Desarrollo de Software Dirigido por Modelos

Memoria Segundo Semestre. FICYT

Jorge Manrubia Díez

Investigación Dirigida por Juan Manuel Cueva Lovelle

Diciembre de 2006

Desarrollo de Software Dirigido por Modelos

Memoria Segundo Semestre. FICYT

Jorge Manrubia Díez

Investigación Dirigida por Juan Manuel Cueva Lovelle

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

Capítulo 1

INTRODUCCIÓN

1.1. Introducción

¿Es el desarrollo de software un verdadero proceso de ingeniería? En actualidad se acepta que la construcción de software es una actividad cuya naturaleza nada tiene que ver con otras actividades de ingeniería. Es por ello que el intento de trasladar al mundo del software los métodos tradicionales de otras disciplinas fracasó: si el desarrollo de software tiene una naturaleza distinta entonces requiere un proceso de naturaleza distinta. Este hecho no es incompatible con que los desarrolladores de software aspiren a conseguir en sus procesos las mismas cualidades alcanzadas en otras disciplinas, habitualmente enumeradas como listas de factores: corrección, reutilización, trazabilidad, calidad, fiabilidad, mantenimiento, documentación, etc.

Un error recurrente en las primeras décadas de evolución del desarrollo software ha sido no otorgar la importancia que se merece a la actividad de codificación. En una aproximación a otras disciplinas de ingeniería, la codificación se asemejaba a la construcción, y por lo tanto era una tarea predecible si los modelos software eran lo suficientemente elaborados. No ha sido hasta tiempos relativamente recientes cuando se han detectado numerosos problemas en este planteamiento. Bajo ningún aspecto la construcción de software es predecible. La complejidad inherente al software hace que aparezcan cambios en sus requisitos desde el momento que comienza su desarrollo y el acometer los cambios no es sencillo. Al hecho de que los modelos tengan que ser trasladados al código manualmente hay que sumarle el hecho de que el código esté fuertemente impregnado por la amalgama de tecnologías utilizadas en la actualidad. Ambos aspectos entorpecen el desarrollo de software y obligan buscar alternativas que permitan afrontar los cambios con mayor agilidad.

El primer problema significa que sigue existiendo en los procesos de desarrollo de software un componente “artesanal”. Lo único que une a los modelos software con su representación en código fuente es la pericia del programador. Los cambios

que se produzcan en los modelos tienen que ser acometidos manualmente sobre el código fuente y, dada su distinta naturaleza, ambas perspectivas del sistema tienden a alejarse conforme éste crece. El mantenimiento manual de ambas perspectivas, que tienden a separarse cada vez más y a volverse cada vez más complejas, hace que sea imposible garantizar los factores de calidad anteriormente mencionados. El mecanismo más eficiente encontrado hasta la fecha para abordar el problema de cómo llevar los cambios desde que suceden hasta que se codifican ha sido la adopción de modelos de ciclo de vida iterativos. También han resultado exitosas la introducción de prácticas de desarrollo que hacen de la actividad de codificación un proceso más robusto.

El segundo problema, relativo a cómo las tecnologías impregnan el código fuente que representa a los modelos software, presenta también serias consecuencias. En primer lugar, hace que la brecha existente entre modelos y código se incremente. Si en los modelos es sencillo separar el modelo funcional del modelo tecnológico de la aplicación, en el código fuente esta tarea se torna imposible. Esto hace que la única representación de los programas que puede ejecutarse, su código fuente, quede inexorablemente ligada a plataformas tecnológicas concretas. En un ámbito donde los artefactos tecnológicos se renuevan en cuestión de pocos años, este problema afecta directamente al mantenimiento y reutilización del software a medio-largo plazo. Por otra parte, este problema afecta directamente a la eficiencia y fiabilidad de la actividad codificación. Un código fuente donde se entremezclan módulos de funcionalidad correspondientes a diferentes dominios resulta difícil de manipular manualmente de una manera eficiente. En los últimos años han surgido interesantes iniciativas encaminadas a solucionar este problema, como por ejemplo la separación de incumbencias y la programación orientada a aspectos [Kiczales et al., 1997], la utilización de objetos sencillos[1] sobre los que se añaden servicios externamente o la Inversión de Control (IoC) [Fowler, 2004].

En este contexto, se está produciendo un nuevo movimiento en el ámbito del desarrollo software denominado "dirigido por modelos" o *Model Driven Development* (MDD). Lo que se plantea en este movimiento es elevar un peldaño el nivel de abstracción con respecto a cómo se desarrolla software en la actualidad. Para conseguir el nuevo nivel de abstracción, este movimiento plantea que los modelos software sean artefactos de desarrollo de primer nivel. Estos modelos serán precisos, completos y no ambiguos, de tal modo que su traducción a código fuente ejecutable pueda ser, total o parcialmente, realizada de forma automática. Del mismo modo que un compilador puede traducir de forma automática código fuente escrito en un lenguaje de programación de alto nivel a código máquina entendible por un ordenador, sería posible disponer de herramientas que transformasen modelos formales a código fuente de forma automática. De materializarse esta propuesta, haría que el modo en el que desarrollamos software hoy fuese visto como un proceso en extremo rudimentario, del mismo modo que hoy consideramos la manera en la que se desarrollaba software hace 4 décadas una disciplina primitiva.

[1] Está extendido el convenio de denominar a dichos objetos POJO (Plain Old Java Object), aun cuando no se trate de la plataforma Java. Puede encontrarse su motivación en [Fowler, 2000].

Dentro del movimiento del desarrollo de software dirigido por modelos, ha surgido con fuerza una iniciativa denominada *Model Driven Architecture* (MDA). La razón es que tiene detrás un consorcio como el *Object Management Group* (OMG) y está edificada sobre un conjunto de estándares, entre los que destaca UML. Esta propuesta ha generado fuertes reacciones en la comunidad, tanto positivas por lo ambicioso de sus objetivos, como negativas y extremadamente escépticas, por la inexistencia en la actualidad de tecnologías y especificaciones lo suficientemente maduras como para construir y transformar en código modelos software precisos, formales y de propósito general.

1.2. Objetivos de la Investigación

Esta investigación partirá de un análisis del estado actual de propuestas para el desarrollo dirigido por modelos. Se comenzará analizando la propuesta MDA, estudiando sus aspectos positivos y sus carencias. A continuación se analizarán otras aproximaciones al desarrollo por modelos y tecnologías relacionadas. El objetivo en última instancia es descubrir qué hace falta para convertir la promesa planteada por el paradigma MDD en una realidad. Este objetivo se materializará a través de los siguientes subobjetivos:

1.2.1. Estándares y especificaciones MDD

La propuesta de MDA se edifica sobre un conjunto de estándares que han sido desarrollados en el seno del OMG. Estos estándares a menudo son extremadamente grandes y complejos y carecen de soporte comercial real. En esta investigación se analizarán los puntos fuertes y débiles encontrados en los estándares MDA con el fin de descubrir qué modificaciones o enfoques de uso pueden hacerlos más adecuados para hacer realidad la iniciativa. Para ello, además de analizarse la especificación de lo estándares en sí, se prestará una especial atención al uso que se ha hecho de ellos tanto en el ámbito académico como en comercial.

Dentro de otras propuestas MDD existen aproximaciones complementarias a MDA para construir sistemas basados en un enfoque de desarrollo dirigido por modelos. También existen aproximaciones que difieren radicalmente del planteamiento MDA. En esta investigación se analizarán con detalle los fundamentos y especificaciones de estas propuestas.

1.2.2. Enfoques de Implementación MDA

El movimiento MDA ha surgido recientemente (fue propuesto en el 2001). Como se ha mencionado, sus ambiciosos objetivos han provocado y siguen provocando fuertes reacciones en la comunidad de la ingeniería del software. Además, muchos de sus principales estándares tienen una extensión considerable, y permiten diferentes usos e interpretaciones. Por ello, no es de extrañar que existan en torno a MDA

diferentes posturas acerca de cómo implementar la iniciativa. En la investigación se identificarán los razonamientos que justifican cada postura y se recogerán los mejores aspectos de cada aproximación.

1.2.3. Especificación y Manipulación de Modelos

En MDD los artefactos de desarrollo más importantes son los modelos y, por ello, dos de los aspectos más importantes de la iniciativa hacen referencia a cómo pueden modelarse los diferentes aspectos de un sistema software y, además, cómo puede hacerse que estos modelos puedan ser manipulados por un sistema informático de forma automatizada. Ambos aspectos serán analizados con detalle en la investigación.

Con respecto al modelado software, un aspecto especialmente conflictivo es cómo se modela el comportamiento de los modelos de los sistemas, teniendo en cuenta que este modelado debe realizarse a un elevado nivel de abstracción y con una semántica los suficientemente precisa como para permitir que la generación automática de código. En la investigación se prestará una especial atención a las diferentes aproximaciones que existen para modelar el comportamiento de los sistemas para concluir cual es el enfoque o cómo pueden combinarse los mejores aspectos de cada aproximación para permitir un modelado de comportamiento efectivo.

1.2.4. Transformación de Modelos

Además de la especificación de modelos, el otro componente fundamental de MDD hace referencia a cómo pueden transformarse los modelos en el código fuente de programas ejecutables. En general, se hablará de la transformación de unos modelos en otros, considerando el código fuente como el resultado de una fase de transformación modelo a texto de los modelos refinados en anteriores etapas. En el ámbito MDA, se han producido recientemente importantes modificaciones en la estandarización realizada por el OMG.

La transformación de modelos y la generación automática de código comprenden disciplinas complejas y que admiten diferentes técnicas y aproximaciones. Dado su papel fundamental en MDD, este área será analizada también con detalle en esta memoria.

