir de ella, una clase Java que implementa el comportamiento descrito en el lenguaje QVT. Esta última, como toda clase Java, puede ser integrada en cualquier aplicación y ejecutada desde cualquier máquina virtual. Para realizar su trabajo, la herramienta implementa un parser y un compilador El parser lee código QVT y obtiene a partir de él, su metamodelo. El compilador toma el metamodelo y escribe el texto del código fuente de la clase Java.

9.1.5 Together Architecture/QVT Borland

Together [Together] es un producto de Borland que integra la IDE de Java. Tiene sus orígenes en la integración de JBuilder con la herramienta de modelado UML. Permite a los arquitectos de software trabajar focalizados en el diseño del sistema de manera gráfica; luego la herramienta se encarga de los procesos de análisis, de diseño y de generar modelos específicos de la plataforma a partir de modelos independientes de la plataforma. El proceso de transformación de modelo a modelo se especifica usando QVT operacional y la transformación de modelos a texto se define directamente en Java (con JET). La herramienta provee un editor con resaltador de sintaxis y sugerencias al escribir el código; soporta la sintaxis de OCL y ofrece soporte para debugging y rastreo automático de las transformaciones.

9.1.6 VIATRA 

Viatra [CHM+ 02] (acrónimo inglés de "VIsual Automated model TRAnsformations") es una herramienta para la transformación de modelos que actualmente forma parte del framework VIATRA2, implementado en lenguaje Java y se encuentra integrado en Eclipse. Provee un lenguaje textual para describir modelos y metamodelos, y transformaciones llamados VTML y VTCL respectivamente. La naturaleza del lenguaje es declarativa y está basada en técnicas de descripción de patrones, sin embargo es posible utilizar secciones de código imperativo. Se apoya en métodos formales como la transformación de grafos (GT) y la máquina de estados abstractos (ASM) para ser capaz de manipular modelos y realizar tareas de verificación, validación y seguridad, así como una temprana evaluación de características no funcionales como fiabilidad, disponibilidad y productividad del sistema bajo diseño. Como puntos débiles podemos resaltar que Viatra no se basa en los estándares MOF ni QVT. No obstante, pretende soportarlos en un futuro mediante mecanismos de importación y exportación integrados en el framework.

9.1.7 Tefkat 

Tefkat [LS 05] es un lenguaje de transformación de modelos y un motor para la transformación de modelos. El lenguaje está basado en F-logic y la teoría de programas estratificados de la lógica. Tefkat fue uno de los subproyectos del proyecto Pegamento, desarrollado en Distributed Systems

Technology Centre (DSTC) de Australia. El motor está implementado como un plugin de Eclipse y usa EMF para manejar los modelos basados en MOF, UML2 y esquemas XML. Tefkat define un mapeo entre un conjunto de metamodelos origen en un conjunto de metamodelos destino. Una transformación en Tefkat consiste de reglas, patrones y plantillas. Las reglas contienen un término origen y un término destino. Los patrones son términos origen compuestos agrupados bajo un nombre, y los plantillas son términos destino compuestos agrupados bajo un nombre. Estos elementos están basados en la F-logic y la programación pura de la lógica, sin embargo, la ausencia de símbolos de función significa una reducción importante en la complejidad. Tefkat define un lenguaje propio con una sintaxis concreta parecida a SQL, especialmente diseñado para escribir transformaciones reusables y escalables, usando un conceptos del dominio de alto nivel más que operar directamente con una sintaxis XML. El lenguaje Tefkat está definido en términos de EMOF (v2.0), y está implementado en términos de Ecore. El lenguaje es muy parecido al lenguaje Relations de QVT.

9.1.8 EPSILON

Epsilon [Epsilon] es una plataforma desarrollada como un conjunto de plug-ins (editores, asistentes, pantallas de configuración, etc.) sobre Eclipse. Ofrece un lenguaje llamado Epsilon Object Language (EOL), basado en OCL que puede ser utilizado como lenguaje de gestión de modelos o como infraestructura a extender con nuevos lenguajes específicos de dominio. En la actualidad define tres lenguajes: Epsilon Comparison Language (ECL), Epsilon Merging Language (EML), Epsilon Transformation Language (ETL), para comparación, composición y transformación de modelos respectivamente. Brinda soporte completo al estándar MOF mediante modelos EMF y documentos XML a través de JDOM. Esta herramienta pretende dar soporte al manejo de múltiples modelos. Ofrece constructores de programación convencional (agrupación y secuencia de sentencias), permite la modificación de modelos, provee depuración e informe de errores, etc. Soporta mecanismos de herencia y rastreo asi como también comprobación automática del resultado en composición y transformación de modelos.

9.1.9 AToM3

AToM3 [Atom3] (acrónimo inglés de “A Tool for Multi-formalism and Meta-Modeling”) es una herramienta para modelado en muchos paradigmas. Las

dos tareas principales de AToM3 son metamodelado y transformación de modelos. El metamodelado se refiere a la descripción o modelado de diferentes clases de formalismos usados para modelar sistemas (aunque se enfoca en formalismos de simulación y sistemas dinámicos, las capacidades de AToM3 no se restringen a sólo esos). Transformación de modelos se refiere al proceso automático de convertir, traducir o modificar un modelo en un formalismo dado en otro modelo que puede o no estar descrito en el mismo formalismo. En AToM3 los formalismos y modelos están descriptos como grafos, para realizar una descripción precisa y operativa de las transformaciones entre modelos. Desde una meta especificación (en un formalismo de entidad relación), AToM3 genera una herramienta para manipular visualmente (crear y editar) modelos descriptos en el formalismo especificado. Las transformaciones de modelos son ejecutadas mediante reescritura de grafos. Las transformaciones pueden entonces ser expresadas declarativamente como modelos basados en grafos. Algunos de los metamodelos actualmente disponibles son: Entidad/Relación, GPSS, autómatas finitos determinísticos, autómatas finitos no determinísticos, redes de Petri y diagramas de flujo de datos. Las transformaciones de modelos típicas incluyen la simplificación del modelo, generación de código, generación de simuladores ejecutables basados en la semántica operacional de los formalismos así como transformaciones que preservan el comportamiento entre modelos en diferentes formalismos.

9.1.10 MOLA

El proyecto Mola [Mola] (acrónimo inglés de “MOdel transformation LAnguage”) consiste de un lenguaje de transformación de modelos y de una herramienta para la definición y ejecución de transformaciones. El objetivo del proyecto Mola es proveer un lenguaje gráfico simple y entendible para definir las transformaciones entre modelos. El lenguaje para la definición de la transformación Mola es un lenguaje gráfico, basado en conceptos tradicionales como pattern matching y reglas que definen como se transforman los elementos. El orden en el cual se aplican las reglas es el orden tradicional de los constructores de programación (secuencia, loop y branching). Los procedimientos Mola definen la parte ejecutable de la transformación. La unidad principal ejecutable es la regla que contiene un pattern y acciones. Un procedimiento está construido con reglas usando constructores de la programación estructural tradicional, es decir, loops, branchings y llamadas a procedimientos, todos definidos de una manera gráfica. La parte ejecutable es similar a los diagramas de actividad de UML. Mola usa una manera simple para definir metamodelos: diagramas de clases UML,

los cuales consisten sólo en un subconjunto de los elementos de UML (clases, asociaciones, generalizaciones y enumerativos). Solamente soporta herencia simple. Actualmente el metamodelo completo (el metamodelo fuente y el metamodelo destino) deben estar en el mismo diagrama de clases. Adicionalmente se le agregan asociaciones para mapear los elementos del metamodelo fuente en el destino.

9.1.11 Kermeta

El lenguaje Kermeta [Kermeta] fue desarrollado por un equipo de investigación de IRISA (investigadores de INRIA, CNRS, INSA y la Universidad Rennes). El nombre Kermeta es una abreviación para “Kernel Metamodeling” y refleja el hecho que el lenguaje fue pensado como una parte fundamental para el metamodelado. La herramienta que ejecuta las transformaciones Kermeta está desarrollada en Eclipse, bajo licencia EPL. Kermeta es un lenguaje de modelado y de programación. Su metamodelo conforma el estándar EMOF. Fue diseñado para escribir modelos, para escribir transformaciones entre modelos y para escribir restricciones sobre estos modelos y ejecutarlos. El objetivo de esta propuesta es brindar un nivel de abstracción adicional sobre el nivel de objetos, y de esta manera ver un sistema dado como un conjunto de conceptos (e instancias de conceptos) que forman un todo coherente que se puede llamar modelo. Kermeta ofrece todos los conceptos de EMOF usados para la especificación de un modelo. Un concepto real de modelo, más precisamente de tipo de modelo, y una sintaxis concreta que encaja bien con la notación de modelo y metamodelo. Kermeta ofrece dos posibilidades para escribir un metamodelo: escribir el metamodelo con Omondo, e importarlo o escribir el metamodelo en Kermeta y traducirlo a un archivo ecore usando la función “kermeta2ecore”.

9.2 Herramientas de transformación de modelo a texto

En esta sección describimos las principales herramientas que facilitan la transformación de modelos a texto.

9.2.1 M2T la herramienta para MOF2Text

El proyecto M2T de la comunidad Eclipse está trabajando en la implementación de una herramienta de código abierto que soporte al

lenguaje estándar MOF2Text [MOF2Text]. Esta implementación se realizará en dos etapas. En la primera etapa, prevista para 2010, se entregará una versión compatible con las características básicas (el 'core') del lenguaje Mof2Text, mientras que la segunda versión soportará también sus características avanzadas.
Los componentes planificados para esta herramienta son los siguientes:

- Componentes autónomos (basados en EMF)
  - Motor: el motor generará texto basado en el API reflexivo de EMF.
  - Parser: el compilador creará una representación EMF del template.
- Componentes del IDE Eclipse
  - Eclipse Builder con detección y marcado de errores.
  - Editor con ayudas para la redacción (ej. auto-completado y resaltado).
  - Debugger

9.2.2 MOFScript

La herramienta MOFScript [Oldevik 06] permite la transformación de cualquier modelo MOF a texto. Por ejemplo, permite la generación de código Java, EJB, JSP, C#, SQL Scripts, HTML o documentación a partir de los modelos. La herramienta está desarrollada como un plugin de Eclipse, el cual soporta el parseo, chequeo y ejecución de scripts escritos en MOFScript. MOFScript está basado en QVT, es un refinamiento del lenguaje operacional de QVT. Es un lenguaje textual, basado en objetos y usa OCL para la navegación de los elementos del metamodelo de entrada. Además, presenta algunas características avanzadas, como la jerarquía de transformaciones y mecanismos de rastreo.
Algunas características de MOFScript son:

- El lenguaje permite especificar mecanismos de control básicos como loops y sentencias condicionales.
- El lenguaje permite manipulación de strings con operaciones básicas.
- La herramienta puede generar texto a partir de cualquier modelo basado en un metamodelo MOF, como por ejemplo, modelos UML.
- La herramienta permite especificar el archivo salida para la generación del texto.
- La herramienta mantiene las trazas entre los modelos y el texto generado.
- El editor de scripts posee ayuda para completar el código.

- Define reglas concretas y abstractas y permite la re definición de reglas.
- La ingeniería inversa todavía no es parte de la herramienta.

9.3 Conclusiones

Actualmente hay una gran variedad de herramientas que soportan transformaciones tanto de modelo a modelo como de modelo a texto. De las soluciones referidas, sólo un subconjunto ofrece compatibilidad con los estándares del OMG. La tabla 9-1 muestra, considerando los lenguajes presentados anteriormente, cuáles de ellos usan lenguajes estándares para la definición de los metamodelos (es decir, son compatibles con MOF), y para la definición de las transformaciones (es decir, se basan en QVT o bien implementan alguno de sus niveles).

Tabla 9-1: *Compatibilidad de las herramientas de transformación*

Herramienta	Lenguaje de metamodelado	Lenguaje de transformación
ATL	Compatible con MOF	un híbrido parecido a QVT
ModelMorf	Compatible con MOF	
Medini QVT	Compatible con MOF	QVT (sólo el nivel declarativo)
SmartQVT	Compatible con MOF	QVT (sólo el nivel operacional)
Together QVT	Compatible con MOF	QVT (sólo el nivel operacional)
VIATRA		
TefKat	Compatible con MOF	
Epsilon		
AToM3		
MOLA	Compatible con MOF	
Kermeta	Compatible con MOF	
M2T	Compatible con MOF	basado en QVT operacional
MofScript	Compatible con MOF	basado en QVT operacional

Las herramientas que utilizan modelos compatibles con MOF presentan diferentes maneras para definir estos metamodelos, desde la escritura manual de un archivo en formato XMI, pasando por la definición de un nuevo lenguaje (como es el caso de ATL y el lenguaje km3) o por la importación de diagramas de clases UML o EMF desde otras herramientas. En general, estas herramientas todavía se encuentran en un estado inmaduro, varias de sus funcionalidades requeridas no han sido implementadas y sufren de errores de programación aún no corregidos.

CAPÍTULO 10

Por Diego García y Fernando Palacios

10. Pruebas dirigidas por modelos

En este capítulo presentaremos una introducción a la verificación y validación del software poniendo énfasis en el testing (pruebas) dirigido por modelos. Inicialmente explicaremos los distintos tipos de pruebas que se realizan para garantizar la calidad del software. A continuación mostraremos un perfil de pruebas estandarizado que se utiliza para especificar los modelos de prueba. En las últimas secciones, describiremos el framework JUnit para la elaboración de pruebas y definiremos las transformaciones que deben aplicarse al modelo de prueba para generar los tests JUnit y tambien veremos como el framework EasyMock brinda soporte a la generación de métodos con comportamiento. Finalmente, mencionaremos algunas propuestas para la validación de las transformaciones de modelos en sí mismas.

10.1 Verificación y validación

Los términos verificación y validación (V&V) se refieren a los procesos de comprobación y análisis que aseguran que el software funcione de manera acorde con su especificación y cumpla las necesidades de los clientes. Verificación y validación son dos conceptos distintos. La verificación determina si un producto de software de una fase cumple los requisitos de la fase anterior. La validación determina si el software satisface los requisitos del usuario. Sobre estas definiciones podemos afirmar que:

- La validación sólo se puede hacer con la activa participación del usuario.

- La validación establece si estamos haciendo el software correcto.
- La verificación establece si estamos haciendo el software correctamente.

Hay distintas técnicas de V&V, entre ellas la verificación formal de programas y modelos, chequeo de modelos (model checking) y testing.
La verificación formal es un método de verificación estática (se realiza mediante un análisis del propio código del programa, a partir de una abstracción o de una representación simbólica), en el que partiendo de un conjunto axiomático, reglas de inferencia y algún lenguaje lógico (como la lógica de primer orden), es posible encontrar una demostración o prueba de corrección de un programa. Entre las herramientas para la verificación formal de programas podemos citar a ESC/Java (Extended Static Checker for Java) [ESC 2], desarrollada por el Compaq Systems Research Center para encontrar errores en tiempo de ejecución de programas Java por análisis estático del texto del programa. Las recientes versiones de ESC/Java están basadas en Java Modeling Language (JML)[JML 2].
El model checking es un método automático de verificación de un sistema formal. Éste es descrito mediante un modelo, que debe satisfacer una especificación formal definida mediante una fórmula, a menudo escrita en alguna variedad de lógica temporal. Entre las herramientas de model checking se encuentra Java PathFinder [JPF 2], empleada para el bytecode ejecutable de Java.
Las técnicas de V&V comentadas anteriormente son bastante complicadas y requieren tener experiencia en métodos formales, en cambio el testing es una alternativa práctica. Esta técnica consiste en el proceso de ejecutar repetidamente un producto para corroborar que satisface los requisitos e identificar diferencias entre el comportamiento real y el comportamiento esperado (IEEE Standard for Software Test Documentation, 1983). De acuerdo con la definición anterior, el testing juega un rol importante para asegurar la calidad del software. Aunque cabe señalar que esta técnica no puede garantizar la ausencia de errores, sólo puede detectar su presencia [DDH 72].
El éxito del testing depende fuertemente de nuestra capacidad para crear pruebas útiles para descubrir y eliminar problemas en todas las etapas del desarrollo [Bei 90]; esto requiere diseñarlas sistemáticamente desde la fase de análisis. Los casos de prueba determinan el tipo y alcance de la prueba. Dada la complejidad del proceso, resulta indispensable su automatización (al menos parcial).
En el contexto de MDD, el Model-Driven Testing (MDT) [UL 07] es una de las propuestas más recientes que enfrentan este problema. El MDT es

una forma de prueba de caja negra [Bei 95] que utiliza modelos estructurales y de comportamiento, descritos por ejemplo con el lenguaje estándar UML, para automatizar el proceso de generación de casos de prueba.
MDT reduce el costo de mantenimiento de los tests, ya que cuando se produce un cambio en el modelo, los desarrolladores sólo tienen que regenerar (automáticamente) los tests para reflejar dichos cambios. Las herramientas de MDT refuerzan la comunicación del equipo porque los modelos proporcionan una vista clara y unificada del sistema y de las pruebas.