1.3. Organización de Esta Memoria

La presente memoria comprende los capítulos que a continuación se enumeran. Al finalizar cada capítulo, se recogerán las principales conclusiones del mismo:

- En el Capítulo 2 se analizarán los principales conceptos manejados en la iniciativa MDD. Además, se estudiarán diversos problemas existentes en los procesos de desarrollo software actuales y se analizará cómo podría una solución MDD darles respuesta.

- En el Capítulo 3 se estudiarán los aspectos relacionados con la especificación y manipulación de modelos en MDD. Para ello, se analizarán con detalle las dos especificaciones que forman el corazón de la propuesta MDA: UML y MOF.
- En el Capítulo 4 se analizará el estado actual de propuestas para la transformación de modelos. Se analizarán contemplando el ciclo de vida completo que debe seguir un proceso de transformación. Esto es, transformaciones modelo a modelo y transformaciones modelo a texto.
- En el Capítulo 5 se estudiarán diversas implementaciones reales de sistemas MDD que existen en la actualidad.
- En el Capítulo 6 se recogerán las conclusiones generales de la investigación.
- Finalmente, se incluyen tres anexos correspondientes a un listado de acrónimos, la bibliografía y el índice de esta Memoria.

Capítulo 2

DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS

2.1. La Propuesta de MDA

A finales del año 2000, el OMG presentó la iniciativa *Model Driven Architecture* (MDA) como una propuesta para el desarrollo de software utilizando modelos [Soley and *OMG Staff Strategy Group*, 2000]. La esencia de esta iniciativa es que el proceso de crear software debería ser dirigido por la formulación de modelos en lugar de por la escritura manual de código fuente.

Con este fin, se definió una arquitectura de estándares y un conjunto de directrices que permiten expresar las especificaciones de software como modelos. Estos modelos representarían todos los aspectos del programa final ejecutable, por lo que deberían ser formales, precisos y con una semántica bien definida.

Una de las principales ventajas que tendría un proceso de desarrollo dirigido por modelos es la independencia tecnológica. En la actualidad, el desarrollo de aplicaciones está fuertemente impregnado de las decisiones tecnológicas que se adopten. Éstas abarcan opciones *middelware*, *frameworks* arquitectónicos, servicios de persistencia y transaccionales, infraestructuras de seguridad o los lenguajes de programación utilizados. Los programas quedan inexorablemente ligados a las tecnologías que utilicen, y las tecnologías en el mundo de la informática son extremadamente volátiles. La iniciativa MDA y los estándares que engloba tienen como objetivo situar los modelos en un nivel de abstracción tal que su realización en múltiples plataformas sea posible.

Además de los modelos, el otro componente fundamental de MDA son las transformaciones. El proceso de transformar una especificación software en un programa ejecutable sería automático. El código fuente de las aplicaciones se debería generar a partir de los modelos en un proceso de transformación, del mismo modo que el

código máquina se genera a partir del código fuente escrito en un lenguaje de alto nivel en un compilador tradicional.

La iniciativa MDA apuesta en definitiva por elevar el nivel de abstracción a la hora de desarrollar software. Los únicos artefactos que deberían manejar los desarrolladores son los modelos que especifican la aplicación. La actividad de modelar software debería conducir los procesos de desarrollo y los modelos de los sistemas deberían poder ser transformados automáticamente en una aplicación completamente funcional para una plataforma de ejecución determinada.

2.1.1. Conceptos Básicos

A continuación se recoge una descripción de los principales conceptos que conforman el núcleo de MDA. El objetivo es introducir los principales términos utilizados en la iniciativa MDA del OMG, que serán de uso frecuente a lo largo de esta Memoria.

2.1.1.1. Modelo

En el contexto de MDA, los modelos son representaciones formales del funcionamiento, comportamiento o estructura del sistema que se está construyendo [Raistrick et al., 2004]. El término "formal" especifica que el lenguaje utilizado para describir el modelo debe de tener una semántica y sintaxis bien definida.

Un concepto clave a la hora de construir modelos es el de *"abstracción"*. Hace referencia al hecho de ignorar la información que no es de interés para el dominio del problema a resolver. Los modelos representan elementos físicos o abstractos correspondientes a una realidad simplificada, adecuada al problema que se necesita resolver.

2.1.1.2. Metamodelo

El término metamodelo se utiliza para identificar el modelo de un lenguaje de modelado [Mellor et al., 2004]. Definirá la estructura, semántica y restricciones para la familia de modelos que puede ser construida con dicho lenguaje de modelado.

Los metamodelos dan respuesta al requisito de formalidad que MDA impone a los lenguajes de modelado. Permiten especificar sin ambigüedad los conceptos de los lenguajes utilizados para especificar modelos. En particular, el disponer de una especificación precisa de diferentes modelos a diferentes niveles de abstracción, habilita una operación clave en MDA: la transformación automática entre ellos por parte de un computador (§ 2.1.1.6).

2.1.1.3. Plataforma

En MDA, el término "plataforma" se utiliza generalmente para hacer referencia a los detalles tecnológicos y de ingeniería que no son relevantes de cara a la funcionalidad esencial del sistema. Esto uso del término se encuadra dentro de la

separación fundamental que se realiza en MDA entre la especificación de un sistema y su plataforma de ejecución.

Se define plataforma como la especificación de un entorno de ejecución para un conjunto de modelos [Mellor et al., 2004]. Estará formada por un conjunto de subsistemas y tecnologías que proporcionan una funcionalidad determinada a través de una serie de interfaces y patrones de uso determinados[Miller and Mukerji, 2003]. Son ejemplos de plataformas *Java Platform, Enterprise Edition* (Java EE) [Sun, 2005], *Common Object Request Broker Architecture* (CORBA) [OMG, 2004a] o .NET [Thai and Lam, 2002].

2.1.1.4. Modelo Independiente de la Plataforma (PIM)

Se denomina *Platform Independent Model* (PIM) a una vista del sistema centrada en la operación del mismo que esconde los detalles necesarios para una determinada plataforma [Miller and Mukerji, 2003]. Esta vista muestra la parte correspondiente de la especificación del sistema que no varía al cambiar de una plataforma a otra. El PIM contendrá un grado de independencia tal que permita su utilización en diferentes plataformas de características parecidas.

Tal y como se ha visto en § 2.1.1.3, el término plataforma abarca tanto los sistemas operativos como las realizaciones de plataformas software que sobre ellos se implementen. Es habitual que estas plataformas, como .NET o Java EE, ejecuten los sistemas sobre una máquina virtual [Howe, 2002]. Estas máquinas virtuales son especificaciones de un procesador computacional, que pueden ser realizadas sobre múltiples plataformas computacionales, siendo típicamente su desarrollo lógico[Ortín Soler, 2001]. Las máquinas virtuales consiguen de este modo la independencia de de los programas que ejecutan con respecto a la plataforma de ejecución final (habitualmente un sistema operativo funcionando sobre una configuración hardware determinada). Es importante señalar que MDA propone un nivel de abstracción mayor, puesto que se buscará la independencia de la plataforma y dichas máquinas virtuales serán consideradas plataformas de ejecución.

En la Figura 2.1 se recoge un escenario de utilización de MDA que ilustra diferentes niveles de abstracción y los artefactos de ejecución manejados en cada uno de ellos. En este escenario típico, una herramienta MDA recoge los diferentes PIM que especifican el sistema y los transforma en un lenguaje de alto nivel soportado por la plataforma de destino. En el caso de que la plataforma destino utilice una máquina virtual que no sea un intérprete puro [Cueva Lovelle, 1998], se requerirá un proceso de compilación que obtenga el código máquina soportado por la máquina virtual. Este es el caso de las plataformas Java y .NET. Es habitual que, por razones de eficiencia, dicha máquina virtual sea compilada en cada sistema operativo como código máquina nativo, que será ejecutado finalmente por el hardware utilizado en cada caso.

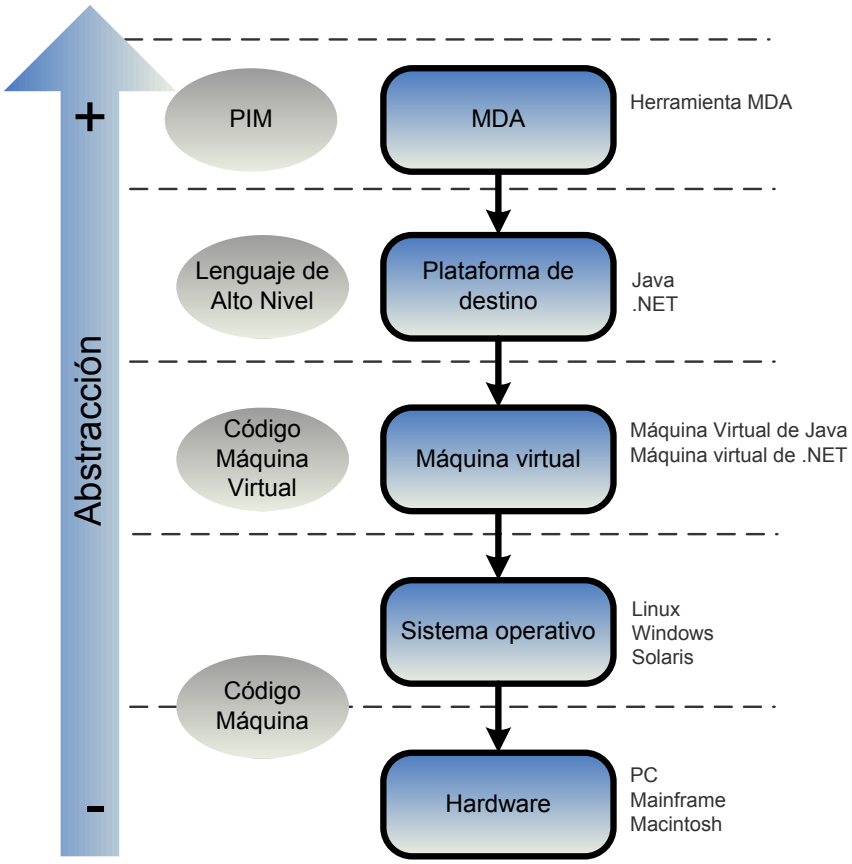

Figura 2.1: Ejemplo de niveles de abstracción e independencia con respecto a la plataforma en MDA.

2.1.1.5. Modelo Específico a la Plataforma (PSM)

Un *Platform Specific Model* (PSM) es una vista del sistema desde el punto de vista de la plataforma específica de ejecución [Miller and Mukerji, 2003]. Combina los diferentes PIM que especifican el sistema con los detalles acerca de cómo utiliza un tipo concreto de plataforma.

La diferencia entre un modelo PSM y uno PIM es el conocimiento que incorpora acerca de la plataforma final de implementación. Un PIM no incorpora ninguna información acerca de la plataforma de destino. Por su parte, un PSM contiene toda la información necesaria para hacer posible su implementación sobre una plataforma concreta. Es decir, contendrá toda la información necesaria para generar el código de la aplicación a partir de él [Mellor and Balcer, 2002].

Existen autores que consideran el código fuente de la aplicación un modelo PIM [Raistrick et al., 2004]. No obstante, en la propuesta del OMG el código fuente de la aplicación se considera un resultado de una transformación sobre el PIM [Miller and Mukerji, 2003]. Esta implementación del programa en un lenguaje de programación determinado se conoce en ocasiones como *Platform Specific Imple-*

mentation (PSI).

En relación con el concepto de PSM aparece el concepto de *"modelo de la plataforma"* [Miller and Mukerji, 2003]. Será el modelo que represente las diferentes abstracciones que conforman la plataforma final, así como los servicios que proporciona ésta. Estos conceptos podrán ser utilizados para construir los PIM, a la hora de especificar el uso que hace la aplicación de la plataforma sobre la que se ejecuta.

Aunque en la guía de referencia de MDA del OMG [Miller and Mukerji, 2003] se realiza una separación clara entre los dos tipos de modelos manejados en MDA: PIM y PSM, algunos autores consideran que la distinción no siempre es posible [Kleppe et al., 2003]. Esto es debido a que los modelos de las aplicaciones suelen contener información relacionada con la plataforma o tipo de plataforma sobre la que se ejecutarán. Por ejemplo, al especificar que un método de una clase es transaccional, puede considerarse que el modelo se convierte en específico a una plataforma tecnológica que soporte transacciones.

2.1.1.6. Transformación

En MDA se contemplan dos tipos fundamentales de transformaciones (Figura 2.2):

- La transformación de un PIM en un PSM.
- La transformación de un PSM en el código fuente de la aplicación.

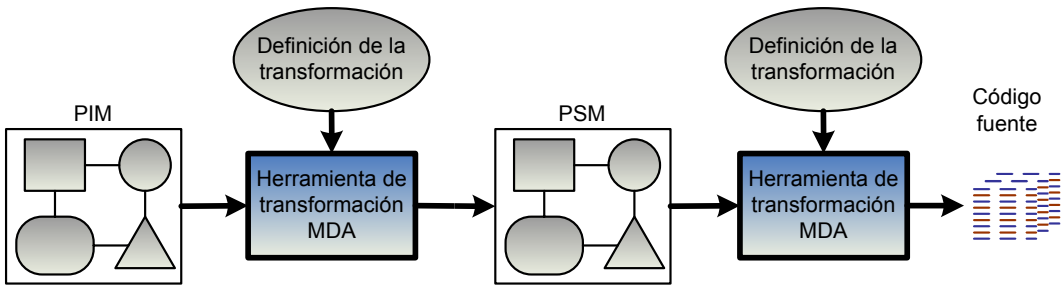

Figura 2.2: Transformaciones y herramientas de transformación.

En la Figura 2.2 se representa esta doble transformación. Una herramienta de transformación MDA recogerá el modelo independiente de la plataforma PIM y lo transformará en un modelo específico a la plataforma PSM. Para realizar esta transformación, hará uso de una metainformación que especifique cómo realizar la transformación. Esta metainformación se denomina "definición de la transformación" [Kleppe et al., 2003] o *mapping rules* (reglas de mapeo) [Miller and Mukerji, 2003, Mellor et al., 2004]. En general, suele utilizarse el término *"mapping"* (mapeo) para denominar "transformación".

Además de las definiciones de transformación, es habitual etiquetar o marcar los modelos para guiar los procesos de transformación. Por ejemplo, para diferenciar

datos persistentes de datos temporales. O para distinguir entre procesos remotos o locales. Esta metainformación se denomina genéricamente como "marcas" (*marks*) [Mellor et al., 2004]. Las marcas se utilizan tanto en los PIM como en los PSM.

Las dos transformaciones pueden ser realizadas por la misma herramienta de transformación o por herramientas distintas. Es precisamente cómo se aborda la construcción de dichas herramientas de transformación, junto con las definiciones de cómo se realizan las transformaciones, uno de los aspectos claves que diferencian las diferentes líneas de investigación en torno al paradigma MDA[1].

2.1.2. Impacto de MDA en los Procesos de Desarrollo

Una manera habitual de ilustrar el gran avance que supondría MDA en la la historia de la ingeniería del software es analizar su impacto en el proceso de desarrollo de software. Los autores que defienden la necesidad de esta iniciativa, consideran que el desarrollo de herramientas MDA es un requisito indispensable para convertir el proceso de crear software una verdadera disciplina de ingeniería [Frankel, 2003, Kleppe et al., 2003, Raistrick et al., 2004].

2.1.3. Proceso de Desarrollo Tradicional

2.1.3.1. La Iteración como Respuesta al Cambio

En la Figura 2.3 se representan las diferentes etapas generales que habitualmente se siguen a la hora de desarrollar una aplicación:

1. Un análisis de los *requisitos* de la aplicación.

2. La construcción de un *modelo de análisis* que, de una manera formal, especifique qué funcionalidad debe contener la aplicación.

3. El paso a un *modelo de diseño* que recoja con detalle los diferentes componentes del sistema informático que satisfaga los requisitos de análisis.

4. La escritura del *código fuente* que realice el modelo de diseño construido en una plataforma tecnológica determinada.

5. El *despliegue* del sistema operativo en un entorno de ejecución.

Estas etapas generales son básicamente las mismas desde que fuesen formalizadas en el proceso de desarrollo propuesto por Royce en 1970 [Royce, 1970]. Desde entonces se han producido avances sustanciales en el ámbito de los procesos de desarrollo. Entre éstos, destacan los relativos a los *modelos de ciclos de vida* que sustentan los diferentes procesos.

[1] De hecho, en la Guía de Referencia de MDA [Miller and Mukerji, 2003], la información que especifica la transformación se representa con una caja blanca vacía.

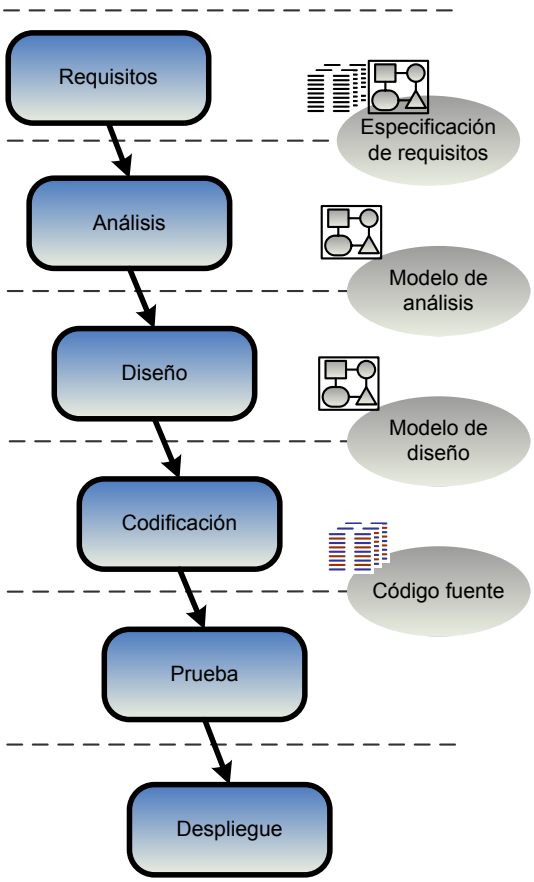

Figura 2.3: Etapas en el modelo de desarrollo de software tradicional.

El modelo de ciclo de vida propuesto por Royce se conoce en la actualidad como modelo de ciclo de vida en cascada (*waterfall model*). En dicho modelo las diferentes etapas de desarrollo se suceden en secuencia, con una iteración nula entre etapas o reducida a la etapa inmediatamente anterior. Este modelo de ciclo de vida se dio por fracasado, entre otras razones, por su dificultad para afrontar los cambios que se produjesen durante el proceso de desarrollo, en especial los cambios en los requisitos.

Debe señalarse que Royce en su propuesta [Royce, 1970] sí contempló la necesidad de iteración entre etapas y el hecho de que esta iteración no necesariamente debe producirse entre etapas consecutivas. Royce consideró que los cambios en las fases de análisis y codificación tenían un impacto mínimo en el resto de etapas. La etapa fundamental sería la de diseño, y en base a ella, propuso una serie de recomendaciones para afrontar los cambios que se produjesen en el análisis de requerimientos y en el diseño. De una manera un tanto injusta, su propuesta suele recordarse en la actualidad únicamente por los aspectos negativos del modelo de ciclo de vida en cascada y no cómo uno de los primeros intentos de formalización de un proceso de desarrollo en la historia de la ingeniería del software.