10.2 Pruebas de software

Las pruebas de software son los procesos que permiten identificar posibles fallos de implementación. Dichas pruebas se integran durante todo el proceso de desarrollo de software cubriendo los distintos aspectos de la aplicación. Para determinar el nivel de calidad del software deben efectuarse una serie de medidas o pruebas [Art Testing] que permitan comprobar el grado de cumplimiento respecto de las especificaciones iniciales del sistema.
Esta sección describe los distintos tipos de prueba que cubren todos los aspectos del software que se deben probar para asegurar la calidad del mismo.

- **Prueba de unidad:** su objetivo es probar y asegurar el correcto funcionamiento de un módulo de código (como por ejemplo clases, módulos, subsistemas). Estas pruebas son creadas por los desarrolladores en su trabajo diario; se requiere tener conocimiento detallado acerca de la estructura interna del diseño del programa. La idea es escribir casos de prueba para cada función no trivial o método en el módulo de forma que cada caso sea independiente del resto.

- **Prueba de integración:** la prueba de integración se realizan una vez que se han aprobado las pruebas unitarias, y asegura la correcta interacción, compatibilidad y funcionalidad de las distintas partes que componen el sistema.

- **Prueba de validación:** es el proceso de revisión que el sistema de software producido cumple con las especificaciones y que cumple su cometido, además de comprobar que lo que se ha especificado es lo que el usuario realmente quería. Estas pruebas se sub-clasifican de la siguiente forma:

- Prueba de aceptación: el objetivo de la prueba de aceptación es verificar si el sistema tiene sentido desde el punto de vista del cliente y cumple con sus requisitos. Esta prueba la realiza el usuario o cliente con el fin de determinar si acepta la aplicación.Pruebas alfa: realizadas por el usuario con el desarrollador como observador en un entorno controlado.
- Pruebas beta: realizadas por el usuario en su entorno de trabajo y sin observadores.
- Prueba funcional: se basa en la ejecución, revisión y retroalimentación de las funcionalidades previamente diseñadas para el software. Las pruebas funcionales se hacen mediante el diseño de modelos de prueba que buscan evaluar cada una de las opciones con las que cuenta el paquete informático.

- **Caja Blanca:** se realiza sobre las funciones internas de un módulo. Las pruebas de caja blanca se llevan a cabo en primer lugar, sobre un módulo concreto, para luego realizar las de caja negra sobre varios subsistemas (integración).

- **Caja Negra:** esta prueba verifica el software a partir de las entradas que recibe y las salidas o respuestas que produce, sin tener en cuenta su funcionamiento interno. Se interesa en qué es lo que hace, pero sin dar importancia a cómo lo hace.

- **Prueba no funcional:** esta prueba valida los requisitos no funcionales para la arquitectura designada, como seguridad, robustez, instalación o como el sistema se recupera de las caídas, fallas del hardware, u otros problemas.

Debajo mostramos una clasificación de los principales tipos de pruebas funcionales:

- **Prueba de carga:** se ocupa de verificar las métricas de rendimiento (tales como tiempo de respuesta y flujo de datos) y escalabilidad del sistema en la carga. Determina hasta donde puede soportar el sistema determinadas condiciones extremas.

- **Prueba GUI o Usabilidad:** prueba que la interfaz de usuario (GUI) sea intuitiva, amigable, accesible y funcione correctamente. Determina la calidad de la experiencia de un usuario en la forma en la que se interactúa con el sistema.

- **Prueba de regresión:** prueba la aplicación en conjunto, debido a cambios (adición o modificación) producidos en cualquier módulo o funcionalidad del sistema y puede implicar la re-ejecución de otros tipos de prueba. Generalmente, las pruebas de regresión se llevan a cabo durante cada interacción, ejecutando otra vez las pruebas de la iteración anterior con el objetivo de asegurar que los defectos identificados en la ejecución anterior de la prueba se han corregido y que los cambios realizados no han introducido nuevos defectos o reintroducido defectos anteriores.

10.3 El lenguaje U2TP

Desde comienzos de 2004 el OMG cuenta con un lenguaje estándar oficial para diseñar, visualizar, especificar, analizar, construir y documentar los artefactos de los sistemas de prueba. Este lenguaje se implementó utilizando el mecanismo de perfiles de UML 2.0 y recibió el nombre de UML 2.0 Testing Profile (U2TP) [U2TP 04]. Este perfil proporciona un marco formal para la definición de un modelo de prueba adhiriendo al enfoque de caja negra. Aplicando este enfoque las pruebas se derivan de la especificación del artefacto que se somete a la verificación, donde éste es una "caja negra" cuyo comportamiento sólo se puede determinar estudiando sus entradas y salidas.
Al ser un perfil basado en UML hereda las mismas características:

- División de capas, estableciendo una arquitectura de metamodelo de 4 niveles.
- Particionamiento al utilizar paquetes para refinar las estructuras.
- Extensibilidad que permite adaptar los perfiles de pruebas a plataformas tecnológicas (por ejemplo, J2EE/EJB, .NET/COM+) y dominios específicos (por ejemplo, finanzas, telecomunicaciones).

El perfil se organiza en cuatro grupos lógicos que cubren los aspectos de pruebas y los diagramas que representan las vistas del sistema estructural y de comportamiento. Los diagramas estructurales permiten describir la arquitectura (en inglés test architecture) y los datos de la prueba (en inglés test data). La arquitectura define los conceptos relacionados con la estructura y configuración de las pruebas, es decir los elementos (y sus relaciones) que participan en esta. Los datos proporcionan las estructuras y la semántica de los valores procesados. Los diagramas de comportamiento (en inglés test behavior) permiten expresan conceptos relacionados con los aspectos dinámicos. Estos

diagramas describen la manera en que colaboran grupos de objetos para un determinado caso de prueba. También pueden mostrar restricciones de tiempo (en inglés test time) de manera precisa y completa.

10.3.1 Tipos de prueba en U2TP

El perfil U2TP soporta la especificación de diversos tipos de prueba. En las siguientes sub secciones describiremos los principales conceptos (figura 10-1) [MDT-Dai].

Prueba de arquitectura
La prueba de arquitectura define un conjunto de conceptos agregados a los estructurales de UML 2.0 que permiten especificar los aspectos estructurales de un contexto de pruebas cubriendo los componentes y el sistema bajo pruebas, su configuración, los elementos y sus relaciones establecidas en una prueba. Uno o más objetos pueden identificarse como Sistema Bajo Pruebas (en inglés System Under Test, SUT). Los Componentes de Pruebas (en inglés Test Component) son objetos dentro de un sistema de pruebas que pueden comunicarse con el SUT u otros componentes para verificar el comportamiento de las pruebas. El Contexto de Pruebas (en inglés Test Context) permite a los usuarios agrupar los casos de pruebas (en inglés test cases), y describir su correspondiente Configuración (en inglés Test Configuration) por ejemplo para establecer la conexión entre los componentes de pruebas y el SUT. Además define el Control de Pruebas (en inglés Test Control) que posibilita definir el orden de ejecución de los casos de pruebas. El Árbitro (en inglés Arbiter) evalúa el resultado del veredicto final de un contexto de prueba. Los valores definidos son Pass (Pasa), Fail (Falla), Inconclusive (Incierto) y Error en ese orden, aunque se pueden redefinir implementando la interface Arbiter. El Scheduler controla la ejecución de la prueba y los componentes de pruebas; mantiene la información acerca de qué componentes existen en todo momento y es responsable de la creación de los componentes de pruebas, como así también de que inicien y terminen en forma sincronizada.

Prueba de comportamiento
Adicionalmente a los conceptos de comportamiento descriptos en UML 2.0, el perfil de pruebas agrega nuevos conceptos para especificar el comportamiento de las pruebas, sus objetivos y la evaluación de los SUT. El objetivo de prueba (en inglés Test Objective) es un elemento

que describe lo que se debería probar y está asociado al caso de prueba (en inglés Test Case). Los diagramas UML de comportamiento, como por ejemplo el diagrama de máquinas de estado o secuencia, pueden utilizarlo para definir un estímulo de prueba (en inglés Test Stimuli), observaciones, controles, invocaciones, coordinación y acciones de pruebas. La prueba de comportamiento se especifica en un caso de prueba, que es una operación del contexto de prueba especificando cómo el conjunto de componentes interactúan con el SUT para la prueba. La acción de validación (en inglés Validation Action) se ejecuta por un componente de prueba local que informa al Arbiter sobre el veredicto final. El veredicto (en inglés Verdict) de prueba muestra el resultado de la ejecución de la prueba, los posibles resultados son: pasa (en inglés pass), incierto (en inglés inconclusive), falla (en inglés fail) y error.

Prueba de datos

Las pruebas de datos utilizan símbolos especiales (en inglés wildcards) para manejar eventos inesperados, o eventos con distintos valores. Existen tres tipos de estos elementos: los que se refieren a cualquier valor, los que especifican un valor o ninguno y aquellos para los que se omiten los valores. Los Pool de Datos (en inglés Data Pool) se utilizan con los contextos o componente de prueba. Los selectores de datos (en inglés Data Selectors) son operaciones para obtener datos de las pruebas desde los data pool o particiones de datos (en inglés data partitions). Las reglas de codificación (en inglés coding rules) permiten definir la codificación y decodificación de los datos de pruebas cuando se comunica con el SUT.

Prueba de tiempo

Los conceptos de tiempo son esenciales para proveer una descripción precisa y completa al especificar casos de pruebas. Los conceptos de tiempo ya descriptos por UML 2.0 no cubren todos los requisitos para especificarlos, por lo tanto el perfil de pruebas agrega un conjunto adicional de conceptos de tiempo que ayudan a describir restricciones y observaciones de tiempo, además de cronómetros (en inglés timers) para cuantificar la ejecución de las pruebas. Los cronómetros son necesarios para manipular y controlar el comportamiento de las pruebas además de utilizarse para asegurar la finalización de los casos de pruebas. Los husos horarios (en inglés time zones) agrupan y sincronizan los componentes de pruebas dentro de un sistema distribuido donde cada uno de ellos pertenece a determinado grupo de tiempo permitiendo comparar los eventos de tiempo dentro del mismo huso horario.

Tabla 10-1: *Conceptos de la arquitectura U2TP*

Prueba de arquitectura **(test architecture)**	Prueba de comportamiento **(test behavior)**	Prueba de datos **(test data)**	Prueba de tiempo **(test time)**
SUT	Test objective	Wildcards	Timer
Test component	Test case	Data pool	Time zone
Test context	Defaults	Data partition	
Test configuration	Validation action	Data selector	
Test control	Verdicts	Coding rules	
Arbiter			
Scheduler			

10.3.2 Conceptos básicos

La figura 10-1 muestra la especificación de la clase Cart que hemos presentado en los capítulos anteriores. Utilizaremos esta clase para generar nuestra primera prueba de unidad empleando el perfil de pruebas.

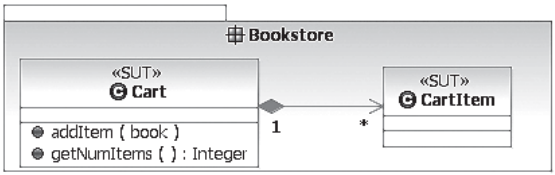

Figura 10-1. Especificación de la clase Cart

Los objetivos de la prueba consisten en verificar las siguientes condiciones:

- Cuando se crea el objeto Cart no contiene elementos.
- Cuando se agrega un ítem al carrito vacío éste contiene exactamente un ítem.

El diagrama en la figura 10-2 ilustra al paquete Bookstore y BookstoreTest, los cuales agrupan las clases del sistema y de las pruebas respectivamente. El sistema se limita a clases definidas en el paquete Bookstore. La dependencia indica que hay clases del paquete BookstoreTest que dependen de clases del paquete Bookstore. Por lo tanto, esta dependencia explicita el hecho de que si se altera la interfaz del paquete de Bookstore, se debe modificar también el paquete BookstoreTest.

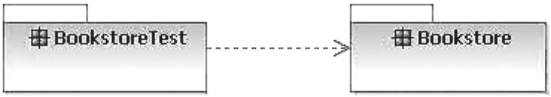

Figura 10-2. Paquete BookstoreTest

El diagrama de clases aplicado en la disciplina de testing describe las clases de objetos que participan de la prueba y las relaciones que existen entre ellos. Para lograr esta adaptación el perfil provee varios estereotipos que generalmente se aplican a operaciones, clases e interfaces UML. La figura 10-3 muestra un diagrama de clases que representa la arquitectura inicial del modelo. La clase CartTest está en el paquete BookstoreTest y la clase Cart es una clase del paquete Bookstore. En este diagrama aparecen dos nuevos estereotipos aplicados a clases e interfaces UML, ellos son <<TestContext>> y <<SUT>>.

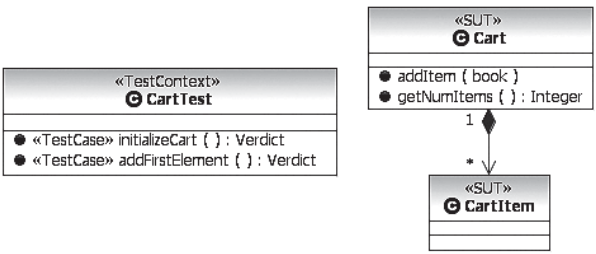

Figura 10-3. Arquitectura inicial del modelo de prueba

El estereotipo <<TestContext>> (Contexto de Prueba) puede agrupar un conjunto de casos de prueba (en inglés Test Cases). En el ejemplo está aplicado a la clase CartTest, la cual contiene dos casos de prueba.

El estereotipo <<SUT>> es un acrónimo inglés de System Under Test (Sistema Bajo Prueba), y puede consistir de varios objetos. En el ejemplo está aplicado a la case Cart, y se utiliza para representar al sistema, subsistema, o componente que se somete a la prueba.
Los diagramas de clases también muestran los casos de prueba efectuados sobre la aplicación (SUT). En el ejemplo la clase CartTest tiene dos operaciones con el estereotipo <<TestCase>>, indicando los casos de prueba que se deben verificar. Estos tienen acceso a todos los elementos del contexto de la prueba, incluso a los elementos del SUT. Estas operaciones retornan un veredicto (Verdict), los posibles resultados son:

- Pasa (en inglés pass): el sistema bajo la prueba cumple las expectativas.
- Incierto (en inglés inconclusive): no fue posible determinar un resultado.
- Falla (en inglés fail): diferencia entre los resultados esperados y reales.
- Error: un error en el ambiente de prueba.

Los diagramas de comportamiento definen conceptos relacionados con los aspectos dinámicos de la prueba. Estos tipos de diagramas especifican las acciones y evaluaciones necesarias para chequear el objetivo de la prueba, el cual describe lo que se debe probar.
El primer objetivo de nuestra prueba es comprobar que cuando se crea el objeto Cart está vacío (figura 10-4).