Tras el modelo de ciclo de vida en cascada, se sucedieron muchos otros. Entre ellos, el modelo en espiral de Boehm [Boehm, 1988], el evolucionario [Gilb, 1988, May and Zimmer, 1996] y el modelo incremental [Sommerville, 2001].

Todos estos modelos tienen en común una apuesta por la iteración cómo herramienta principal para abordar los cambios en el software. Se abandona la idea de poder desarrollar completamente cada etapa en una sola iteración. En especial, se acepta que los cambios en los requisitos del software son inevitables [Beck, 1999], por los que el proceso de desarrollo debe estar preparado para absorberlos (Figura 2.4).

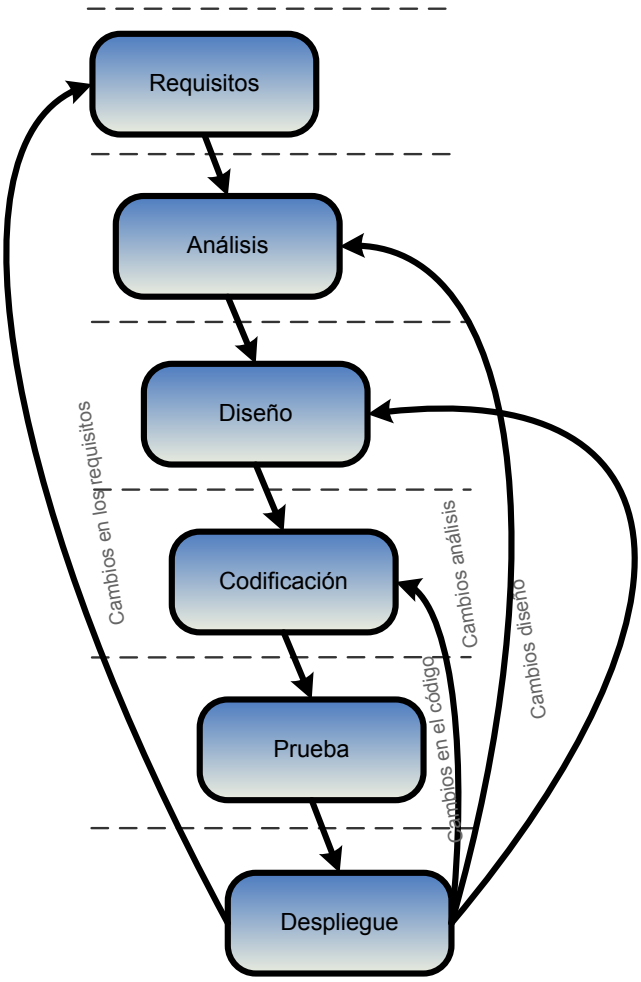

Figura 2.4: Proceso de desarrollo con un ciclo de vida iterativo.

Uno de los métodos iterativos de desarrollo orientado a objetos más populares es el *Unified Proccess* (UP) o Proceso Unificado [Jacobson et al., 1999]. La implementación más adoptada de este método es el *Rational Unified Proccess* (RUP) [Kruchten, 2000].

2.1.3.2. Métodos Ágiles

Tras adoptar los métodos de desarrollo un modelo de ciclo de vida iterativo, a la hora de su puesta en práctica se detectó otro problema, relacionado con los artefactos producidos y consumidos entre las diferentes etapas. En las fases previas a la codificación se producen una gran cantidad de artefactos en forma básicamente de documentos de texto y diagramas, pero estos están desconectados entre sí y del código fuente que representan [Kleppe et al., 2003].

Conforme los sistemas van siendo modificados, la distancia existente entre el código y los artefactos de modelado previamente construidos se incrementa. Ante esta situación se presentan dos alternativas opuestas:

- En primer lugar, puede retrocederse a la fase de desarrollo apropiada y acometer el cambio. El problema de esta aproximación son los cambios en cascada que se producen en las fases subsiguientes y en los artefactos producidos en cada una de ellas. Es lo que se ha denominado un "exceso de burocracia" de algunos procesos de desarrollo [Fowler, 2005c].

- En segundo lugar, puede optarse por realizar los cambios únicamente en el código. Esta es una opción a menudo adoptada en la práctica, cuando no se dispone de tiempo para modificar los artefactos de modelado. Como consecuencia, los productos obtenidos de las primeras fases de desarrollo van progresivamente perdiendo valor. Dejan de representar una especificación exacta del código. Las tareas de análisis, modelado y documentación previas a la codificación dejan de considerarse imprescindibles. Sin embargo, tras décadas de experiencia en el desarrollo de software se acepta que, ante proyectos de cierta envergadura, dichas tareas deben realizarse [Kleppe et al., 2003].

Desde mediados de los 90 se produjo un movimiento en el campo de la ingeniería del software que intentó en parte mitigar los problemas mencionados: la aparición de los "métodos ágiles" (*agile methods*) englobados bajo lo que se denominó "desarrollo de software ágil" (*agile software development*) [Larman, 2003].

Suele decirse que la aparición de los métodos ágiles fue provocada en parte como reacción a los "métodos pesados" (*heavyweight methods*)[2]. Éstos comprenden los procesos de desarrollo rígidos, fuertemente regulados y que realizan un gran énfasis en la planificación, fuertemente influenciados por otras disciplinas de ingeniería [Fowler, 2005c]. En ocasiones también se denomina a estos métodos "métodos predecibles" (*predictive methods*) [Larman, 2004] por su motivación de intentar planificar con gran detalle largos plazos del proceso de desarrollo. Frente a este enfoque, los métodos iterativos en general, y los métodos ágiles en particular, también se denominan "métodos adaptativos". En vez de intentar evitar el cambio, prevén aceptarlo cuándo se produzca, adaptando el software y el proceso en sí mismo para ello.

[2]Por ello, también se conoce a los métodos ágiles como "métodos ligeros". Debe señalarse que la *Agile Alliance*[Agile Alliance, 2006] constituida en febrero de 2001 adoptó como término oficial "ágil".

Los diferentes métodos ágiles comparten una serie de características comunes [Larman, 2004]:

1. Acortan el plazo de iteración y apuestan por un desarrollo evolutivo. La forma final del programa será el resultado de sucesivas modificaciones a partir de un entregable inicial.

2. Utilizan una planificación adaptativa. La propia planificación del proceso se modifica conforme se avanza en el desarrollo, para adaptarse a las circunstancias que en cada momento concurran.

3. Apuestan por la entrega incremental. Las nuevas características y modificaciones sobre el sistema se integran cuanto antes en el sistema entregable.

4. Incorporan diversas prácticas de desarrollo que buscan permitir una respuesta rápida y flexible a los cambios.

El método ágil más conocido es *Extreme Progamming* (XP) [Beck, 1999]. Como ejemplo de métodos ágiles anteriores a XP pueden citarse Scrum [Schwaber and Beedle, 2001] y la familia de métodos Crystal [Cockburn, 2004]. Con respecto al UP, el hecho de considerarlo pesado o ágil dependerá de su implantación concreta, pues realmente se trata de un framework de procesos que debe adaptarse a cada organización y proyecto en concreto. Como señala Martin Fowler en [Fowler, 2005c] en la industria pueden encontrarse utilizaciones del UP que van desde rígidos modelos en cascada a instancias perfectamente ágiles.

Los métodos ágiles también han sido objetivo de múltiples críticas habiéndose creado cierta controversia en torno a ellos. Si bien existe un consenso generalizado acerca de la idoneidad de las buenas prácticas sobre las que descansan, se han señalado algunos problemas con los mismos, en especial con XP. Entre ellos pueden destacarse: la faltan de datos empíricos que prueben su efectividad y su no escalabilidad a la hora de su aplicación en proyectos grandes [Turk et al., 2002, Beust, 2006], la excesiva dependencia del conocimiento tácito de los miembros del equipo de desarrollo debido a la ausencia de una documentación formal [Cockburn, 2001], o el alto acoplamiento entre las diferentes prácticas de XP y la falta de documentación de sus interrelaciones [Vanderburg, 2005].

Un error habitual es caracterizar XP como un método anárquico donde lo único importante es el código de los programas. Así, en [Kleppe et al., 2003] se llega a afirmar en relación a XP:

> *One of the reasons for this is that it acknowledges the fact that the code is the driving force of software development. The only phases in the development process that are really productive are coding and testing.*

Realmente, para XP la actividad más importante es la de diseño. En el "Manifiesto Ágil" (*Agile Manifesto*) [Beck et al., 2001], en el principio número 10 se señala que la excelencia técnica y los buenos diseños aumentan la agilidad. Lo que sucede es que los métodos ágiles consideran la dificultad de transformar los diseños en código y proponen diversas prácticas para que ésta transformación sea eficaz.

2.1.3.3. Modelado y Codificación

Un error que ha estado presente en multitud de enfoques metodológicos de desarrollo software ha sido obviar la importancia de la actividad de codificación. En una aproximación a otras disciplinas de ingeniería asentadas, se equiparó la codificación a la fase de construcción de cualquier otra ingeniería. Así, Winston W. Royce afirmaba en 1970 [Royce, 1970], con relación a la fase de codificación:

> *(. . .) generally these phases are managed with relative ease and have little impact on requirements, design, and testing.*
>
> *(. . .) If in the execution of their difficult and complex work the analysts have made a mistake, the correction is invariably implemented by a minor change in the code with no disruptive feedback into the other development bases.*

Martin Fowler profundiza en este aspecto [Fowler, 2005c] analizando las diferencias de los procesos de desarrollo de software con los procesos de construcción de otras ingenierías. En éstas, la actividad de diseño es la más difícil de predecir y requiere la gente con mayor formación y creatividad. La actividad de construcción es más fácil de predecir, requiere gente con menos formación y es, en general, mucho más costosa en términos económicos que el diseño. Una vez que se tiene el diseño, puede planificarse la construcción y ejecutarse ésta de una manera predecible.