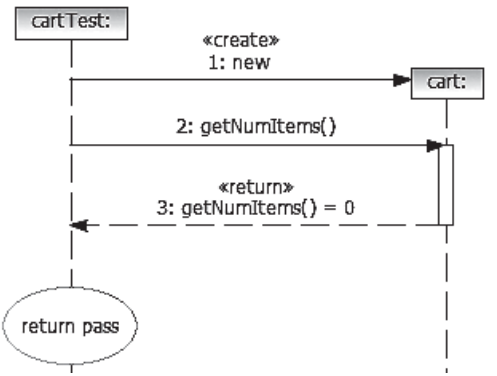

Figura 10-4. Diagrama de secuencia de la prueba de unidad de initializeCart

El segundo objetivo de la prueba es comprobar que cuando se agrega un ítem al carrito este contiene exactamente un ítem (figura 10-5).
.

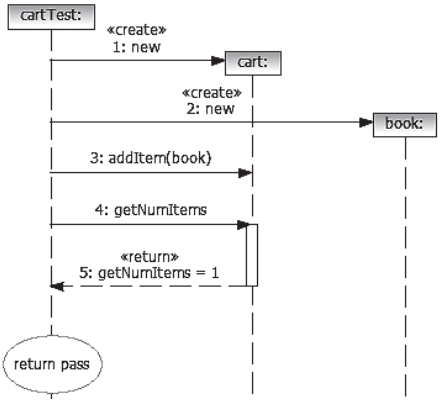

Figura 10-5. Diagrama de secuencia de la prueba de unidad de addFirstElement

10.3.3 Conceptos avanzados

En esta sección presentaremos algunos conceptos avanzados del perfil de pruebas (U2TP), aplicados en otro ejemplo que presentamos a continuación. El objetivo de la prueba es verificar que el sistema recupera el detalle del libro seleccionado y lo muestra en la página de detalle. Esta prueba asumirá que los libros se recuperan correctamente de la base de datos, emulando su acceso. La figura 10-6 ilustra el diagrama de secuencia para el comportamiento "exhibir los detalles del Libro". El controlador ViewBookController es el foco de nuestra atención.

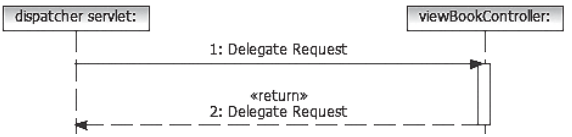

Figura 10-6. Diagrama de secuencia "exhibir los detalles del Libro".

La figura 10-7 muestra las partes públicas de los paquetes involucrados en este ejemplo. El paquete Controllers importa el paquete DAOs, donde se especifican las interfaces de acceso a datos, y el paquete Domain Model donde se especifican los elementos del modelo. La interface BookSpecDao define el protocolo de acceso a los datos.

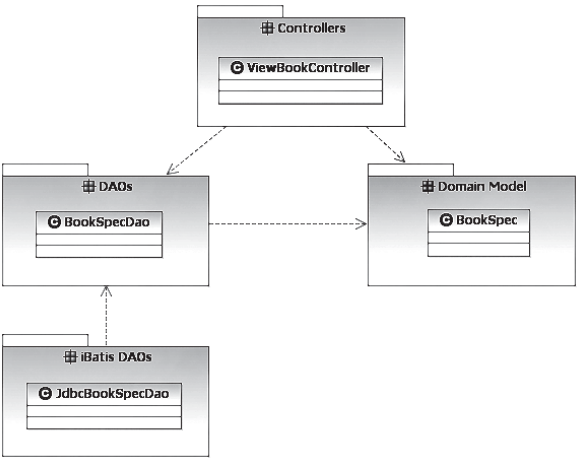

Figura 10-7. Elemento del sistema a ser probado

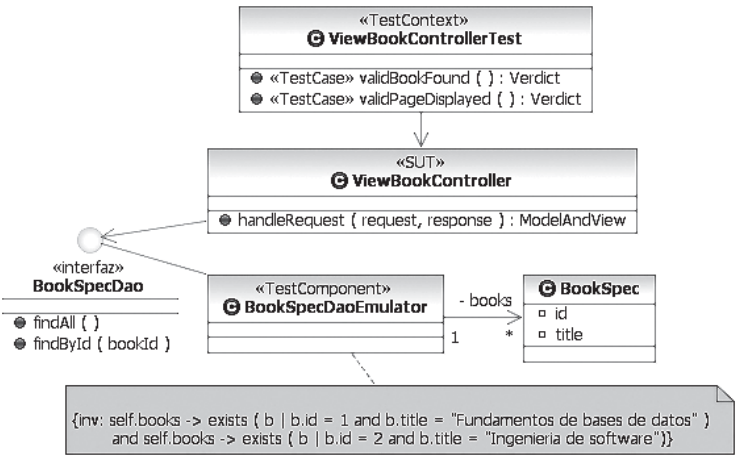

Figura 10-8. El paquete BookstoreTest para la prueba de visualización del detalle de libro

El paquete BookstoreTest ilustrado en la figura 10-8 contiene todos los elementos necesarios para especificar nuestras pruebas; importa elementos del paquete Bookstore para acceder a los elementos necesarios del sistema. El paquete BookstoreTest consiste de un contexto de prueba denominado ViewBookControllerTest, el cual tiene dos casos de prueba: validBookFound y validPageDisplayed, los cuales representan los dos pasos básicos del curso normal del caso de uso respectivo. El primero de ellos verifica que se recupera el detalle del libro seleccionado, mientras que el segundo comprueba que se muestra la página de detalle. La clase BookSpecDaoEmulator del paquete de prueba tiene el estereotipo <<TestComponent>>. Este estereotipo se utiliza para identificar las clases del sistema de prueba. Estas clases pueden comunicarse con otras clases componentes o con el SUT con el objetivo de especificar los casos de prueba. Volviendo al ejemplo, la clase BookSpec DaoEmulator implementa la interfaz BookSpecDao, y se utiliza para emular el acceso de los datos. La figura 10-8 muestra una restricción OCL, utilizada para especificar las condiciones que deben cumplir los libros contenidos por el emulador. Estas restricciones se deben satisfacer para garantizar la validez del resultado de la prueba.

Figura 10-9. Estructura del contexto de la prueba

La figura 10-9 muestra la configuración de la prueba, que especifica como el SUT y los componentes de prueba son utilizados para este contexto en particular. Se pueden especificar posibles formas de comunicación mediante puertos y conectores. Cada configuración debe consistir de al menos un SUT. La estructura de ViewBookControllerTest es muy simple, consiste de un SUT y un componente. El SUT es tipado por la clase ViewBookController del paquete Bookstore, y se conecta con el dao emulado.

Para llevar a cabo las pruebas vamos a utilizar el método público handleRequest(request, response), definido en la clase AbstractController de Spring. Para ello debemos emular las interfaces HttpServletRequest y HttpServletResponse. Por lo tanto, tenemos dos nuevos componentes en nuestro modelo de prueba.

El primer objetivo de la prueba es comprobar que el sistema recupera el detalle del libro seleccionado (figura 10-10).

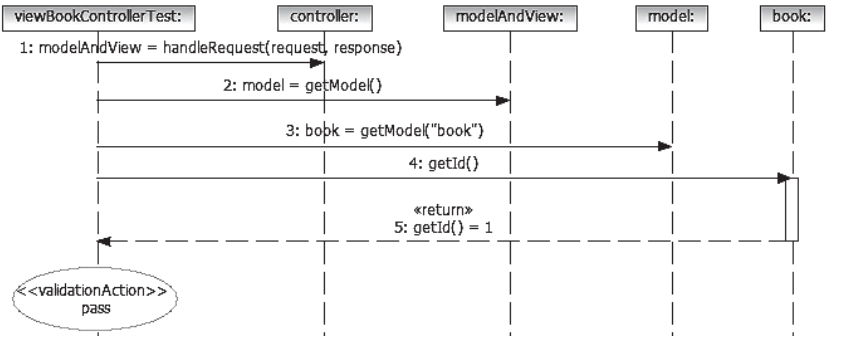

Figura 10-10. Diagrama de secuencia de la prueba de unidad de validDetailsFound

El segundo objetivo de la prueba es comprobar que el sistema muestra el libro seleccionado en la página de detalle (figura 10-11).

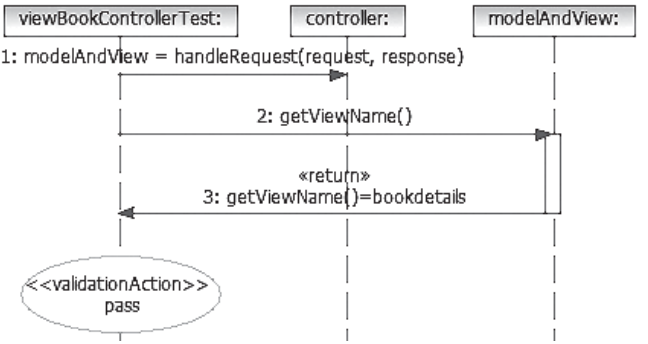

Figura 10-11. Diagrama de secuencia de la prueba de unidad de validPageDisplayed

Consideremos ahora un curso alternativo: ¿Qué sucede si el identificador del libro dado no existe? En tal caso el sistema debería desplegar una página que indique que el libro no fue encontrado. El objetivo de la prueba es verificar que si el identificador no existe el sistema no recupera el detalle del libro seleccionado y muestra la página de detalle que indica que el libro no fue encontrado (figura 10-12). En esta prueba también asumiremos que los libros se recuperan correctamente de la base de datos, emulando su acceso. En este caso vamos a definir un nuevo contexto de prueba para especificar este curso alternativo, ilustrada en la figura 10-13.

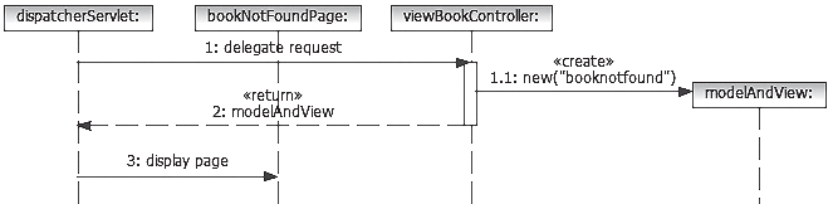

Figura 10-12. Diagrama de secuencia del curso alternativo de exhibir un libro

Figura 10-13. Especificación de la clase ViewBookNotFoundControllerTest

Los casos de prueba del curso alternativo son similares a los del curso normal (ViewBookControllerTest). La única diferencia es que este último utiliza un identificador válido (1), mientras que para documentar la nueva prueba necesitamos un identificador que no utilizamos (por ejemplo, -1). Para el diseño de los casos de prueba se utiliza el patrón template [GHJV 94] (figura 10-14). La clase abstracta ViewBookControllerTest define los dos casos de prueba (validBookDetails y validPageDisplayed) y tres métodos abstractos. Los métodos verifyBookDetails y verifyPageDisplayed verifican las condiciones de los casos de prueba.

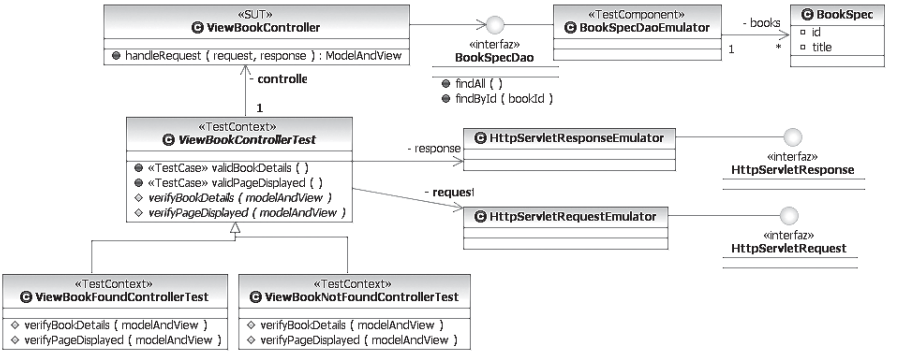

Figura 10-14. Jerarquía de ViewBookControllerTest

Las figuras 10-15 y 10-16 ilustran los diagramas de secuencias para verificar que se muestre la página correspondiente.

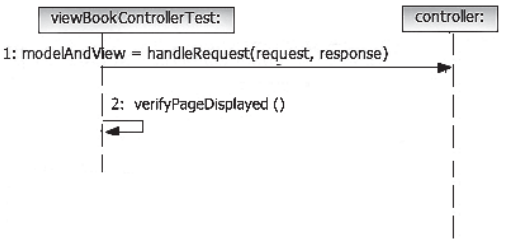

Figura 10-15. Diagrama de secuencia del caso de prueba validPageDisplayed

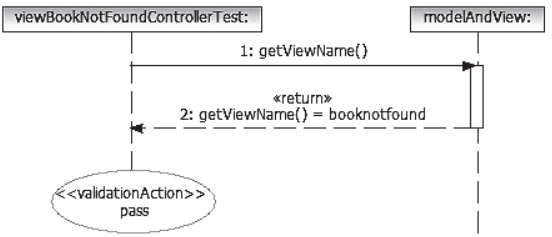

Figura 10-16. Diagrama de secuencia del curso alternativo de verifyPageDisplayed

10.4 JUnit y EasyMock

En los últimos años se han desarrollado un conjunto de frameworks denominados XUnit que facilitan la elaboración de pruebas unitarias en diferentes lenguajes. La familia XUnit se compone, entre otros, de las siguientes herramientas:

- SUnit (sunit.sourceforge.net) para Smalltalk.
- JUnit (www.junit.org), TestNG (testng.org), JTest (www.parasoft.com), JExample (www.ohloh.net/p/jexample) son entornos de pruebas Java.
- CPPUnit (sourceforge.net/apps/mediawiki/cppunit) para C++.
- NUnit (www.nunit.org) para la plataforma .NET y Mono.
- Dunit (dunit.sourceforge.net) para Borland Delphi.
- PHPUnit (www.phpunit.de) para aplicaciones realizadas en PHP.

Estos frameworks resultan muy útiles para controlar las pruebas de regresión, validando que el nuevo código cumple con los requisitos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación.
JUnit [JUnit] es un framework de código abierto creado por Erich Gamma y Kent Beck que se utiliza para escribir y ejecutar los tests de aplicaciones escritas en Java. Este framework se ha popularizado por la comunidad de eXtreme Programming [XP] y es ampliamente utilizado por los desarrolladores que implementan casos de pruebas unitarias en Java. En los últimos años, se convirtió en el estándar de facto de los tests unitarios.
JUnit incluye formas para presentar los resultados de las pruebas que pueden ser en modo texto, gráfico (AWT o Swing) o como tarea en Ant [Ant].
Las pruebas se representan en JUnit como clases Java y pueden definir cualquier cantidad de métodos públicos de prueba. Por convención, los métodos utilizan el prefijo “test” delante del nombre del caso de prueba, con el fin de que JUnit pueda identificarlos y ejecutarlos; los métodos “test” no tienen parámetros.
JUnit provee una librería (Assert) para poder comprobar el resultado esperado con el real. La prueba falla si la condición de la aserción no se cumple, generando un informe del fallo. Disponemos de métodos assert para comprobar si dos objetos son iguales (assertEquals), para comprobar si un objeto es nulo (assertNull), para comprobar si un objeto no es nulo (assertNotNull) y para comprobar si dos variables apuntan a un mismo objeto (assertSame).

10.4.1 ¿Por qué utilizar JUnit?

El framework JUnit presenta las siguientes ventajas:

- **Incrementa la calidad y la estabilidad del software**: con el uso de JUnit se incrementa la confianza en que los cambios del código realmente funcionan. Esta confianza permite ser más agresivo con la refactorización del código y la adición de nuevas características. Los tests validan la estabilidad del software y dan confianza en que los cambios no causarán efectos negativos en el software.
- **Es simple**: con JUnit es posible escribir y correr rápidamente los tests, incentivando de este modo su uso. El compilador "testea" la sintaxis del código y los tests "validan" la integridad del código.
- **Los test JUnit chequean sus propios resultados y proporcionan feedback inmediato:** los tests JUnit se pueden ejecutar automáticamente y chequean sus propios resultados. Cuando se corren los tests, se obtiene un feedback visual inmediato indicando si se ha pasado o fallado el test.
- **Es de Código Abierto.**
- **Los test JUnit pueden componerse como un árbol de conjuntos de tests**: los tests JUnit se pueden organizar en suites de tests que contienen ejemplares de tests e incluso otras suites. El comportamiento compuesto de los tests JUnit permite ensamblar colecciones de tests y automáticamente hacer un test regresivo de toda la suite de tests con un sólo paso. También se pueden ejecutar los tests de cualquier capa dentro del árbol de suites.

Para crear un test con JUnit 3.x debemos seguir estos pasos:

- Crear una clase que herede de la clase *TestCase*

```
public class MiTest extends TestCase
```

- Opcionalmente podemos crear un constructor con un parámetro de tipo String e invocar al constructor de la clase padre pasándole dicho String.

```
public MiTest(String nombre) {
 super(nombre)
}
```

- En cada clase de prueba se reescriben opcionalmente los métodos setUp() y tearDown():
  - **setUp()** se ejecuta antes de cada caso de prueba, método "test", y se utiliza para inicializar el entorno de pruebas. Ejemplo: inicializar variables de instancias, establecer conexiones con la base de datos.
  - **tearDown()** se llama después de la ejecución de cada caso de prueba con el objetivo de liberar determinados recursos del sistema, como los recursos u objetos inicializados en el método setUp().
- Crear uno o varios métodos que empiecen con "test", sin parámetros ni retorno de resultado.

```
public void testEjemplo() {
  ...
}
```

Aquí se listan las principales diferencias entre las versiones 3.x versus 4.5 o superior:

- Incluye anotaciones Java (Java 5 annotations):
  - @Test sustituye a la herencia de la clase *TestCase*.
  - @Before y @After como sustitutos de *setUp()* y *tearDown()*.
  - Se añade @Ignore para deshabilitar tests.
- Se elimina la distinción entre errores (errors) y fallos (failures).

10.4.2 Trabajando con EasyMock

EasyMock [EasyMock] es una librería de código abierto que extiende al framework JUnit permitiendo crear Objetos Fantasmas (en inglés Mock Objects) para simular parte del comportamiento del código de dominio. Los objetos fantasmas los crea dinámicamente, permitiendo además que devuelvan un resultado concreto para una entrada concreta.
EasyMock presenta las siguientes ventajas:

- Evita escribir manualmente los objetos que simulan determinados comportamientos para cada caso de prueba.
- Aumenta la confianza al realizar cambios en los nombres de métodos o reordenamiento de parámetros.
- Soporta el retorno de valores y excepciones.
- Soporta la ejecución de métodos en un orden dado para uno o más objetos mocks.

La restricción de EasyMock 2 es que únicamente trabaja con Java 2 versiones 5 o superior. EasyMock soporta únicamente la creación por defecto de objetos fantasmas a partir de sus interfaces. De todas maneras, EasyMock Class Extension derivado de EasyMock genera objetos mocks para interfaces y clases.
En términos generales, estos son los pasos para usar EasyMock:

1. Crear el objeto *mock* a simular a partir de su **interface** o **clase** (utilizando EasyMock Class Extension)**:**

```
HttpServletRequest request = EasyMock.createMock(HttpServlet
Request.class);
```

2. Si se necesita grabar el comportamiento esperado se puede proporcionar los parámetros de entrada concretos y los objetos de salida concretos con **expect**:

```
EasyMock.expect(request.getParameter("bookID")).andReturn("1");
```

3. Generar la implementación del *mock* con **replay**:

```
EasyMock.replay(request);
```

4. Si se especificó comportamiento, se podría verificar que el método del *mock* fue ejecutado con **verify**:

```
EasyMock.verify(request);
```

Así ya tendríamos un *mock* para el HttpServletRequest y podríamos probar el controlador pasándole este *mock*:

```
BookDetailsController controller = new BookDetailsController();

        controller.handleRequest(request, response);
```

Estos son algunas de las alternativas y agregados a EasyMock:

- Powermock (code.google.com/p/powermock) basado en EasyMock permite agregar comportamiento mock a métodos estáticos, constructores, clases y métodos finales y privados.
- EasyMock Property Utilities (www.stephenduncanjr.com/projects/easymock-propertyutils/index.html) es un agregado a EasyMock

que permite usar propiedades del estilo JavaBeans para los argumentos cuando se utiliza EasyMock.

- Jmock (www.jmock.org) es otro framework para generar objetos fantasmas en forma dinámica. Su principal debilidad es la poca flexibilidad para readaptarse a los cambios en el código, pudiendo romper casos de pruebas anteriores.
- Mockito (www.mockito.org) muy similar a EasyMock aunque los pasos de uso son algo distintos. Mockito no soporta el concepto de configurar un comportamiento por lo tanto no existen las verificaciones de los objetos mocks.
- Hamcrest (code.google.com/p/hamcrest) es una librería que se puede utilizar con EasyMock.

10.5 Mapeo del perfil UML de pruebas a JUnit

En esta sección se describe el mapeo del perfil UML de pruebas a JUnit. Este mapeo considera primariamente al framework JUnit, y en los casos donde no exista una asociación hacia JUnit, se mencionan ejemplos de cómo el framework tiene que extenderse para soportar los conceptos incluidos en el perfil. Complementariamente usamos el framework EasyMock como mecanismo de extensión a JUnit, por ejemplo para traducir el estereotipo de prueba <<TestComponent>>.

Tabla 10-2: *U2TP vs. JUnit*

UML Testing Profile	JUnit
Test Behavior:	
Test Control	Sobrecargar el método “runTest” de JUnit TestCase. Con esto se especifica la secuencia de ejecución de los Test Cases.
Test Case	El Test Case se representa con una operación en JUnit. Este método público pertenece a la clase del Contexto de Prueba (Test Context), por convención comienza con el prefijo “test” y no tiene argumentos para no utilizar el Control de Prueba (Test Control).

Test Invocation	Un Test Invocation es una operación que se puede llamar desde otra operación Test Case o desde el Control de Prueba (Test Control).
Test Objective	Se puede usar haciendo una llamada al método “setName” del framework de pruebas.
Stimulus	No existe un concepto similar aplicable en JUnit. Los Estímulos son parte directa de la implementación de los Test Cases (métodos de pruebas).
Observation	No existe un concepto similar aplicable en JUnit. Las Observaciones son parte directa de la implementación de los Test Cases (métodos de pruebas).
Coordination	Puede implementarse a partir de cualquier mecanismo de sincronización disponible para los Componentes de Pruebas. Ejemplo utilizando semáforos.
Default	No existe un concepto similar aplicable en JUnit. El mecanismo de excepciones en Java se podría utilizar para implementar la jerarquía default del perfil de pruebas
Verdict	En JUnit, los veredictos puede ser: *pass* (pasa), *fail* (falla), *error*. No existe un veredicto *inconclusive* en JUnit, por lo que se considera como un veredicto *fail*. Este último valor ya no existe en JUnit versión 4 o superior.
Validation Action	Se consideran las llamadas a la librería Assert de JUnit.
Log Action	No existe un mecanismo de log en JUnit.
Test Log	No existe un mecanismo de log en JUnit.
Test Architecture:	
Test Context	El contexto de prueba se representa a través de una clase que hereda de la clase TestCase de JUnit versión 3.8 (o bien utiliza la anotación @Test en versiones superiores).
Test Configuration	No existe un concepto similar aplicable en JUnit.

Test Component	No existe un componente de pruebas en JUnit. Para simularlo se utilizan extensiones a JUnit como Mock Objects (objetos fantasmas) como los mencionados anteriormente. EasyMock framework es una alternativa para complementar esta característica.
System Under Test (SUT)	El sistema bajo prueba no necesita traducirse específicamente a JUnit ya que cualquier clase del sistema puede considerarse como una clase utilitaria o una clase SUT.
Arbiter	Se puede considerar como una propiedad del Test Context de un tipo de resultado de la prueba (TestResult). El algoritmo por defecto genera los valores *Pass*, *Fail* y *Error* similar a un Veredicto, aunque esos valores se pueden redefinir por el usuario.
Scheduler	Se puede representar como una propiedad del Test Context.
Utility Part	Cualquier clase disponible en el classpath de Java se puede considerar como una clase utilitaria o parte del SUT.
Test Data:	
Wildcards	No existe un concepto similar aplicable en JUnit.
Data pool	Agrupa todas las operaciones de acceso al pool de datos en una clase.
Data partition	Clase que hereda del Data Pool con métodos para acceder a la partición de datos.
Data Selector	Es un método del Data Pool o Data Partition.
Coding Rules	No existe un concepto similar aplicable en JUnit.
Time Concepts:	
Time zone Timer	JUnit no soporta conceptos de tiempo, aunque se podrían implementar a través de las APIs estándares disponibles para manipular el tiempo.

Test Deployment:	
Test Artifact Test Node	El despliegue está fuera del alcance de JUnit.

10.6 Transformando modelos de prueba a código

En esta sección describimos las reglas de transformación para generar el código JUnit a partir del modelo de prueba escrito en U2TP; estas reglas son bien conocidas [U2TP 04].
¿Qué reglas debemos usar para generar el esqueleto del código de las pruebas? Primero debemos decidir cómo vamos a traducir los conceptos estructurales:

- Las clases raíces con el estereotipos <<TestContext>> (contexto de prueba) son traducidos como clases que heredan de TestCase en JUnit 3.x, para versiones superiores se utiliza la anotación **@Test**. Cabe señalar que el concepto de contexto de prueba existe en el frameworks JUnit pero definido de forma diferente.

- Las clases no raíces con el estereotipos <<TestContext>> son traducidos como clases respetando la jerarquía establecida en el modelo de prueba.

- Las operaciones con el estereotipo <<TestCase>> son traducidas como operaciones de la clase que representa al contexto de prueba. Por convención, los nombres de estas operaciones deben comenzar con “test” y no deben tener parámetros. Los posibles resultados son:

  - Pasa (en inglés pass): el sistema bajo la prueba cumple las expectativas.
  - Falla (en inglés fail): diferencia entre los resultados esperados y reales.
  - Error: un error (excepción) en el ambiente de prueba.

  En JUnit no hay un veredicto incierto (en inglés inconclusive), por lo general éste es traducido como una falla (fail).

- Las operaciones sin estereotipos son traducidas como operaciones Java de la clase correspondiente.

- Las clases con el estereotipo <<TestComponent>> no pueden ser traducidas directamente a JUnit ya que éste no maneja este concepto. Sin embargo, es posible utilizar ciertas extensiones como EasyMock y EasyMock Class Extension para crear los componentes de prueba.

- Las clases con el estereotipo <<SUT>> no necesitan traducirse; cualquier clase en el *classpath* puede considerarse como una clase de utilidad o una clase de SUT.

A continuación se muestra el código resultante del modelo de la figura 10-14. Podemos ver que existe una clase por cada clase con el estereotipo <<TextContext>>:

```
public abstract class ViewBookControllerTest extends TestCase {

  private ViewConBookController controller;
  private HttpServletRequest request;
    private HttpServletResponse response;
  private BookSpecDao dao;

  @Override
  protected void setUp() throws Exception {
    super.setUp();
    this.controller = new ViewConBookController();
    this.resquest=EasyMock.createMock(HttpServletRequest.class);
    this.response=EasyMock.createMock(HttpServletResponse.class);
    this.dao = EasyMock.createMock(BookSpecDao.class);
    this.controller.setBookSpecDaoEmulator(dao);
  }

  @Override
  protected void tearDown() throws Exception {
    super.tearDown();
    this.controller = null;
    EasyMock.verify(request);
    EasyMock.verify(dao);
  }

  public void testValidPageDisplayed() throws Exception {
  }

  protected abstract void verify(ModelAndView modelAndView);
```

```
  public void testValidDetailsFound() throws Exception {

  }
}

public class ViewBookFoundControllerTest extends ViewBookControllerTest {

  @Override
  protected void verifyBookDetails(ModelAndView modelAndView) {
  }

  @Override
  protected void verifyPageDisplayed(ModelAndView modelAndView) {
  }
}
public class ViewBookNotFoundControllerTest extends ViewBookControllerTest {

  @Override
  protected void verifyBookDetails(ModelAndView modelAndView) {
  }

  @Override
  protected void verifyPageDisplayed(ModelAndView modelAndView) {
  }
}
```

¿Qué reglas debemos usar para generar el código de los métodos de prueba? Debemos decidir cómo vamos a traducir los conceptos dinámicos (diagramas de secuencia). Por lo general esta transformación no es 100 % automatizada, es decir requiere que el programador complete el código resultante. Vamos a tomar como entrada de la transformación los diagramas 10-15 y 10-16. El primero de ellos determina el comportamiento para el caso de prueba validPageDisplayed:

```
public void testValidPageDisplayed() throws Exception {
  EasyMock.replay(request);
  EasyMock.replay(response);
  ModelAndView modelAndView=controller.handleRequest(request, response);
  this.verifyPageDisplayed();
}
```

El segundo diagrama especifica las validaciones de la página de detalle efectuadas en el curso alternativo:

```
@Override
protected void verifyPageDisplayed(ModelAndView modelAndView) {
   assertEquals("The bookdetails page should be displayed", "bookdetails",
                modelAndView.getViewName());
}
```

10.7 Implementación de la transformación de modelos a texto

Las transformaciones modelo a texto (M2T) crean simplemente un documento en formato de texto, a partir de la definición del metamodelo del modelo fuente. En esta sección presentamos la definición formal de las transformaciones de modelos (PIM) utilizando el lenguaje MOFScript [Oldevik 06] para generar el código de pruebas JUnit. Las definiciones elaboradas permiten que las transformaciones puedan ser ejecutadas en forma automatizada.

La herramienta MOFScript (www.eclipse.org/gmt/mofscript) permite la transformación de cualquier modelo MOF a texto, generando por ejemplo código Java, SQL Scripts, HTML o documentación a partir de los modelos. MOFScript provee un lenguaje de metamodelo independiente que permite el uso de cualquier clase de metamodelo y sus instancias para la generación de texto.

Presentamos parte del código de la definición de la transformación para convertir los modelos de pruebas escrito en UML2 a código JUnit.

```
//Define la transformación de un modelo UML2
texttransformation umlmodelgenerator (in
    uml:"http://www.eclipse.org/uml2/2.1.0/UML") {

  ...

  //Propiedades de estereotipos U2TP.
  property U2TP_VERDICT      : String = "Verdict"
  property U2TP_TEST_CONTEXT : String =
"TestContext"
  property U2TP_TEST_CASE    : String = "TestCase"

  /**
   * Módulo principal de entrada.
   */
  uml.Model::main() {
    //Recorre todos los paquetes del modelo y
procesa
```

```
    //procesa sus objetos.
    self.ownedMember->forEach(p:uml.Package) {
      p.mapPackage()
    }
  }


/**
 * Mapea una instancia UML a una clase Java o
 * JUnit Test case.
 */
uml.Class::mapClass(packageName : String) {
  ...
  /**
   * Si la clase tiene estereotipo <<TestContext>>
y no
   * super clase entonces extiende de TestCase,
sino
   * extiende de su super clase.
   */
  if (!self.superClass.isEmpty()) {
    <% extends %>self.superClass.first().name <%%>
  } else if (self.superClass.isEmpty() &&
            self.hasStereotype(U2TP_TEST_CONTEXT))
{
    <% extends TestCase%>
  }
  ...
  if (self.hasStereotype(U2TP_TEST_CONTEXT)) {
<%  /**
     * @param name
     */
    public %> self.name<%(String name) {
      super(name);
    }
    /* (non-Javadoc)
     * @see junit.framework.TestCase#setUp()
     */
    @Override
    protected void setUp() throws Exception {
      super.setUp();
    }
    /* (non-Javadoc)
     * @see junit.framework.TestCase#tearDown()
     */
```

```
    @Override
    protected void tearDown() throws Exception {
      super.tearDown();
    }
%>
  }
```

10.8 Validando la semántica de la transformación

En este capítulo nos hemos referido a la verificación y validación del software a través de sus modelos. Sin embargo esto no resulta suficiente dentro del paradigma MDD ya que los modelos además de ser entidades descriptivas, son entidades productivas que van sufriendo sucesivas transformaciones a lo largo del proceso de desarrollo. Por ello, además de analizar los modelos en sí mismos debemos también analizar sus transformaciones.