Trasladar este modelo al desarrollo de software significaría volver al modelo en cascada, cuyos inconvenientes ya se han mencionado. Martin Fowler añade a estos inconvenientes una interesante pregunta[Fowler, 2005c]: *¿Puede obtenerse un diseño que haga de la codificación una actividad de construcción predecible?. Y si esto es posible ¿será rentable en términos de costes?*. Esta cuestión ya no hace referencia al intento de capturar todos los requisitos en una iteración, sino a la dificultad de transformar los modelos en código.

Las razones de esta dificultad serán analizadas en los capítulos posteriores de esta Memoria. Entre otras pueden señalarse la inmadurez de los lenguajes y métodos de modelado software, la imposibilidad de probar los modelos, la impedancia que existe a la hora de codificar un modelo software con los lenguajes de programación existentes o las dificultades que añade a la codificación la necesidad de considerar los aspectos tecnológicos de la plataforma final de ejecución.

Estas dificultades han llevado a algunos autores a considerar el código fuente como parte de la especificación del software. Bajo este enfoque, Jack Reeves propone [Reeves, 1992] considerar el código fuente como un documento más de diseño y que realmente la fase de construcción del software la realizan el compilador y el enlazador (*linker*).

En este sentido, Steve Cook afirma [Cook, 2004] que la única forma de describir aplicaciones hasta la fecha ha sido el código fuente. Esto ha impedido hacer que el desarrollo de software puede ser implementado como una cadena de producción, donde cada participante acepta una entrada en forma de activos de información,

añade sus conocimientos y pasa los resultados a los siguientes participantes en la cadena. Como resultado, afirma que las únicas personas que están profundamente involucradas en la producción de una aplicación son los programadores.

Las anteriores reflexiones reflejan un problema fundamental en el modelado de software que afecta sustancialmente a los procesos de desarrollo. Este problema es la gran distancia existente entre:

- Los modelos software, es decir, las descripciones del software realizadas a un elevado nivel de abstracción (§ 2.1.1.1).
- El código fuente que comprende la aplicación y que, en última instancia, realiza los modelos desarrollados sobre una plataforma tecnológica concreta.

Ante este problema, un enfoque pragmático y con gran aceptación en la actualidad es el de los métodos ágiles. Éstos le dan una gran importancia al modelado, pero entendido como una herramienta de comunicación, no de documentación [Larman, 2004]. Bajo este enfoque, no se construyen modelos detallados del software ni tampoco se modelan todas las partes de éste. Los problemas más sencillo de diseño se resolverán directamente en la fase de codificación. Los modelos se centran en especificar las partes más complejas o conflictivas de la aplicación.

Esta filosofía de modelado fue bautizada como "modelado ágil" (*agile modelling*) en 2002 [Ambler and Jeffries, 2002], aunque ya había sido puesta en práctica por los autores del movimiento ágil desde mucho antes. Entre los autores que defienden este enfoque se encuentran destacados ingenieros como Martin Fowler [Fowler, 2003e], Craig Larman [Larman, 2004] y Kent Beck [Beck, 1999].

Otro enfoque distinto al problema es apostar por la generación automática de código a partir de los modelos que especifican el software. MDA utiliza este planteamiento.

2.1.3.4. Problemas en los Procesos de Desarrollo Actuales

Tal y como se ha visto en el apartado anterior, la desconexión entre los artefactos de modelado, y entre éstos y el código fuente del programa, hace que el coste de un modelado formal, detallado e integral dificulte e incluso desaconseje su realización.

Pero el no realizar un modelado formal plantea un importante problema de **mantenimiento**. Si no se dispone de una documentación formal, se produce inexorablemente una dependencia del conocimiento tácito del equipo de desarrollo. Éste conocimiento se define como la suma del conocimiento de todas las personas del equipo que desarrolla el proyecto [Cockburn, 2001].

Una vez superado el periodo de desarrollo principal del software, es habitual que los equipos de desarrollo reduzcan su tamaño. Con lo que el conocimiento tácito se muestra insuficiente para el mantenimiento del software[3]. Por otra parte,

[3]Para solucionar este problema en el caso de XP, Cockburn propone planificar el desarrollo de documentación utilizando *"story cards"* como si fuesen nuevas características del software a desarrollar.

es habitual que el cliente exija una documentación formal que describa el diseño del sistema, como mecanismo de protección contra la desaparición precisamente del conocimiento tácito del equipo de desarrollo.

Por otra parte, puede señalarse una serie de problemas comunes a los métodos de desarrollo actuales. Centrándonos en los métodos con un ciclo de vida iterativo, que son los que gozan de una mayor aceptación en la actualidad, se detectan problemas de:

Productividad. Existe una gran redundancia provocada por el mantenimiento manual de los entregables del proceso [Raistrick et al., 2004]. Cada elemento del modelo de análisis aparece de nuevo en el modelo de diseño y cada elemento del modelo de diseño aparece de nuevo en el código fuente. Con unos requisitos cambiando constantemente, esta redundancia puede entorpecer enormemente la transición entre fases y, con ello, el proceso de desarrollo.

Con respecto a la transición entre las etapas de análisis y diseño, para distinguir ambas fases a menudo se dice que el análisis representa el "qué" y el diseño el "cómo" [Booch, 1993]. También que el análisis trata sobre hacer lo correcto y el diseño con hacerlo correctamente [Larman, 2004]. Se acepta que:

- El *análisis* hace énfasis en la investigación del problema y de sus requisitos, en lugar de en su solución.
- El *diseño* hace énfasis en una solución conceptual, a nivel tanto de software como de hardware, que cumple todos los requerimientos. No entra en su ámbito la implementación de la solución.

Sin embargo, en la práctica, la frontera entre lo que se considera análisis y diseño es difusa. De hecho, los métodos de desarrollo actuales apuestan por una transición gradual del análisis al diseño. En este sentido, una de las principales ventajas del análisis y diseño orientado a objetos con respecto al paradigma estructurado, es que se manejan los mismos artefactos de modelado en ambas fases, lo que facilitaba enormemente la transición[4]. Sin embargo este enfoque presenta inconvenientes:

- ¿Cuándo debe dejarse de analizar? Dado que el modelo de análisis no contiene ningún aspecto de la implementación no puede ejecutarse y por lo tanto, tampoco puede probarse formalmente su corrección. El criterio de completud y corrección de los modelos de análisis dependerá por lo tanto de la opinión de los analistas. La única manera de verificar que el modelo de análisis está bien construido, es realizar todo el proceso de transformación y realizar baterías de pruebas sobre el programa final una vez desarrollado.

[4]Esta reflexión fue realizada por Raúl Izquierdo en un curso de doctorado al que asistí.

- Cuando se introducen en el diseño los aspectos tecnológicos de la solución, se obtiene una mezcla de los aspectos relacionados con el problema a resolver con los aspectos relacionados con una solución concreta al mismo [Raistrick et al., 2004]. Estos aspectos, que corresponden a perspectivas totalmente distintas del proceso de desarrollo, se entremezclan manualmente y rápidamente se vuelven inseparables. Esto dificulta tanto abordar en el diseño los cambios que se produzcan en el análisis, así como cambiar aspectos tecnológicos en el diseño.

Calidad. Cada componente del diseño está sujeto a su propio proceso de transformación manual para convertirse en parte de la aplicación. Por ello, la calidad del código fuente resultante es directamente proporcional a la calidad del programador que realiza la transformación [Raistrick et al., 2004][5].

La herramienta más básica de la que disponen los desarrolladores para verificar la calidad del software, entendida ésta como el grado en éste cumple con los requisitos [ISO, 2000], es la realización de pruebas (*testing*) [Myers et al., 2004]. En los procesos de desarrollo tradicional, el único artefacto sobre el que pueden realizarse pruebas para verificar su corrección es el código fuente. Este código refleja los resultados de todas las fases de desarrollo anteriores: especificación de requisitos, modelo de análisis y modelo de diseño. Esto tiene importantes consecuencias:

- Para verificar cualquiera de los modelos construidos durante el desarrollo de un programa deben acometerse todas las etapas hasta su codificación.
- Cuando se detecta un fallo en las pruebas éste puede deberse a errores de codificación, tecnológicos, de diseño o de análisis. El procedimiento de detección y corrección de errores se entorpece.

Una enfoque interesante para el *testing* es el planteado por XP [Beck, 1999]. Una práctica fundamental que sustenta esta metodología es la construcción de test unitarios para cada unidad funcional (módulo). Los módulos se integran rápidamente en la aplicación entregable y su codificación termina cuando todos los test creados se ejecutan correctamente. Cada vez que se acomete un cambio en el software en el validan los test, modificándose si son requeridos. Al conjunto de técnicas centradas en la práctica de escribir primero un caso de prueba (*test case*) e implementar posteriormente únicamente el código necesario para pasar la prueba se denominó posteriormente *Test-Driven Development* (TDD) [Beck, 2002]. Su principal objetivo es cumplir los requisitos manteniendo el código en su estado más simple posible.