Actualmente, los lenguajes de transformación se enfocan principalmente en cómo expresar las transformaciones descuidando los aspectos relacionados con su validación. En general, la validación de la transformación puede incluir propiedades como corrección sintáctica de la transformación con respecto a su lenguaje de especificación y corrección sintáctica de los modelos producidos por la transformación (por ejemplo, los trabajos presentados en [LBM 06] y [KUSTER 06] incluyen la validación de estas propiedades). Pero pocas propuestas tratan la consistencia semántica de la transformación, es decir la preservación de la corrección del modelo destino con respecto al modelo origen correspondiente.

Sin embargo, el problema de la validación semántica de transformación de modelos no es un desafío nuevo originado por la filosofía de MDD. La idea de desarrollo de software basada en pasos incrementales, se remonta a la teoría de refinamientos de Dijkstra [Dijkstra 76]. Esta teoría cuenta con gran popularidad y desarrollo en la comunidad de los métodos formales, donde la mayoría de los esquemas de refinamiento se basan en el principio de sustitución [DB 01]. Normalmente el refinamiento se verifica demostrando que el sistema concreto simula al sistema abstracto.

Adaptar aquellas definiciones bien fundamentadas de refinamiento a la validación de la transformación de modelos se torna un gran desafío. En [PG 08] y [PG 06] se describe una propuesta formal ágil para la validación semántica del refinamiento de modelos. Los autores definen las

estructuras de refinamiento en términos de UML / OCL, resultando equivalentes a las estructuras empleadas en los lenguajes formales. La ventaja de esta propuesta reside en que se reemplaza la aplicación de lenguajes matemáticos ajenos al área de modelado, que normalmente no son aceptados por los modeladores, por lenguajes estándar que han sido ampliamente adoptados.

10.9 Conclusiones

Los términos verificación y validación (V&V) se refieren a los procesos de comprobación y análisis que aseguran que el software esté acorde con su especificación y cumpla las necesidades de los clientes. Contamos con distintas técnicas de V&V, entre ellas la verificación formal de programas y modelos, chequeo de modelos (model checking) y testing. En particular el testing es una alternativa práctica. Esta técnica consiste en el proceso de ejecutar repetidamente un producto para verificar que satisface los requisitos e identificar diferencias entre el comportamiento real y el comportamiento esperado.

Model-Driven Testing (MDT) es una técnica de prueba de caja negra que utiliza modelos de prueba (tanto estructurales como de comportamiento) para automatizar la generación de los casos de prueba. El perfil denominado UML 2.0 Testing Profile (U2TP) proporciona un marco formal para la definición de dichos modelos de prueba.

En este capítulo hemos presentado una implementación de este perfil y hemos definido las reglas de transformación para generar código JUnit a partir de los modelos de prueba.

Como aplicación, hemos modelado pruebas de unidad y pruebas más avanzados para una parte del sistema Bookstore, mostrando así la factibilidad de agilizar el proceso de testing gracias a la generación automática del código de las pruebas, tal como es propuesto por MDT.

CAPÍTULO 11

11. El proceso de desarrollo dirigido por modelos

En este capítulo explicaremos como planificar y administrar un proyecto de desarrollo de software dirigido por modelos. ¿Cuáles son las consecuencias de aplicar MDD al proceso de desarrollo de software? Comparando el proceso MDD con el proceso tradicional observamos que muchas cosas permanecen iguales. Los requisitos del sistema aún necesitan ser capturados y el sistema aún necesita ser probado e instalado. Lo que cambia de manera substancial son las actividades de análisis, diseño de bajo nivel y codificación. Nos referiremos a los siguientes temas:

- ¿Cuáles son las nuevas tareas involucradas en este tipo de proyectos?
- ¿Quién es responsable de las nuevas tareas?
- ¿Cuánto cuestan estas nuevas tareas?
- ¿Cómo debe organizarse el equipo de trabajo?
- ¿Cómo debe modificarse el proceso de desarrollo para obtener el mayor beneficio de MDD?

11.1 Entendiendo las tareas y los roles en el proyecto de desarrollo

El desarrollo de software dirigido por modelos tiene un efecto profundo sobre la forma en que construimos las aplicaciones de software. Las organizaciones y los proyectos frecuentemente dependen de expertos clave, quienes toman las decisiones respecto al sistema. MDD permite capturar su experiencia en los modelos y en las transformaciones, permitiendo de esta forma que otros miembros del equipo puedan aprovecharla sin requerir su presencia. Además, este conocimiento se mantie-

ne aún cuando los expertos se alejen de la organización. Por otra parte, el costo de desarrollo y de testing se reduce significativamente al automatizar gran parte del trabajo de generación del código y otros artefactos a partir de los modelos. A través de la automatización MDD favorece la generación consistente de los artefactos reduciéndose el número de errores.
Para lograr los beneficios de MDD el proceso de desarrollo de software debe adaptarse. En las siguientes secciones analizaremos los pasos necesarios para planear y administrar un proyecto dirigido por modelos.

11.1.1 Las tareas

El administrador de un proyecto MDD en realidad debe controlar un proyecto dentro de otro proyecto (figura 11-1). El proyecto interno consiste en fabricar herramientas MDD para que sean usadas por el equipo que trabaja en el desarrollo del proyecto externo.

Figura 11-1. Proyecto de desarrollo MDD: un proyecto dentro de otro proyecto.

El hecho de tener dos proyectos acoplados nos obliga a organizar y planificar más cuidadosamente, especialmente al comienzo del proyecto. A las dificultades usuales asociadas con cualquier proyecto de desarrollo, ahora debemos sumarle un conjunto adicional de dependencias internas. Debemos identificar, desarrollar e instalar las herramientas MDD

requeridas antes de que los desarrolladores las necesiten. Esto implica que los flujos de las tareas de ambos proyectos están interrelacionados. La parte positiva es que podemos decidir como distribuir el esfuerzo entre ambos proyectos. Por ejemplo, al identificarse un nuevo requisito, podríamos temporariamente mover algunos desarrolladores del proyecto externo hacia el proyecto de desarrollo de herramientas, de tal forma que ellos puedan mejorar las herramientas disponibles.

11.1.2 Los Roles

MDD distingue dos roles diferentes en el proceso de desarrollo: aquellos que crean la aplicación y aquellos que desarrollan las herramientas. Esta separación no es algo nuevo sino que la encontramos aplicada ya en otras industrias. En el desarrollo basado en componentes, algunas personas fabrican los componentes, o bien la unidad organizacional completa podría actuar como una fábrica de componentes, y otras personas usan estos componentes para crear sus aplicaciones. Similarmente, en MDD algunas personas crean la plataforma común para varios proyectos y otras personas desarrollan los productos usando los elementos de esta plataforma común. El desarrollo de software está cambiando su enfoque desde la idea de tener generalistas a la idea de tener especialistas, en forma similar a otras empresas e industrias. Esta separación de dos roles en MDD no significa que las personas son necesariamente diferentes. A veces quienes desarrollan la plataforma MDD también la usan. Lo que resulta crucial es que los desarrolladores más experimentados se encarguen de desarrollar la plataforma MDD. Estos desarrolladores experimentados pueden especificar mejor que aquellos con menos experiencia la automatización del proceso de desarrollo en términos de lenguajes de modelado, modelos, transformaciones entre modelos, generadores de código y frameworks del dominio. Además, ellos cuentan con el reconocimiento necesario entre sus colegas. Concretamente en MDD podemos identificar los siguientes roles:

- Los expertos en el dominio son personas que tienen conocimiento acerca del dominio del problema. Ellos conocen la terminología, conceptos y reglas del dominio y frecuentemente han estado involucrados en su definición. Los desarrolladores de aplicaciones también califican aquí cuando han desarrollado múltiples aplicaciones de tipo similar en el pasado; usualmente haber creado sólo una no es suficiente ya que podrían no contar con la experiencia necesaria para realizar generalizaciones que permitan encontrar las abstracciones apropiadas. Por ejemplo,

cuando desarrollamos aplicaciones de negocios, como el Bookstore descrito en el capítulo 5, los expertos en el dominio serían los expertos en ventas y gerentes.

- Los desarrolladores del lenguaje seleccionan y/o crean el lenguaje de modelado si no se dispone de ninguno que se ajuste al dominio de nuestro problema y lo formalizan mediante algun mecanismo (por ejemplo, un metamodelo o un perfil de UML). Además proveen guías sobre como usarlo, tales como manuales y ejemplos de modelos. Los desarrolladores del lenguaje trabajan conjuntamente con los expertos en el dominio y con usuarios clave del lenguaje. En general se usarán herramientas de metamodelado para implementar el lenguaje, las cuales han sido descritas en el capítulo 4, y no se requerirá programación adicional sino simplemente aplicar dichas herramientas o frameworks.

- Los modeladores (o ingenieros del PIM) aplican los lenguajes de modelado. Este grupo es usualmente el más grande. Son los encargados de crear los modelos a partir de los cuales se generará la aplicación. Los modelos también sirven para otros propósitos; por ejemplo, los ingenieros de prueba pueden usar los modelos para planear sus casos de prueba, como vimos en el capítulo 10. Por su parte, los ingenieros de emplazamiento pueden producir sus programas de instalación y configuraciones del producto usando los modelos. Además, a partir de los modelos los gerentes pueden obtener reportes y métricas sobre el estado del desarrollo.

- El ergonomista puede ayudar a los desarrolladores del lenguaje a mejorar la usabilidad del lenguaje. Si bien la usabilidad es siempre un factor relevante, este rol puede ser particularmente significante en ciertos casos. Por ejemplo, cuando creamos soluciones para el desarrollo de interfaces de usuario (IU), puede ser importante que los modelos se correspondan ajustadamente con los productos reales. La persona que define los estilos de la IU puede también ayudar a definir el lenguaje de modelado. El rol del ergonomista también es relevante cuando el lenguaje es usado por personas sin conocimientos informáticos y/o cuando los usuarios pertenecen a distintos países y culturas.

- Los desarrolladores de las transformaciones y de los generadores de código especifican las transformaciones del PIM al PSM y del PSM al código. Ellos son expertos en el uso de lenguajes de transformación como QVT, ATL y Mof2Text. También poseen buen conocimiento sobre la plataforma de implementación y el framework del dominio.

- Los expertos en el framework del dominio (o ingenieros del PSM) son usualmente arquitectos de software o desarrolladores con experiencia en el uso y desarrollo de componentes y frameworks. Ellos pueden proveer implementaciones que se usarán como referencia y pueden especificar como usar el framework del dominio o bien construirlo si no existe aún un framework para el dominio en cuestión.
La figura 11-2 muestra las conexiones entre los roles que se juegan en el proceso MDD, los artefactos que se producen y las herramientas que se utilizan para producirlos.

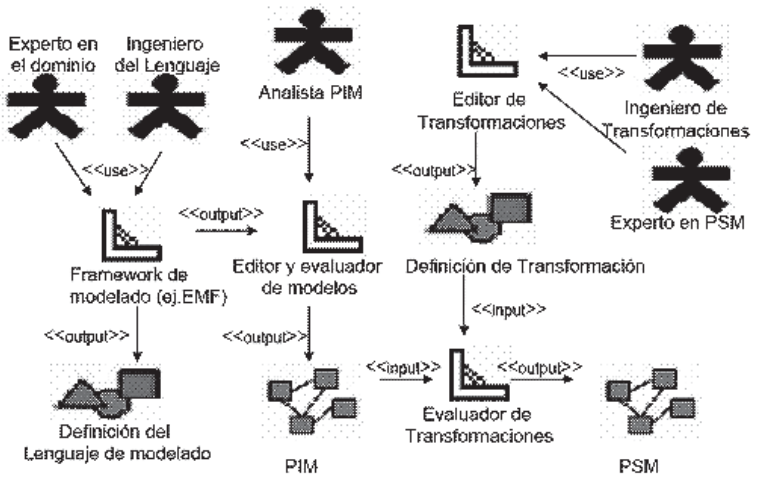

Figura 11-2. Los principales roles, artefactos y herramientas en el proceso MDD

11.2 Planificación y seguimiento del proyecto de desarrollo

Una vez que se han definido las tareas, las habilidades requeridas y las dependencias entre las tareas, el proceso de planificación y seguimiento de un proyecto MDD es similar al de cualquier proyecto de desarrollo de software. Se analiza la envergadura de las tareas y se le asignan los recursos apropiadamente. Se necesitan las habilidades usuales de un gerente de proyecto para detectar los lugares donde el cronograma no

se cumple y realizar los ajustes apropiados a la asignación de recursos cuando resulte necesario.
Las secciones siguientes proveen algunos consejos adicionales acerca del esfuerzo de planificación y seguimiento del proyecto.

11.2.1 Usando un proceso iterativo

Es aconsejable separar el desarrollo en varias iteraciones. En la primera iteración desarrollaremos un ejemplo de cada tipo de modelo y transformación. Esto tiene los siguientes beneficios:
- El equipo de desarrollo gana experiencia y puede estimar cuanto tiempo y esfuerzo implicará la construcción del resto del proyecto.
- Todo el equipo de desarrollo está activo, es decir, los implementadores de transformaciones no tienen que esperar a que todo el modelo PIM este listo para comenzar a trabajar en sus transformaciones.
- Permite mostrar algunos resultados anticipadamente lo cual ayudará a disminuir el escepticismo y aumentar la confianza en el éxito del proyecto.

Las siguientes iteraciones trabajarán sobre la experiencia ya ganada para soportar escenarios de la aplicación cada vez más amplios.

11.2.2 Seguimiento del proyecto

Durante el desarrollo del proyecto es útil tener en cuenta las siguientes guías:
- Realice una inversión explícita en las herramientas de soporte.
- Use a la gente más calificada para desarrollar las herramientas MDD con el objetivo de capturar y automatizar su experiencia.
- Considere que además del código el proyecto generará documentos, configuraciones, reportes y casos de prueba.
- Asegúrese de que su proceso de desarrollo soporta ambientes de prueba además de ambientes de producción.
- Recuerde definir las estrategias de manejo de configuraciones para sus herramientas MDD.
- Asigne tiempo al entrenamiento del equipo sobre el uso de las herramientas MDD.
- Tómese tiempo para considerar si sus herramientas MDD serán reusables en proyectos futuros.

11.2.3 Evaluación del proyecto

Al finalizar un proyecto MDD es útil generar las siguientes métricas:
- El costo de desarrollo de las herramientas.
- La productividad de los desarrolladores de la aplicación al usar las herramientas. Para realizar una comparación justa con respecto a los proyectos tradicionales, será necesario expresarla en términos del esfuerzo que hubiese requerido desarrollar todo el código manualmente.
- El nivel de calidad logado por el equipo de desarrollo.
- El esfuerzo requerido para permitir que las herramientas MDD puedan ser re-usadas en los siguientes proyectos y los beneficios potenciales que esto reporta.

Estas mediciones permitirán evaluar el valor real derivado de aplicar la propuesta MDD y también darán guías acerca de como explotar mejor sus potenciales en los proyectos futuros.