Se trata, como es habitual con los métodos ágiles, de un acuerdo pragmático. Si lo único que puede probarse es el código fuente, ha de darse a su veri-

[5]Por ello los métodos ágiles hacen hincapié en la importancia del talento de las personas que realizan el desarrollo, entendiendo la codificación como un aspecto clave de éste [Fowler, 2005c].

ficación la importancia que se merece. Ésta debe ser una parte fundamental del proceso de desarrollo. Se busca de este modo incrementar la eficiencia de desarrollo, pero no representa una solución completa a los problemas planteados anteriormente.

Trazabilidad de requisitos. Se entiende por trazabilidad de los requisitos como la posibilidad de describir y realizar un seguimiento del ciclo de vida de un requisito durante el proceso de desarrollo [Gotel and Finkelstein, 1994]. Por ejemplo, desde que se descubre en el análisis de requisitos, a través de las fases de desarrollo hasta su codificación y despliegue.

La trazabilidad de requisitos es una herramienta fundamental de otras disciplinas dentro de la ingeniería del software, como por ejemplo la *gestión de la calidad* o el seguimiento (*tracking*) en la *gestión de proyectos*. Su realización se dificulta enormemente por la desconexión entre los artefactos de modelado y el código. Si ya es difícil descubrir el diseño que origina un fragmento de código fuente [Fowler, 2005c], más lo es descubrir el modelo de análisis y mucho más la especificación de requisitos que lo originó.

Las metodologías con un modelo de ciclo de vida iterativo añaden una complejidad adicional a este proceso, puesto que deben considerarse todos los períodos de refinamiento e iteración en todas estas fases [Stepanian, 2004].

Reutilización. En la búsqueda de evitar la duplicación de esfuerzos se han investigado multitud de técnicas y paradigmas que favorezcan la reutilización de artefactos durante las diferentes etapas de desarrollo. Así, pueden citarse la programación orientada a objetos [Booch, 1993]; la tecnología de componentes [Heineman and Councill, 2001]; los patrones arquitectónicos [Buschmann et al., 1996], de diseño [Gamma et al., 1995] y de análisis [Fowler, 1996] y la programación orientada a aspectos [Kiczales et al., 1997]

Aunque se han desarrollado avances significativos en este campo, aun se está lejos de conseguir una reutilización plena de los artefactos construidos. En [Raistrick et al., 2004] se señala que la reutilización únicamente es efectiva en la fase de análisis, puesto que es la única que no considera detalles de implementación. En las fases de diseño, y especialmente en la codificación, los detalles tecnológicos de la plataforma de ejecución impregnan los artefactos construidos, dificultando su reutilización.

Las características de mantenimiento, reutilización, calidad, productividad, trazabilidad, seguimiento son una cuestión recurrente en las taxonomías de objetivos a conseguir para un verdadero proceso de ingeniería del software [Tyrrell, 2001, McCall et al., 1977]. Sin embargo, aunque se han realizado grandes avances para su consecución, siguen existiendo dificultades que impiden su plena consecución y las soluciones planteadas son parciales.

2.1.4. Proceso de Desarrollo utilizando MDA

El enfoque de MDA supone importantes cambios en cómo se acometen las fases en el proceso de desarrollo. Tal y como se explicó en § 2.1.1.6, en MDA se habla de una doble transformación automática: entre un PIM y un PSM y entre un PSM y el código fuente de la aplicación. Este enfoque implica, en primer lugar, cambios sustanciales en las fases de análisis y diseño.

El desarrollo del PIM se corresponderá casi en su totalidad con la fase de análisis en el desarrollo tradicional. Se desarrollará un modelo formal que especifique qué debe realizar la aplicación. Este modelo estará libre de detalles de implementación, pero será extremadamente detallado. Por ejemplo, en el caso de UP, implicarían los sucesivos refinamientos que llevan a satisfacer cada caso de uso: descubrimiento de clases, la asignación de responsabilidades, la creación de objetos, el intercambio de mensajes, etc. Estos aspectos forman parte del diseño, o de la transición del análisis al diseño, en los procesos de desarrollo tradicional, pero en MDA no se contemplan ninguno de los aspectos relativos a la plataforma de ejecución.

El conocimiento experto acerca de la plataforma tecnológica donde se ejecutará la aplicación seguirá siendo necesario y vendrá dado por la definición de la transformación (§ 2.1.1.6). Pero ahora esta información estará separada de la especificación de la aplicación. Se favorecerá la reutilización de los mismos modelos con diferentes definiciones de transformación, y la utilización de las mismas definiciones de transformación con diferentes modelos.

La doble transformación automática PIM– PSM y PSM– código fuente hace que, desde el punto de vista del desarrollador, los modelos se perciban como ejecutables. Esto permitirá validar los modelos realizados. Además permitirá determinar cuándo se ha terminado de modelar: cuando el PIM verifique todos los casos de pruebas diseñados[6] [Raistrick et al., 2004].

Este proceso se representa en la Figura 2.5. MDA haría efectiva la separación de la especificación del modelo de análisis del software y de su diseño, entendido como la realización del análisis en una plataforma concreta. Esto permitiría que ambas actividades fuesen realizadas por grupos de especialistas diferenciados. Los analistas expertos en modelado de software no tienen porqué ser expertos en plataformas tecnológicas concretas y viceversa. En [Raistrick et al., 2004] incluso se propone considerar diseño de la aplicación como uno asunto más a tratar en el análisis.

Tal y como se muestra en la Figura 2.5, la herramienta MDA aceptaría como entradas:

- La especificación de lo que debe realizar la aplicación, dada por el modelo de análisis (PIM).

- La especificación de cómo implementar el modelo de análisis sobre una plataforma concreta, dada por la definición de la transformación (§ 2.1.1.6).

[6]Nótese la analogía con la regla de XP: el desarrollo de un módulo finaliza cuando su código fuente pasa todos los test [Beck, 1999].

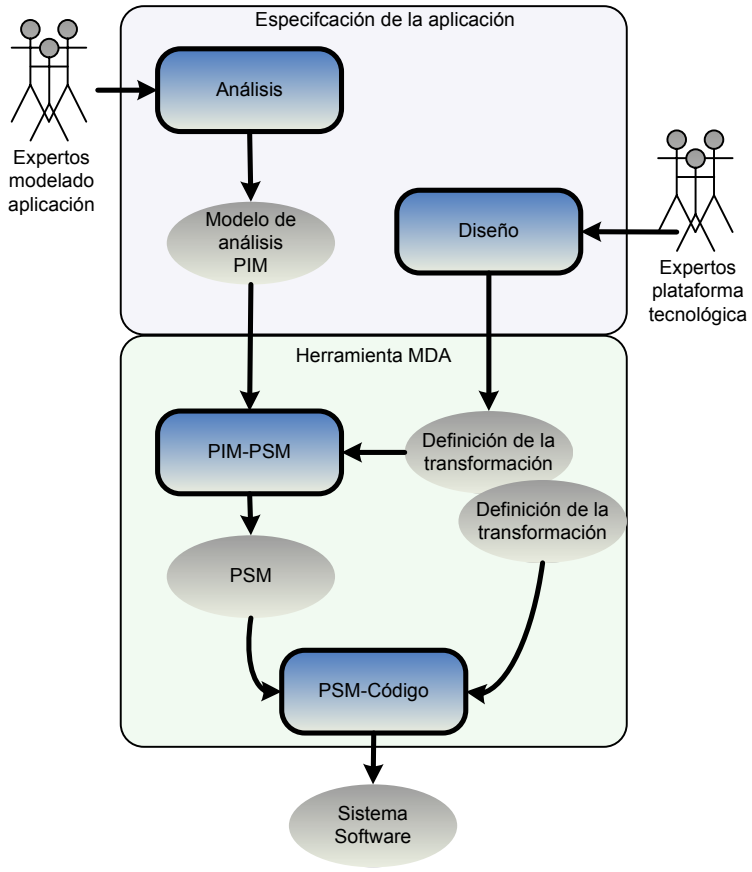

Figura 2.5: Especificación e implementación de aplicaciones con MDA.

A partir de estas entradas, se generaría el PSM que podría ser entonces transformado en el código fuente de la aplicación. Ambas transformaciones serían automáticas.

Con respecto al ciclo de vida utilizado, el proceso de desarrollo utilizando MDA no sería muy diferente del proceso de desarrollo tradicional (Figura 2.6). Los requisitos del software cambian continuamente y la iteración es la manera más eficaz de afrontar el cambio, al igual que sucede en el proceso de desarrollo tradicional revisado en§ 2.1.3.

No obstante, MDA permite hacer que la iteración sea más efectiva. Durante el desarrollo, pueden producirse cambios en los requisitos tecnológicos de la aplicación. Esto implicaría un ajuste en las definiciones de las transformaciones a realizar, reutilizables para la realización de diferentes aplicaciones. Más habituales serán los cambios en los requisitos de negocio. MDA permite absorber estos cambios realizando las modificaciones pertinentes en el modelo de análisis. Al contrario de lo que ocurría con el proceso de desarrollo tradicional, la propagación de estos cambios durante la realización del modelo de análisis son transparentes al desarrollador.

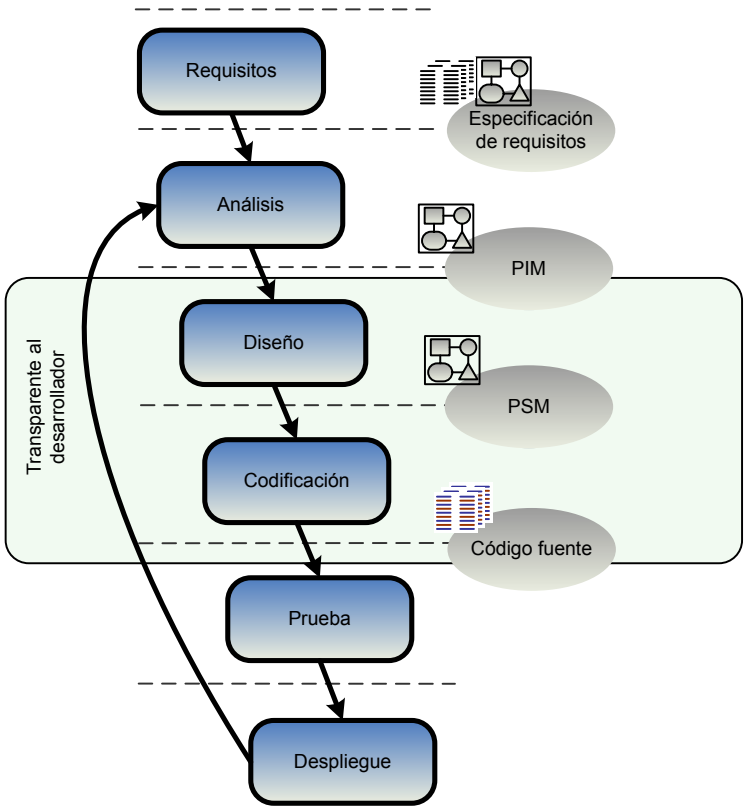

Figura 2.6: Proceso de desarrollo iterativo con MDA.

2.2. Otras Estrategias de Desarrollo Dirigido por Modelos

Al margen de la iniciativa MDA del OMG y el conjunto de estándares que ésta propone, se han desarrollado otras iniciativas que implementan soluciones MDD. Estas iniciativas comparten los fundamentos de MDA, en cuanto a que los modelos son considerados artefactos de primer nivel. También comparten la importancia de los procesos de transformación, que hacen posible la traducción de un conjunto de modelos de entrada en una aplicación ejecutable.

La principal diferencia de estas iniciativas con respecto a MDA radica en su pragmatismo. Buscan una solución concreta actual para lo que en el OMG es ahora un ambicioso objetivo y un conjunto de especificaciones, muchas de las cuales, están actualmente en proceso de desarrollo y sin soporte real por parte de las herramientas existentes. En los capítulos 4 y 5 se analizarán varias propuestas alternativas a los estándares de MDA, o que los utilizan parcialmente.

Capítulo 3

LENGUAJES DE ESPECIFICACIÓN DE MODELOS

3.1. UML como Lenguaje de Definición de Modelos Independientes de la Plataforma

3.1.1. Introducción

El Lenguaje Unificado de Modelado, *Unified Modeling Language* (UML), se define como un lenguaje visual de modelado de propósito general, que se utiliza para especificar, visualizar, construir y documentar los artefactos de un sistema software [Rumbaugh et al., 1998].

La primera versión de lo que finalmente se denominaría UML fue la versión 0.8 del Método Unificado (*Unified Method*)[1] , presentada en Octubre de 1995 en el Congreso *Object-Oriented Programming, Systems, Languages & Applications* (OOPSLA). Incorporaba características de las notaciones propuestas en los métodos orientados a objetos de James Rumbaugh (*Object Modeling Technique* (OMT)) [Rumbaugh et al., 1991] y Grady Booch [Booch, 1993], que habían coincidido en Rational en 1994.

En 1996, Ivar Jacobson, creador de *Object-oriented Software Engineering* (OOSE) [Jacobson et al., 1994], se incorporó a Rational y comenzó a trabajar con Rumbaugh y Booch. Como resultado de este trabajo conjunto se publicó la versión 0.91 de lo que se denominó *Unified Modeling Language* [Booch et al., 1996].

Rational y un grupo de empresas asociadas enviaron la versión 1.0 de UML al grupo de interés en Análisis y Diseño del OMG (*Analysis and Design Task Force*) en

[1] No debe confundirse con el Proceso Unificado que posteriormente desarrollaron los "Three Amigos": Jacobson, Booch y Rumbaugh [Jacobson et al., 1999].

enero de 1997. En septiembre de ese mismo año, enviaron de nuevo una propuesta revisada de UML, versión 1.1, que fue adoptada a finales de 1997 por el OMG como la versión 1.0 de la notación.

A partir de 1997 se sucedieron diferentes versiones de UML (Figura 3.1). En 1998, la versión 1.2. En 1999, la 1.3. La 1.4 en 2001 y la 1.5 en 2002. Desde entonces, y sin que en el momento de escribir esta Memoria[2] se hayan finalizado todas sus partes, el OMG está trabajando en la versión 2.0 del estándar. Algunas de estas revisiones han supuesto cambios significativos tanto en la notación como en la semántica del lenguaje (especialmente en la versión 1.5 y la 2.0). La versión 2.0 de UML incorpora importantes modificaciones internas para adecuarla a la estrategia MDA del OMG. Por lo tanto, cuando en esta Memoria se haga referencia a UML, salvo que se indique lo contrario, se tratará de la versión 2.0 del estándar.

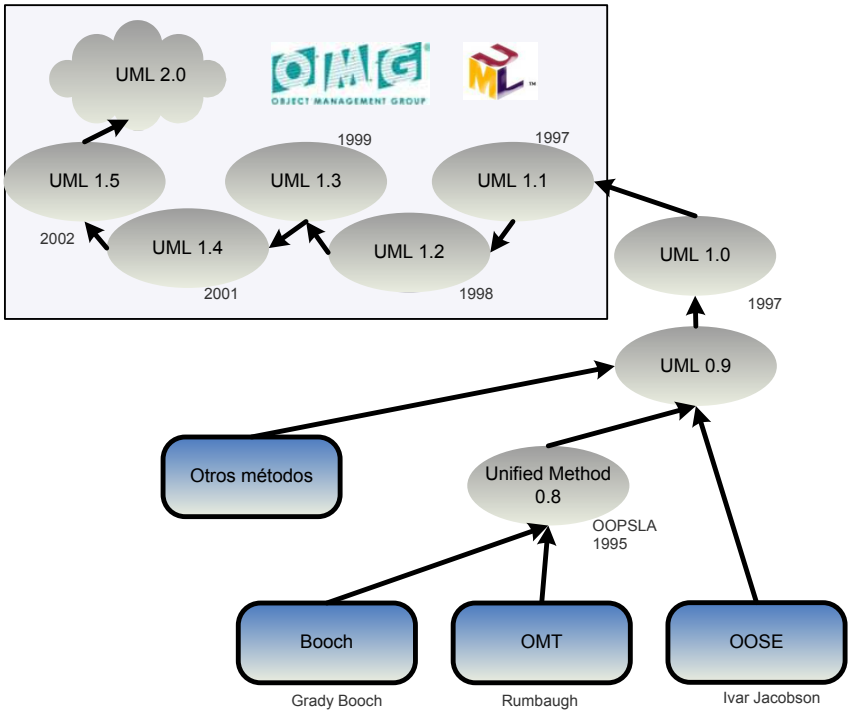

Figura 3.1: Evolución de UML.

3.1.2. Diferentes Maneras de Utilizar UML

UML es un lenguaje amplio y en la comunidad de ingeniería del software existen diferentes aproximaciones acerca de como utilizarlo. La razón radica en que existen diferentes puntos de vista acerca de cómo hacer del desarrollo de software un proceso de ingeniería eficiente.

[2]Julio de 2006

Conviene empezar revisando la clasificación de usos del UML que hacen sus creadores en [Booch et al., 1999]. Establecen que UML es un lenguaje para visualizar, especificar, construir y documentar software:

Visualizar. UML fue concebido como una notación gráfica situada por encima de un metamodelo que especifica su semántica. El utilizar una notación gráfica permite a los desarrolladores expresar utilizando un lenguaje uniforme las abstracciones que construyen de los sistemas en forma de modelos. Por otra parte, la especificación de modelos de forma gráfica permite entender aspectos de los sistemas software que serían imposibles de abordar analizando su código fuente. Además, el poder utilizar una notación gráfica común, facilita la comunicación de conceptos entre diferentes desarrolladores.

Especificar. Por especificar se entiende construir modelos que sean precisos, no ambiguos y completos. Bajo esta definición, el objetivo de UML es permitir especificar todas las decisiones de análisis, diseño e implementación que deban tomarse durante el desarrollo y despliegue de un sistema software.

Construir. UML no es un lenguaje de programación visual, pero sí pueden conectarse directamente sus modelos a diferentes artefactos de implementación, como código fuente o tablas de bases de datos.

Así en [Booch et al., 1999] se habla de la posibilidad de utilizar con UML *ingeniería directa* como la generación de código fuente a partir de un modelo UML. También de *ingeniería inversa*, como las actividades encaminadas a extraer modelos a partir del código fuente, siempre que el código contenga la información suficiente para reconstruir los modelos.

Documentar. Los modelos construidos con UML permiten documentar con detalle la arquitectura de los sistemas así como los artefactos manejados durante su desarrollo. Esta documentación será una valiosa herramienta durante el desarrollo de los sistemas, y durante el mantenimiento posterior a su despliegue.

Esta clasificación de usos potenciales de UML debe entenderse en el momento de la presentación del lenguaje a sus posibles usuarios. Una clasificación distinta, más concreta y basada en usos observados en la práctica, es la que realizan Martin Fowler y Stephen Mellor [Fowler, 2003e]. Bajo esta clasificación, existen tres modalidades de uso de UML: boceto (*sketch*), esquema/plano (*blueprint*) y lenguaje de programación.

UML como Boceto (*Sketch*). En esta modalidad de uso, UML se utiliza para *comunicar* los diferentes aspectos de un sistema, en lugar de para especificarlos formalmente [Fowler, 2003c].