11.3 Administración y re-uso de los artefactos

Modelos, transformaciones, y código son los principales artefactos que se generan y se usan en un proyecto MDD. Estos artefactos pueden ser tan simples como archivos de texto conteniendo propiedades de configuración y documentación, o tan complejos como archivos de proyecto conteniendo modelos, código ejecutable y descriptores de emplazamiento.
Típicamente, los ingenieros de software prefieren diseñar y construir sus propios artefactos, y los gerentes de los proyectos prefieren evitar las dependencias entre proyectos que se generan como consecuencia de compartir artefactos. Sin embargo, la intención de re-usar artefactos siempre está presente, debido a su gran potencial para ahorrar tiempo y costos en el desarrollo de software. MDD abre nuevas posibilidades de re-uso que no deben ser desaprovechadas, por ejemplo:
- Cuando un proyecto define sus modelos conceptuales, es más fácil detectar puntos en común entre lo que se necesita construir y lo que ya fue construido, debido a que ambos modelos conceptuales tienen un nivel de abstracción más alto que el código, eliminando los detalles que oscurecen la identificación de patrones comunes.
- Un pequeño cambio en un perfil de UML y en las transformaciones que lo usan, puede permitir a un proyecto re-usar todo el ambiente de desarrollo de un proyecto anterior.

- Cuando las transformaciones hacen buen uso de archivos de datos de configuración y plantillas, un proyecto puede generar diferentes artefactos con sólo realizar pequeños cambios a dichos archivos.

11.3.1 Re-uso de los artefactos

El éxito de un proyecto MDD depende del re-uso exitoso de los artefactos. El re-uso en MDD incluye el re-uso de patrones, modelos, transformaciones y en menor medida el del código. La administración de estos artefactos, sus descripciones relacionadas y el mantenimiento de los repositorios se torna cada vez más importante, incluyendo las siguientes cuestiones:

- Identificar y recuperar un artefacto para re-usarlo.
- Asegurarse de haber recuperado el artefacto apropiado para la versión de la plataforma destino.
- Analizar la integridad de un artefacto y verificar que sea la última versión o la versión apropiada.
- Asegurarse de que el artefacto esté certificado cuando esto sea requerido.

A la hora de decidir qué métodos y herramientas se utilizarán para la administración de los artefactos, debemos considerar un conjunto de factores, incluyendo servicios de integridad y soporte para el emplazamiento de los artefactos.

11.3.2 Re-uso de las herramientas

Las herramientas MDD que construimos en el sub-proyecto interno serán usadas numerosas veces por los desarrolladores para crear los artefactos de la aplicación en el proyecto externo. También es conveniente que estas herramientas se re-usen en proyectos futuros, amortizando de esta manera el costo insumido en su construcción.
Las herramientas MDD incrementan el porcentaje de tiempo que los ingenieros de software dedican al proceso creativo de diseñar el sistema, comparado con el proceso rutinario de codificar y depurar el software. Como resultado, una vez que ellos han visto los beneficios del proceso MDD, están motivados a explotar y extender las herramientas MDD en los siguientes proyectos. Por lo tanto, es valioso investigar en que casos las herramientas MDD que se crean para un proyecto tienen potencial para ser re-usadas en otros proyectos futuros. Esta evalua-

ción puede realizarse durante la fase de planificación, al comienzo del proyecto, o durante la fase de revisión hacia el final del proyecto.

11.4 Resumen

El administrador de un proyecto MDD en realidad debe controlar un proyecto dentro de otro proyecto. El proyecto interno consiste en fabricar herramientas MDD para que sean usadas por el equipo que trabaja en el desarrollo del proyecto externo. Por lo tanto, MDD distingue dos roles diferentes en el proceso de desarrollo: aquellos que crean la aplicación y aquellos que desarrollan las herramientas. Concretamente podemos identificar los siguientes roles: los expertos en el dominio, los desarrolladores del lenguaje, los modeladores, el ergonomista, los desarrolladores de las transformaciones y de los generadores de código y por último los expertos en el framework del dominio y la plataforma de implementación.

Una vez que ha quedado claro el orden de las tareas en un proyecto MDD, el proceso de planificación y seguimiento es similar al de cualquier proyecto de software. Se definen las tareas, se analiza su envergadura y se le asignan recursos apropiadamente.

En gran medida el éxito de un proyecto MDD depende del re-uso exitoso de los artefactos. El re-uso en MDD incluye el re-uso de patrones, modelos, transformaciones y en menor medida el re-uso del código. La administración de estos artefactos, sus descripciones relacionadas y el mantenimiento de los repositorios se torna entonces un tema relevante.

CAPÍTULO 12

12. El panorama actual y futuro

En este capítulo discutiremos la factibilidad de los objetivos de MDD, así como también los obstáculos que dificultan su avance. Nos referiremos a los desarrollos que esperamos observar en un futuro cercano. Describiremos que aspectos aún no se han logrado implementar, pero que son factibles si se continúa trabajando en la dirección correcta.

12.1 Las promesas de MDD

Con el propósito de enfrentarse a los continuos cambios que sufren los sistemas software y las tecnologías relacionadas con las plataformas de implementación, MDD establece una separación clara entre la funcionalidad básica del sistema y su posterior implementación. Se separa la lógica del negocio de la tecnología de la plataforma sobre la que se implementa, de modo que los cambios de plataforma no afecten al código de la lógica del negocio. Distinguiendo así entre modelos independientes de la plataforma (PIM) y modelos específicos de la plataforma (PSM). Por otro lado, las transformaciones de modelos (en sus variantes modelo-a-modelo y modelo-a-texto) permiten generar de forma automática el código final de la implementación de un sistema a partir de los diferentes modelos creados.
Pueden ser discutibles las realizaciones actuales del concepto MDD, sin embargo es difícil cuestionar sus principios básicos. El incremento de los niveles de abstracción y automatización, así como el uso de estándares, realizado correctamente, son todos principios de innegable utilidad. Además existen varios ejemplos de aplicaciones exitosas de MDD en grandes proyectos industriales [OMGss].

Mirando hacia atrás podemos ver que ha ocurrido un cambio de foco en el desarrollo de sistemas de software a lo largo de los años. Tempranamente se escribian programas usando lenguajes de bajo nivel (como Assembler). Más tarde se produjo un cambio de perspectiva orientada a escribir programas en lenguajes de alto nivel (como Cobol). El hecho de que estos programas de alto nivel tengan que ser traducidos a programas escritos usando lenguajes de bajo nivel es algo que pasa casi inadvertido. Los programadores no necesitan preocuparse sobre la generación de código Assembler, ya que esa tarea ha sido automatizada y podemos confiársela al compilador.
Adoptando la propuesta MDD, el foco se desplaza otro nivel más. Hoy en día la actividad principal es desarrollar código, frecuentemente escrito en un lenguaje de programación orientado a objetos. En el futuro, el foco se desplazará a escribir el PSM, y de allí a escribir el PIM. La gente olvidará el hecho de que el PSM necesita ser transformado a código ya que la generación de código estará automatizada. Y posiblemente años más tarde, la gente también olvidará que el PIM necesita ser transformado a un PSM.

12.2 Los obstáculos que enfrenta MDD

Cuando explicamos el paradigma MDD a los desarrolladores de software, generalmente recibimos una respuesta escéptica. La respuesta típica es: “Esto no puede funcionar. No podemos generar un programa completo a partir de modelos. Siempre necesitaremos ajustar algo en el código generado”. Pareciera que las promesas de MDD resultan demasiado ambiciosas y siguen existiendo controversias significativas sobre su utilidad. Hay variadas razones para el lento ritmo de adopción. Como mencionó Bran Selic en [Selic 08] estas razones se pueden clasificar básicamente en técnicas, culturales y económicas.

Obstáculos Técnicos:
Un problema que afecta a muchos productos de software actuales es la *usabilidad*. En el caso de MDD la usabilidad se pone de manifiesto generalmente en sus herramientas que tienden a ser difíciles de aprender y de utilizar. Esta falencia se debe a la falta de experiencia que poseemos sobre el uso de este tipo de herramientas. Nos resulta difícil decidir que tipo de interfaces y facilidades debería proveer la herramienta para resultar amigable a sus usuarios. Un segundo problema técnico importante es que todavía hay muy poca base teórica para MDD. Mucha de la tecnología MDD disponible ha sido desarrollado de forma *ad hoc* por

equipos profesionales que estaban tratando de resolver problemas específicos. La falta de una teoría sólida se manifiesta también en los problemas de interoperabilidad entre las herramientas.

Obstáculos Culturales:
Existen problemas para la adopción de MDD a causa de la resistencia que se produce al introducir nuevos métodos o herramientas en un entorno productivo que está en funcionamiento. Se necesita tiempo para aprender y adaptarse a nuevas formas de trabajo. Tal vez la cuestión más difícil de superar de todas ellas es la mentalidad conservadora de muchos profesionales del software que ofrecen resistencia al cambio tecnológico.

Obstáculos Económicos:
Hoy en día el entorno empresarial se centra predominantemente en el corto plazo para medir el retorno de la inversión. En consecuencia, es difícil justificar inversiones a más largo plazo en nuevos métodos y herramientas para el desarrollo de software, en especial si el retorno de la inversión no está garantizado.

12.3 Un cambio de paradigma

Creemos que MDD cambiará el futuro del desarrollo de software significativamente. Un argumento para sustentar esta opinión radica en que ya podemos lograr ventajas significativas en productividad, portabilidad, interoperabilidad y esfuerzo de mantenimiento, a través de la aplicación de MDD y sus herramientas de transformación, aunque MDD aún este inmaduro. Es decir que MDD puede ser usado aún cuando no todos sus elementos estén completamente desarrollados e implementados.
Un segundo argumento a favor de MDD proviene de la historia de la computación. Hacia finales de los 60s el uso del lenguaje Assembler comenzó a ser substituido por el uso de lenguajes procedurales. En aquellos días existía también gran escepticismo y con razón, ya que los primeros compiladores no funcionaban muy bien. El proceso de compilación solía ser asistido manualmente y muchos programadores estaban preocupados pensando que el código Assembler generado automáticamente sería menos eficiente que el código que ellos podrían generar manualmente. Muchos de ellos no podrían creer que en pocos años más los compiladores se volverían tan buenos que ya nadie intentaría realizar el trabajo manualmente. Hasta cierto punto, el escepticis-

mo era acertado. Se pierde eficiencia y velocidad y no podemos programar todos los trucos del Assembler en un lenguaje procedural. Sin embargo, las ventajas de los lenguajes procedurales se fueron tornando más obvias. Usando lenguajes de alto nivel podemos escribir software más complejo de una forma más rápida, y el código resultante es más fácil de mantener. Al mismo tiempo, mejores tecnologías de compiladores lograron ir disminuyendo sus desventajas. El código Assembler generado se volvió más eficiente. Hoy aceptamos el hecho de que no debemos programar nuestros sistemas en Assembler.
Lo que MDD nos presenta es un cambio similar al que hemos experimentado en torno al Assembler. Los PIMs son compilados (es decir transformados) a PSMs, los cuales son compilados a código de algún lenguaje de programación, que es finalmente compilado a lenguaje Assembler.
Los compiladores de PIM a PSM (es decir, las herramientas de transformación) no serán eficientes en los próximos años. Sus usuarios necesitarán brindarles asistencia para transformar algunas partes de un modelo. Pero eventualmente las ventajas de trabajar a un nivel más alto de abstracción se tornarán claras y evidentes para todas las personas involucradas en el proceso de desarrollo de software.

12.4 El camino a seguir

Hay al menos cuatro posibles direcciones para mejorar la aceptación y aplicación práctica de MDD: educación, investigación, estándares y desarrollo de herramientas de soporte.

Educación. Como hemos visto, existen importantes dificultades en cambiar la cultura dominante centrada en la tecnología. Es necesario comenzar tempranamente ese cambio en la formación, desde el nivel de grado universitario. Esto significa inculcar a los estudiantes de ingeniería de software, la comprensión y el respeto hacia el usuario. Una necesidad principal es comprender el valor que tiene para sus clientes y usuarios el producto que desarrollan. Esto a su vez necesita un conocimiento de las consecuencias económicas y del entorno económico de dicho producto. Es decir, se necesita una idea que va más allá de las cuestiones tecnológicas inmediatas. Los ingenieros de software deben tener un entendimiento y conocimiento práctico de los factores humanos, económicos y empresariales que influyen en lo que diseñan y construyen. Finalmente, existe la necesidad de incrementar la introducción de métodos MDD en la formación en ingeniería de software. La mayoría de los planes de estudio de

grado incluyen elementos básicos sobre ingeniería basada en modelos, como cursos de UML. Pero, sin los fundamentos teóricos apropiados, generalmente los resultados son insuficientes y poco coherentes.

Investigación. La comunidad científica ha adoptado las ideas de MDD en parte porque ven que es una oportunidad para efectuar un cambio importante en la tecnología de software. Por ejemplo, los lenguajes de modelado se pueden diseñar para evitar la complejidad arbitraria de los actuales lenguajes de programación. Esta complejidad es a veces un obstáculo en la aplicación de métodos eficaces como análisis matemáticos para predecir las características claves de un sistema antes de su construcción. Los nuevos lenguajes de modelado se pueden diseñar para soportar este tipo de análisis. Aun existiendo entusiasmo en los investigadores respecto a la aceptación de MDD, hay ciertos problemas en los esfuerzos para abordarlo. Uno es que gran parte de la investigación se centra en problemas puntuales, pues gran parte de la financiación la proporcionan socios industriales, interesados en soluciones a problemas inmediatos. Esto trae aparejado una carencia de estudio suficiente de los fundamentos teóricos de MDD. Por lo tanto, lo que se necesita es un plan global para investigación en MDD, donde las diferentes áreas de estudio estén claramente identificadas así como las relaciones entre ellas. Sólo cuando esto se entienda correctamente se podrá hablar de una teoría de MDD sistemática y completa.

Estándares. La estandarización permite no sólo la obtención de resultados probados de forma independiente de los proveedores, sino que también facilita la especialización, proporcionando una base común para el intercambio de trabajos entre especialidades. Además de UML y QVT, el OMG propuso otros estándares para conseguir la interoperabilidad entre fabricantes y herramientas de modelado. Algunos ejemplos de esos estándares son OCL para especificar restricciones en los modelos, XMI como formato de intercambio de modelos y MOF como lenguaje de metamodelado. Sin duda, la apuesta por el metamodelado como base teórica para la creación de los lenguajes de modelado ha sido uno de los puntos fuertes de MDD y a lo largo de estos años, MOF ha sido la principal referencia como paradigma de metamodelado. Evidentemente debe haber una estrecha relación entre la investigación, la industria y los organismos de normalización para lograr la definición de estándares de calidad.

Herramientas de soporte. Las herramientas de modelado y de transformación actuales no son lo suficientemente usables, eficientes y confiables. Se necesita trabajar en la construcción de herramientas

mejores, que permitan expresar todo tipo de modelos y que garanticen su transformación completa a código. Además es indispensable que el código generado sea eficiente.

12.5 Resumen

Aunque MDD no ha alcanzado aún su estado de madurez, ha mostrado su potencial para cambiar significativamente la forma en que desarrollamos software. Hoy en día el foco del desarrollo de software está puesto en el código, en el futuro el foco se desplazará a la escritura de PSMs y desde allí a la escritura de PIMs. La gente olvidará el hecho de que estos modelos necesitan ser transformados a código, ya que la generación de código se realizará de forma automática, pasando inadvertida. Surgirán lenguajes cada vez más expresivos que permitirán la especificación completa del sistema, incluyendo tanto sus aspectos estáticos como dinámicos. Estos lenguajes tendrán el mismo estatus que los lenguajes de programación actuales.

A pesar de que existen numerosos ejemplos verificables de aplicación exitosa de MDD en la práctica industrial, que corroboran su viabilidad y valor, el uso de MDD es todavía la excepción y no la norma. Esto se debe a los obstáculos que se deben superar. Aunque muchos de ellos son de carácter técnico, los obstáculos más difíciles de superar son los que se derivan de la actual idiosincrasia dentro de la cultura del desarrollo de software.

Podemos prever una introducción gradual de MDD, facilitada principalmente por cambios en los planes de estudio y por inversión en investigación básica. Es necesario el desarrollo de una teoría sistemática de MDD, que asegure que la correspondiente tecnología y sus métodos son bien comprendidos, útiles y fiables.

Finalmente, la multiplicación de herramientas de código abierto que soportan la visión MDD, sobre todo en el entorno Eclipse, está contribuyendo de forma muy significativa al éxito de MDD como nuevo paradigma de desarrollo. Sin embargo, estas herramientas necesitan ser mejoradas para lograr su aceptación y uso masivo.

Glosario

En este glosario compilamos los acrónimos más usados en entornos MDD. Incluimos básicamente los que corresponden a estándares, como así también los que de algún modo se relacionan con conceptos MDD.

ATL

ATL (ATLAS Transformation Language) es un lenguaje de transformación de modelos y un conjunto de herramientas desarrolladas por el Grupo ATLAS (INRIA & LINA). En el campo de Model-Driven Engineering (MDE), ATL provee formas de producir un conjunto de modelos destino, desde un conjunto de modelos fuente. Desarrollado sobre la plataforma Eclipse, el ATL Integrated Environnement (IDE) provee un número de herramientas estándar de desarrollo (syntax highlighting, debugger, etc.) que facilita el desarrollo de transformaciones en ATL. El proyecto ATL incluye también una librería de transformaciones ATL.

BNF

Acrónimo inglés de Backus Naur Form. Este formalismo es una meta sintaxis usada para expresar gramáticas libres de contexto, es decir, una manera formal de describir cuáles son las palabras básicas del lenguaje y cuáles secuencias de palabras forman una expresión correcta dentro del lenguaje. Una especificación en BNF es un sistema de reglas de derivación.

CIM

Acrónimo inglés de Computation Independent Model. Un CIM es una vista del sistema desde un punto de vista independiente de la computación. Un CIM no muestra detalles de la estructura del sistema. Usualmente al CIM se lo llama modelo del dominio y en su construcción se utiliza un vocabulario que resulta familiar para los expertos en el dominio en cuestión.

CMOF
Acrónimo inglés de Complete Meta-Object Facility. MOF 2.0 define a CMOF con capacidades más avanzadas que MOF.

CORBA
Acrónimo inglés de Common Object Request Broker Architecture (arquitectura común intermediaria en peticiones a objetos), es un estándar definido y controlado por el Object Management Group (OMG). CORBA define las APIs, el protocolo de comunicaciones y los mecanismos necesarios para permitir la interoperabilidad entre diferentes aplicaciones escritas en diferentes lenguajes y ejecutadas en diferentes plataformas, lo que es fundamental en computación distribuida.

DAO
Acrónimo inglés de Data Access Object. Patrón para definir objetos de acceso a datos. El framework Spring proporciona un soporte excelente para DAOs.

DSL
Acrónimo inglés de Domain Specific Language. Refiere a lenguajes definidos usando conceptos del dominio del problema particular.

DSM
La iniciativa DSM (Domain-Specific Modeling) es principalmente conocida como la idea de crear modelos para un dominio, en un DSL apropiado para dicho dominio, es decir especificar directamente la solución usando conceptos del dominio del problema.

Eclipse
Eclipse es una plataforma de software de código abierto independiente de otras plataformas. Esta plataforma, típicamente ha sido usada para desarrollar entornos integrados de desarrollo (del inglés IDE), como el IDE de Java llamado Java Development Toolkit (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse (y que son usados también para desarrollar el mismo Eclipse). Sin embargo, también se puede usar para otros tipos de aplicaciones cliente. Eclipse es también una comunidad de usuarios, extendiendo constantemente las áreas de aplicación cubiertas. Un ejemplo es el Eclipse Modeling Project, que cubre casi todas las áreas de Model Driven Engineering. Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente no lucrativa, que fomenta una comuni-

dad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

Ecore
El metamodelo MOF está implementado mediante un plugin para Eclipse llamado Ecore. Este plugin respeta las metaclases definidas por MOF. Todas las metaclases mantienen el nombre del elemento que implementan y agregan como prefijo la letra "E", indicando que pertenecen al metamodelo Ecore. Por ejemplo, la metaclase EClass implementa la metaclase Class de MOF.

EMF - Eclipse Modeling Framework Project (EMF).
El proyecto EMF es un framework de modelado y facilidades de generación de código para la creación de herramientas de instalación y otras aplicaciones basadas en un modelo de datos estructurados. Desde una especificación de modelo descrito en XMI, EMF ofrece herramientas y soporte en tiempo de ejecución para producir un conjunto de clases Java para el modelo, junto con un conjunto de clases adaptadoras que permiten la visualización y la edición basada en comandos, del modelo, y un editor básico.

EMOF
Acrónimo inglés de Essential Meta-Object Facility. EMOF es la parte de la especificación de MOF 2.0 que se usa para la definición de metamodelos simples, usando conceptos simples.

GEF
GEF es el acrónimo inglés de Graphical Editing Framework. Es un framework implementado para la plataforma Eclipse que ayuda en el desarrollo de componentes gráficos. Consiste de tres componentes principales, draw2d usado para los componentes visuales, Request/ Commands usado durante la edición para crear pedidos y comandos capaces de rehacer y deshacer acciones, y por último una paleta de herramientas para mostrar las opciones al usuario.

GMF
Acrónimo inglés de Graphical Modeling Framework, proyecto Eclipse.

GMT
Acrónimo inglés de Generative Modeling Technologies, proyecto Eclipse.

HSQL
Acrónimo inglés de Hyperthreaded Structured Query Language. Es una base de datos escrita en Java, que permite trabajar en varios "modos" standalone, server, etc. Según sus creadores puede llegar a manejar varios GB de DATA (8GB para tablas de disco). Trabaja con estándar SQL92 y soporta la creación de procedimientos almacenados (escritos en Java). Cuenta con una herramienta para la ejecución de sentencias SQL (Manager Swing) creada en Java Swing.

HTML
Es el acrónimo inglés de HyperText Markup Language, que se traduce al español como Lenguaje de Marcas Hipertextuales. Es un lenguaje de marcación diseñado para estructurar textos y presentarlos en forma de hipertexto, que es el formato estándar de las páginas web. Gracias a Internet y sus navegadores, el HTML se ha convertido en uno de los formatos más populares y fáciles de aprender que existen para la elaboración de documentos para web.

IM
Acrónimo inglés de Implementation Model (Code). El resultado del paso final en el desarrollo es la transformación de cada PSM a código fuente. Ya que el PSM está orientado al dominio tecnológico específico, esta transformación a IM, es bastante directa.

JSP
Acrónimo inglés de JavaServer Pages. Es una tecnología Java que permite generar contenido dinámico para web, en forma de documentos HTML, XML o de otro tipo. Una de las ventajas de utilizar JSP es que se hereda la portabilidad de Java, y es posible ejecutar las aplicaciones en múltiples plataformas sin cambios.

MDA
Acrónimo inglés de Model Driven Architecture, que en español se traduce como arquitectura conducida por modelos, fue presentada por el consorcio OMG (Object Management Group) a finales de 2000 con el objetivo de abordar los desafíos de integración de aplicaciones y los continuos cambios tecnológicos. MDA propone el uso de estándares como MOF, UML, JMI o XMI que en conjunto soportan al paradigma MDD.

MDD
Otro acrónimo relacionado a MDE es Model-Driven Software Development (MDD), que en español se traduce como desarrollo de

software conducido por modelos. Es visto como un sinónimo de MDE, ambos describen la misma metodología de desarrollo de Software.

MDE
Acrónimo inglés de Model Driven Engeenering, en español se traduce como Ingeniería de Software Conducida por Modelos. En MDE los modelos son considerados los conductores primarios en todos los aspectos del desarrollo de software.

MDF
Acrónimo inglés de Meta-Data Framework. Los DSLs no utilizan ningún estándar del OMG para su infraestructura, es decir no están basados en UML; los metamodelos no son instancias de MOF; a diferencia usan por ejemplo MDF el framework de Metadatos para este propósito.

MDT
Acrónimo inglés de Model-Driven Testing, es una propuesta reciente con forma de prueba de caja negra que utiliza modelos estructurales y de comportamiento descritos por ejemplo, con el lenguaje estándar UML, para automatizar el proceso de generación de casos de prueba.

MOF
El lenguaje MOF, acrónimo inglés de Meta-Object Facility, es un estándar del OMG para la MDD. MOF se encuentra en la capa superior de la arquitectura de 4 capas propuesta por el OMG. Provee un meta-meta lenguaje en la capa superior, que permite definir metamodelos. El ejemplo más popular es el metamodelo UML, que describe el lenguaje UML.

Mof2Text
El lenguaje Mof2Text, acrónimo inglés de MOF Model to Text Transformation Language, es el estándar especificado por el OMG para definir transformaciones modelo a texto. Sus metaclases principales son extensiones de OCL, de QVT o de MOF.

OCL
Lenguaje de restricciones para objetos (OCL, por su sigla en inglés, Object Constraint Language) es un lenguaje declarativo para describir reglas que se aplican a metamodelos MOF, y a los modelos UML. En la actualidad es parte del estándar UML. El Object Constraint Language permite definir restricciones y consultas sobre expresiones de objetos de cualquier modelo o metamodelo MOF que no pueden ser expresadas mediante la notación gráfica.

OMG
El Object Management Group u OMG (de su sigla en inglés grupo de gestión de objetos) es un consorcio dedicado a la gestión y el establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como UML, XMI, CORBA. Es una organización no lucrativa que promueve el uso de tecnología orientada a objetos mediante guías y documentos de especificación de estándares. El grupo está formado por compañías y organizaciones de software como Hewlett-Packard (HP), IBM, Sun Microsystems, Apple Computer.

PIM
The Platform Independent Model. Un PIM es un modelo con un alto nivel de abstracción que es independiente de cualquier tecnología o lenguaje de implementación. Dentro del PIM el sistema se modela desde el punto de vista de cómo se soporta mejor al negocio, sin tener en cuenta como va a ser implementado: ignora los sistemas operativos, los lenguajes de programación, el hardware, la topología de red, etc. Por lo tanto un PIM puede luego ser implementado sobre diferentes plataformas específicas.

PSM
The Platform Specific Model. Como siguiente paso, un PIM se transforma en uno o más PSM (Platform Specific Models). Un PIM representa la proyección de los PIMs en una plataforma específica. Un PIM puede generar múltiples PSMs, cada uno para una tecnología en particular. Generalmente, los PSMs deben colaborar entre sí para una solución completa y consistente. Por ejemplo, un PSM para Java contiene términos como clase, interfase, etc. Un PSM para una base de datos relacional contiene términos como tabla, columna, clave foránea, etc.

pUML
Acrónimo inglés para “the precise UML group”. El grupo fue creado en 1997 para reunir a investigadores y desarrolladores de todo el mundo con el objetivo de convertir al Unified Modelling Language (UML) en un lenguaje de modelado formal (es decir, bien definido). El grupo ha desarrollado nuevas teorías y técnicas

QVT
En MDD, QVT (Query/ View/ Transformation) es el lenguaje estándar para describir transformaciones de modelos definido por el OMG. La especificación del lenguaje QVT tiene una naturaleza híbrida, declarativa/ imperativa, con la parte declarativa dividida en una arquitectura de dos niveles.

SPEM
Acrónimo inglés para Software Process Engineering Metamodel. El OMG propone al metamodelo SPEM para describir el proceso de software.

U2TP
Acrónimo inglés para UML 2.0 Testing Profile. Este lenguaje se ha implementado utilizando el mecanismo de perfiles de UML 2.0. Desde 2004 es el estándar del OMG que proporciona un marco formal para la definición de un modelo de prueba bajo la propuesta de caja negra.

UML
Lenguaje Unificado de Modelado (UML, por su sigla en inglés, Unified Modeling Language) es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software. Es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad y es también el estándar oficial, respaldado por el OMG.

XMI
Acrónimo inglés para XML Metadata Interchange. Es un lenguaje de intercambio de modelos del OMG.

XML
Es el acrónimo inglés de eXtensible Markup Language (en español, lenguaje de marcas extensible), es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C). Permite definir la gramática de lenguajes específicos. Por lo tanto XML no es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades. Algunos de estos lenguajes que usan XML para su definición son XHTML, SVG, MathML.

Referencias

[AHM 05] D. H. Akehurst, W. G. Howells, and K. D. McDonald-Maier, Kent Model Transformation Language, Proceedings of Model Transformations in Practice Workshop, MoDELS Conference, Montego Bay, Jamaica, 2005.

[AK 02] D. H. Akehurst and S. J. H. Kent, A Relational Approach to Defining Transformations in a Metamodel, Proceedings of the 5th International Conference on the Unified Modeling Language, Dresden, Germany, pp. 243-258, 2002.

[AKS 03] A. Agrawal, G. Karsai, and F. Shi, Graph Transformations on Domain-Specific Models, Technical Report ISIS-03-403, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37203, 2003.

[ArgoUML] ArgoUML by Tigris.org. http://argouml.tigris.org/

[Art Testing] Myers, G., The Art of Software Testing, Wiley John and Sons, 2004.

[Ant] Apache Ant, http://ant.apache.org/index.html

[ATL] ATL (ATLAS Transformation Language) http://www.eclipse.org/m2m/atl/

[Atom3] Atom3 home page. http://atom3.cs.mcgill.ca/index_html

[BHJ+ 05] Jean Bézivin, Guillaume Hillairet, Frédéric Jouault, Ivan Kurtev, William Piers, Bridging the MS/DSL Tools and the Eclipse Modeling Framework. OOPSLA Workshops, 2005.

[Bei 90] B. Beizer: Software Testing Techniques, Van Nostrand Reinhold, 1990.

[Bei 95] B. Beizer. Black-Box Testing: Techniques for functional testing of software and systems. John Wiley & Sons, Ltd., 1995.

[BM 00] Butler, M. J. and Meagher, M. M. R. Performing Algorithmic Refinement before Data Refinement in B. In: Proc. ZB2000: Formal Specification and Development in Z and B, 2000.

[Booch 04a] Booch, Grady. Saving myself. http://booch.com/architecture/blog. July 22, 2004.

[Booch 04b] Booch, G. et al. An MDA Manifesto. In Frankel, D. and Parodi J. (eds) The MDA Journal: Model Driven Architecture Straight from the Masters, 2004.

[BM 08] Boronat, A. and Meseguer, J.: An Algebraic Semantics for MOF. In: Fiadeiro J. and Inverardi P. (Eds.): FASE and ETAPS 2008, Budapest, Hungary. LNCS, vol. 4961, pp.377–391Springer, April 2008.
[BCR 06] Boronat, A, Carsy J., Ramos, I.: Algebraic specification of a model transformation engine. In: Baresi, Heckel,(eds.) FASE and ETAPS 2006. LNCS, vol. 3922, pp.262–277. Springer, Heidelberg, 2006.
[BW 98] Back, R and von Wright, J., Refinement Calculus: A Systematic Introduction, Graduate texts in Computer Science, Springer Verlag, 1998.
[CESW 04] Tony Clark, Andy Evans, Paul Sammut, James Willans. Applied Metamodelling. A Foundation for Language Driven Development. <http://www. ceteva.com/book.html>, 2004.
[CH 06] Czarnecki K. and Helsen S. Feature-based survey of model transformation approaches. IBM System Journal, Vol. 45, N0 3, 2006.
[CHM+ 02] Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA: visual automated transformations for formal verification and validation of UML models. In: Proceedings 17th IEEE international conference on automated software engineering (ASE 2002), pp. 267–270, Edinburgh, UK, 2002.
[CWM 03] Common Warehouse Metamodel™ (CWM™) Specification, v1.1 formal/2003-03-02
[DB 01] Derrick, J. and Boiten,E. Refinement in Z and Object-Z. Foundation and Advanced Applications. FACIT, Springer, 2001.
[DDH 72] Dahl, O., E. W. Dijkstra and C. A. Hoare. Structured Programming, Academic Press, New York, 1972.
[deLV 02] J. de Lara and H. Vangheluwe, AToM: A Tool for Multi-Formalism and Meta-Modeling. Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering, France, pp. 174–188, 2002.
[DeMarco 79] DeMarco, Tom. Structured Analysis and System Specification. Englewood Cliffs, NJ, 1979.
[Dijkstra 76] Dijkstra, E.W., A Discipline of Programming. Prentice-Hall, 1976.
[EasyMock] http://www.easymock.org
[Eclipse] The Eclipse Project. Home Page. Copyright IBM Corp, 2000.
[EEKR 99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.). Handbook on Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages and Tools. World Scientific, 1999.
[EF 94] Morten Elvang-Gøransson and Robert E. Fields. An Extended VDM Refinement Relation Source. Lecture Notes In Computer Science; Vol. 873 Proceedings of the Second International

Symposium of Formal Methods Europe on Industrial Benefit of Formal Methods Pages: 175 - 189, 1994.
[Egyed 02] Egyed A.: Automated Abstraction of Class Diagrams. ACM Transaction on Software Engineering and Methodology (TOSEM) 11(4), pgs. 449-491, 2002.
[EMF] Eclipse Modeling Framework EMF. http://www.eclipse.org/modeling/emf/
[Epsilon] Epsilon Home page http://www.eclipse.org/gmt/epsilon/
[ER] ER diagram editor for eclipse. http://sourceforge.net/projects/eclipse-erd
[ESC 2] ESCJava2, http://kind.ucd.ie/products/opensource/ESCJava2/
[Fujaba] Fujaba Tool Suite 4, University of Paderborn Software Engineering, http://www.fujaba.de
[GHJV 94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software (ISBN 0-201-63361-2) Addison-Wesley, 1994.
[GLRSW 02] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood, Transformation: The Missing Link of MDA, Proceedings of the 1st International Conference on Graph Transformation, Barcelona, Spain, pp. 90–105, 2002.
[GMF] The Eclipse Graphical Modeling Framework GMF. http://www.eclipse.org/modeling/gmf/
[GR 03] Gerber, Anna and Raymond, Kerry, MOF to EMF: There and Back Again, Proc. Eclipse Technology Exchange Workshop, OOPSLA 2003, Anaheim, USA, pp 66, 2003.
[GS 04] Jack Greenfield, Keith Short, Steve Cook, Stuart Kent. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley, ISBN: 0471202843, 2004.
[GPP 09] Roxana Giandini, Claudia Pons, Gabriela Pérez. A two-level formal semantics for the QVT language. XII Conferencia Iberoamericana en “Software Engineering”, CIbSE 2009. Colombia. ISBN 978-958-44-5028-9, 2009.
[Hoare 69] C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 1969.
[IP 09] Jerónimo Irazábal and Claudia Pons. Model transformation languages relying on models as ADTs. Int Conference on Software Language Engineering (SLE 2009). Denver USA. Lecture Notes in Computer Science. Springer Verlag, 2009.
[JBR 99] Jacobson, I..Booch, G Rumbaugh, J. The Unified Software Development Process. Addison Wesley, 1999.
[JML 2] JML, http://www.cs.ucf.edu/~leavens/JML/
[JET] Eclipse Modeling - M2T – JET. www.eclipse.org/modeling/m2t/?project=jet

[Jones 90] Jones, C.B., Systematic Software Development using VDM,. Prentice Hall. ISBN 0-13-880733-7, 1990.
[JPF 2] Java PathFinder, http://javapathfinder.sourceforge.net/
[JUnit] JUnit testing framework, http://www.junit.org/
[KBC 04] A. Kalnins, J. Barzdins, and E. Celms, Model Transformation Language MOLA, Proceedings of Model Driven Architecture: Foundations and Applications, Linkoeping, Sweden, pp. 14-28, 2004.
[Kermeta] Kermeta home page http://www.kermeta.org/
[KG 06] Milan Karow and Andreas Gehlert. On the Transition from Computation Independent to Platform Independent Models. Proceedings of the 12th Americas Conference on Information Systems, Mexico, 2006.
[Krutchten 00] P. Krutchten. The Rational Unified Process-An Introduction. 2nd Edition, Addison Wesley, 2000.
[Küster 06] Küster, J.M., Definition and Validation of Model Transformations, Journal on Software and Systems Modeling, 5, Springer, pgs 233-259, 2006.
[KWB 03] Kleppe, Anneke G. and Warmer Jos, and Bast, Wim. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
[Lano 96] Lano,K. The B Language and Method. FACIT. Springer, 1996.
[Larman 04] Larman, Craig: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall, third edition, 2004.
[LBM 06] Le Traon, Yves, Baudry, Benoit and Mottu, Jean-Marie, Reusable MDA Components: A Testing- for-Trust Approach, MoDELS 9th International Conference, Lecture Notes in Computer Science 4199, 645-659, 2006.
[LS 05] M. Lawley and J. Steel, Practical Declarative Model Transformation with Tefkat, Proceedings of Model Transformations in Practice Workshop, MoDELS Conference, Montego Bay, Jamaica, 2005.
[MDAG] MDA Guide, OMG Specification. June 2003. http://www.omg.org/docs
[Medini] Medini Home page. http://projects.ikv.de/qvt
[MetaEdit+] MetaEdit+ Modeler http://www.metacase.com/mep/
[MFJ 05] P.-A. Muller, F. Fleurey, and J.-M. Jèzèquel, Weaving Executability into Object-Oriented Metalanguages, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, pp. 264–278, 2005.
[ModelMorf] ModelMorf Home page. http://www.tcs-trddc.com/ModelMorf/index.htm

[MOF] Meta Object Facility (MOF) 2.0 Core Specification. OMG http://www.omg.org/docs
[Mof2Text] MOF Models to Text Transformation Language. OMG Final Adopted Specification. http://www.omg.org/docs/ptc/, 2008.
[Mola] Mola home page http://mola.mii.lu.lv/
[MTF] Model Transformation Framework (MTF), IBM United Kingdom Laboratories Ltd., IBM alphaWorks, http://www.alphaworks.ibm.com/tech/mtf, 2004.
[OCL] OCL 2.0. The Object Constraint Language Specification – for UML 2.0, revised by the OMG, http://www.omg.org, 2004.
[Oldevik 06] Oldevik, Jon. MOFScript User Guide. Version 0.6 (MOFScript v 1.1.11), 2006.
[OMG] Object Management Group (OMG). http://www.omg.org/
[OMGss] OMG MDA Success Stories (web page) http:// www.omg.org/mda/products_success.htm.
[OMG RFP 04] MOF Model to Text Transformation Language. Request For Proposal. OMG Document: ad/2004-04-07, 2004.
[Oslo] OSLO Home page. http://oslo-project.berlios.de/
[Patrascoiu 04] O. Patrascoiu, YATL: Yet Another Transformation Language, Proceedings of the 1st European MDA Workshop, Twente, The Netherlands, pp. 83–90, 2004.
[Peterson 81] Peterson, James Lyle. Petri Net Theory and the Modeling of Systems. Prentice Hall. ISBN 0-13-661983-5, 1981.
[PG 08] Claudia Pons and Diego Garcia. A Lightweight Approach for the Semantic Validation of Model Refinements. Electronic Notes in Theoretical Computer Science (ENTCS). ELSEVIER. ISSN 1571-0661, 2008.
[PGPB 09] C. Pons R. Giandini G. Pérez G. Baum. A Two-level Calculus for Composing Hybrid QVT Transformations. XXVIII Jornadas Chilenas de Computacion. 10-14, Santiago, Chile. IEEE Computer Society, 2009.
[PG 06] Claudia Pons and Diego Garcia. An OCL-based Technique for Specifying and Verifying Refinement-oriented Transformations in MDE. Publication: series Lecture Notes in Computer Science. ISSN 0302-9743. vol. 4199, pp. 645-659, Publisher © Springer-Verlag, 2006.
[Poernomo 08] Iman Poernomo. Proofs-as-Model-Transformations. Proc. of ICMT 2008, LNCS 5063, pp. 214-228. Springer, 2008.
[pUML] pUML group. http://www.cs.york.ac.uk/puml/
[QVT] Query/View/Transformation (QVT) Specification. Final Adopted Specification ptc/07-07-07. OMG. 2007.
[QVTR] Query/View/Transformation Request for Proposal, OMG, 2002.

[Rational Modeler] IBM Rational Software Modeler V7.5 http://www-01.ibm.com/software/awdtools/modeler/swmodeler/
[RS 07] Doug Rosenberg and Matt Stephens. Use Case Driven Object Modeling with UML. Apress, 2007.
[Selic 08] Selic Bran. Manifestaciones sobre MDA. Novatica. Revista de la Asociación de Técnicos de Informática. Nro. 192, 2008.
[Spivey 92] Mike Spivey. The Z Notation: A Reference Manual, 2nd edition, Prentice Hall International Series in Computer Science, 1992.
[Taentzer 03] G. Taentzer, AGG: A Graph Transformation Environment for Modeling and Validation of Software, Application of Graph Transformations with Industrial Relevance (AGTIVE'03) 3062, pp. 446–453, 2003.
[Together] Together Home page. http://www.borland.com/us/products/together/index.html
[U2TP 04] Consortium: UML 2.0 Testing Profile. Final Adopted Specification at OMG (ptc/04-04-02), 2004.
[UL 07] M. Utting and B. Legeard. Practical model-based testing, a tools approach. Morgan Kaufmann Publishers, 2007.
[UML 03] UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification. http://www.omg.org, 2003.
[VP 04] D. Varro´ and A. Pataricza, Generic and Meta-Transformations for Model Transformation Engineering, Proceedings of the 7th International Conference on the Unified Modeling Language, Lisbon, Portugal, pp. 290–304, 2004.
[VJ 04] MTL and Umlaut NG: Engine and Framework for Model Transformation, D. Vojtisek and J.-M. Jèzèquel, Journal title: ERCIM News, Volume 58, 2004
[VVP 02] D. Varró, G. Varró, and A. Pataricza, Designing the Automatic Transformation of Visual Languages, Science of Computer Programming 44, No. 2, 205–227, 2002.
[Willink 03] E. D. Willink, UMLX: A Graphical Transformation Language for MDA, Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Anaheim, CA(2003), pp. 13–24, 2003.
[WK 03] Jos Warmer, Anneke Kleppe. The Object Constraint Language: Getting Your Models Ready for MDA, 2ª edición, Addison Wesley, ISBN: 0321179366, 2003.
[WK 05] Wimmer, Manuel and Kramler, Gerhard. Bridging Grammarware and Modelware. Lecture Notes in Computer Science. Springer Berlin / Heidelberg. Vol. 3844/2006 Satellite Events at the MoDELS 2005 Conference. pg. 59, 2005.
[XP] eXtreme Programming, http://www.extremeprogramming.org/.

Las autoras

Claudia Pons es investigadora del Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET) de Argentina. Obtuvo el grado de Doctor en Ciencias de la Facultad de Ciencias Exactas de la Universidad Nacional de La Plata (UNLP), en 2000. Su área de interés se centra en el modelado de software y los métodos formales. Es miembro del laboratorio de investigación LIFIA y forma parte del plantel de profesores de la Facultad de Informática de la UNLP y de la Universidad Abierta Interamericana (UAI). Actualmente dirige proyectos de investigación y desarrollo y es autora de varios artículos científicos referidos al tema. Ha sido miembro del comité científico de la conferencia internacional MoDELS (Model Driven Engineering Languages and Systems) y presidente del Argentinean Symposium on Software Engineering (ASSE).
Página personal: www.lifia.info.unlp.edu.ar/en/claudia.htm

Roxana Giandini es Doctora en Ciencias Informáticas de la UNLP. Su tesis doctoral abordó el problema de la transformación de modelos en MDD, aportando un nivel de madurez en este tópico. Ejerce la docencia en carreras de grado y posgrado en la Facultad de Informática de la UNLP y en la UAI, donde aplica temas relativos al ámbito de la investigación. Participa y coordina Proyectos en el área de modelado del laboratorio LIFIA de la UNLP. Dirige trabajos finales de grado y tesis de posgrado. Realizó actividades de capacitación y asesoramiento, en el sector público y privado. Actualmente es Miembro del Comité Directivo de CIbSE “Conferencia Iberoamericana en Software Engineering”. Es autora de diversos artículos en Conferencias y Workshops.
Página personal: http://www.lifia.info.unlp.edu.ar/en/roxana.htm

Gabriela Pérez es Licenciada en Informática de la UNLP y actualmente se encuentra trabajando en su tesis doctoral. Es miembro del laboratorio de investigación LIFIA y es docente en cátedras de grado y de postgrado en temas vinculados a la orientación a objetos y al modelado de software. Es autora de artículos presentados en conferencias y workshops relacionados con estos temas. Realiza investigación y desarrollo en tecnologías de modelado, especialmente Eclipse MF; ha participado en proyectos en el IBM Research Center en Yorktown Heights, Nueva York y en 2004 obtuvo el primer puesto en el International Challenge for Eclipse (ICE 2004).

Esta publicación se terminó de imprimir
en el mes de marzo de 2010,
en la ciudad de La Plata,
Buenos Aires,
Argentina.

Con esta Colección, Edulp y la Facultad del Informática de la UNLP aunan recursos para la producción de libros que constituyan no solo una herramienta vital en del desarrollo educativo dentro de la formación académica, sino también un apropiado espacio de difusión del estado actual de las investigaciones en el área de informática.

A lo largo de estos años hemos visto surgir el Desarrollo de Software Dirigido por Modelos (MDD) como una nueva área dentro el campo de la ingeniería de software. MDD plantea una nueva forma de entender el desarrollo y mantenimiento de sistemas de software con el uso de modelos como principales artefactos del proceso de desarrollo. En MDD, los modelos son utilizados para dirigir las tareas de comprensión, diseño, construcción, pruebas, despliegue, operación, administración, mantenimiento y modificación de los sistemas.

En este libro explicamos los fundamentos de MDD, respondiendo a preguntas tales como "¿Qué son los modelos, cómo se construyen y cómo se transforman hasta llegar al código ejecutable?". También nos referimos a los estándares que soportan a MDD y discutimos las ventajas que se introducen en el ciclo de vida del software como consecuencia de adoptarlo.

El libro contiene un ejemplo completo de un desarrollo dirigido por modelos. El desarrollo comienza con la definición de un modelo abstracto expresado en UML y finaliza con el despliegue de un programa ejecutable escrito en Java. La transformación del modelo a código es realizada a través de la aplicación de transformaciones expresadas en un lenguaje estándar. Este ejemplo brinda un panorama completo y comprensible sobre los aspectos técnicos de MDD.

Si bien este libro está dirigido principalmente a estudiantes de carreras de grado y postgrado relacionadas con la ingeniería de sistemas de software, también constituye una lectura recomendable para cualquier persona, con conocimientos básicos sobre sistemas, que esté interesada en incursionar en el desarrollo de software guiado por modelos.

Este libro intenta ser fácil de abordar, ya que presenta los temas de manera autocontenida, gradual, sistemática y recurriendo a numerosos ejemplos.

Profesionales del software con mayor experiencia también pueden beneficiarse con la lectura de este libro, ya que obtendrán un mejor entendimiento de MDD que los ayudará a juzgar cuándo y cómo las ideas de MDD pueden ser aplicadas en sus proyectos.

ISBN 978-950-34-0630-4

Mc Graw Hill **Educación**

edulp Editorial de la Universidad de La Plata