Bajo este modo, sigue siendo posible realizar actividades de ingeniería directa e inversa con los modelos y el código, sin embargo el aspecto fundamental

es la *selección* de los aspectos a modelar. El objetivo último es utilizar los bocetos para facilitar la comunicación de ideas y alternativas sobre los aspectos que se están desarrollando.

Esta modalidad de uso de UML es la más extendida en la actualidad. Según Steve Cook es el único modo de uso de UML que satisface completamente las aspiraciones de los creadores de UML: eliminar las diferencias entre los diferentes modos de representar diseños orientados a objetos [Cook, 2004]. Existe un consenso en la actualidad acerca de la idoneidad de UML para crear documentación informal que represente diseños orientados a objetos.

Este enfoque de modelado ha sido adoptado por los artífices del movimiento ágil, tal y como se explicó en § 2.1.3.3. Los bocetos son informales y tienen un carácter dinámico. Pueden realizarse y modificarse rápidamente y de una forma colaborativa, por ejemplo, utilizando una pizarra. Los bocetos también sirven para documentar, pero centrándose en la comunicación en lugar de en la completud de la especificación. Encaja por lo tanto con el planteamiento del modelado ágil [Ambler and Jeffries, 2002].

UML como Plano (*Blueprint*). En esta modalidad los modelos UML se centran en la completud. Los diagramas serán lo suficientemente completo como para que puedan ser transformados sistemáticamente, de una manera automática o manual, en código fuente. Este uso está inspirado en otras disciplinas de ingenería [Fowler, 2003d]: un diseñador elabora un diseño lo suficientemente detallado como para que un programador lo codifique. Aunque los roles de diseñador y programador pueden pertenecer a la misma persona, habitualmente será un diseñador con experiencia el que diseñe para un equipo de programadores [Fowler, 2003a].

Los planos detallados pueden ser utilizados para todos los aspectos del sistema, o únicamente para determinados aspectos concretos. Es habitual que un diseñador elabore un plano que únicamente especifique los interfaces de los subsistemas y que los desarrolladores se encarguen de los detalles de implementación internos a cada módulo.

Los planos UML detallados facilitan las actividades de ingeniería directa e inversa, aunque también requieren de herramientas más sofisticadas, típicamente herramientas *Computer-Aided Software Engineering* (CASE) especializadas.

UML como Lenguaje de programación. Bajo este enfoque, los modelos UML contienen toda la información necesaria para poder ser transformados en código ejecutable [Fowler, 2003b]. UML es a la herramienta que lo transforma lo que el código fuente a un compilador tradicional.

Esta modalidad de uso de UML es la más reducida de las tres, con un reducido soporte a nivel comercial. También es la que requiere herramientas más sofisticadas.

Desde el punto de vista de MDA y de esta investigación, las modalidades de uso más relevantes son las dos últimas. En el contexto de MDA, cuando la transformación entre los PIM y los PSM es automática, se tiene UML como Lenguaje de Programación. Cuando requieren alguna intervención manual, se tiene UML como Plano [Fowler, 2003d].

Un cuarto uso de UML, relacionado con UML como Lenguaje de programación, es el denominado UML Ejecutable (*Executable UML*). Tiene su origen en los trabajos realizados por Sally Shlaer y Stephen J. Mellor [Shlaer and Mellor, 1988, Shlaer and Mellor, 1991], y define una extensión de UML que permite que los modelos sean ejecutables. Por su importancia de cara a esta investigación, este enfoque será analizado en profundidad en § 3.2.

3.1.3. La Especificación de UML

El lenguaje UML viene definido por un conjunto de documentos publicados por el OMG [OMG, 2006b]. La versión actual de UML es la 2.0 y está compuesta por cuatro partes, aunque no todas ellas han sido completadas en el momento de escribir esta Memoria.

UML 2.0 *Superstructure*. Especifica el lenguaje desde el punto de vista de sus usuarios finales. Define 6 diagramas estructurales, 3 diagramas de comportamiento, 4 diagramas de interacción así como los elementos que todos ellos comprenden. Es la única de las cuatro partes que comprenden la especificación de UML 2.0 que ha sido completada [OMG, 2004b].

UML 2.0 *Infrastructure*. Especifica las construcciones que conforman los cimientos de UML. Para ello, define el núcleo de un metalenguaje que podrá ser reutilizado para definir los metamodelos de otros lenguajes: alinea el metamodelo de UML con *Meta Object Facility* (MOF) (ver § 3.3).

UML 2.0 *Object Constraint Language* (OCL). OCL es un lenguaje que permite ampliar la semántica de los modelos UML mediante la definición de precondiciones, postcondiciones, invariantes y otras condiciones [OMG, 2005e].

UML 2.0 *Diagram Interchange*. Extiende el metamodelo de UML con un paquete adicional que modela información de carácter gráfico asociada a los diagramas, permitiendo el intercambio de modelos conservando su representación original [OMG, 2005j].

Aunque en versiones iniciales de UML se centraban en la notación y en explicar el significado de cada elemento gráfico, las versiones más recientes incluyen un modelo formal de la propia semántica de UML. Esta definición formal se realiza especificando el metamodelo de UML[3] [Rumbaugh et al., 1998]. Se deno-

[3]Los conceptos de metamodelado, su importancia para MDA y el estándar *Meta Object Facility* (MOF) serán revisados con mayor profundidad en § 3.3.

mina metamodelo porque define los elementos de un lenguaje de modelado, esto es, las construcciones que UML utiliza. El metamodelo de UML viene a su vez expresado como UML. Se trata de un ejemplo de un intérprete metacircular [Wand and Friedman, 1988], es decir, es un lenguaje definido en términos de s mismo.

Aunque existen métodos de especificación más rigurosos, el enfoque basado en metamodelado utilizado para definir UML ofrece la ventaja de ser más intuitivo y práctico para los usuarios de UML y, en especial, para los desarrolladores de herramientas [OMG, 2005k].

En la Figura 3.2 se observa un ejemplo del metamodelo de UML. En él puede verse como diversos artefactos de modelado de UML son representados en términos de clases, generalizaciones y asociaciones. Una clase (`Class`) contiene Operaciones (`Operation`). Una Operación es una Característica de comportamiento (`BehavioralFeature`) la cual agrega parámetros (`Parameter`) y tiene un Tipo (`Type`), que será el tipo de retorno de la operación. Los Tipos primitivos (`PrimitiveType`) y las Clases son Clasificadores (`Classifier`). Los clasificadores a su vez son Tipos. De este modo, una Clase o un Tipo primitivo podrá ser el tipo de retorno de una operación.

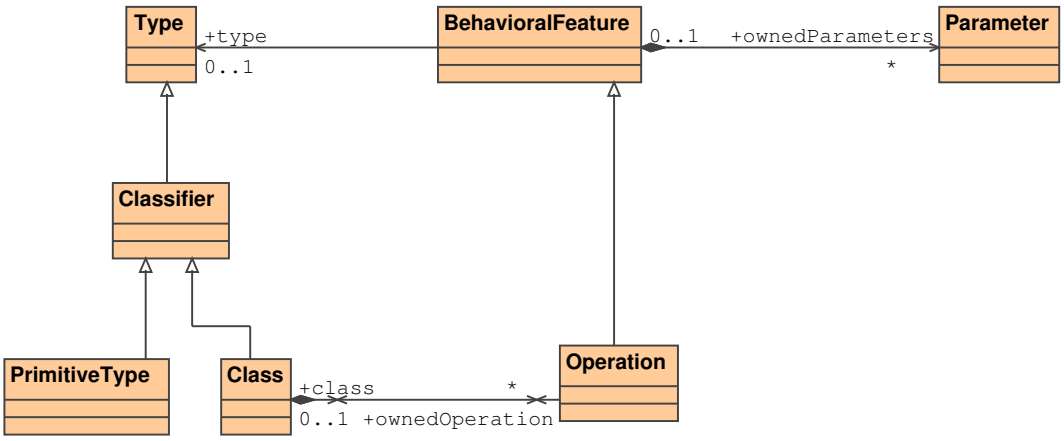

Figura 3.2: Ejemplo de fragmento del metamodelo de UML.

Un concepto clave para entender la arquitectura de UML es que, aunque la mayor parte de sus usuarios únicamente conocen la notación gráfica, ésta es sólo una de las posibles representaciones de su metamodelo. UML define la sintaxis de una lenguaje que sirve para modelar, pero se trata de lo que su especificación denomina "sintaxis abstracta" (*abstract syntax*) [OMG, 2004b]. La sintaxis abstracta de UML vendrá dada por su metamodelo: el metamodelo de UML asocia los conceptos que manejan los usuarios de UML con una sintaxis abstracta, en lugar de con la sintaxis de una notación gráfica.

Sobre la existencia de una única sintaxis abstracta, pueden existir diferentes sintaxis concretas para expresar modelos UML. Un ejemplo sería la notación gráfica de

UML. Otros ejemplos de sintaxis concretas serían *Human-Usable Textual Notation* (HUTN) [OMG, 2004c] (§ 3.3.4.2) o una sintaxis XML basada en *XML Metadata Interchange* (XMI) [OMG, 2005b] (§ 3.3.4.1).

En la Figura 3.3 se muestra un ejemplo sencillo. Dada una clase `Persona` con un atributo `edad` de tipo entero, su representación por medio de la sintaxis abstracta de UML comprende la utilización de dos elementos del metamodelo de UML: un objeto `Class` que representa la metaclase de `Persona` y un objeto `Property` que representa el atributo. Esta representación en forma de sintaxis abstracta admite diferentes representaciones. En la parte superior de la Figura 3.3 puede observarse una representación utilizando la notación gráfica de UML y una representación como XML basada en XMI.

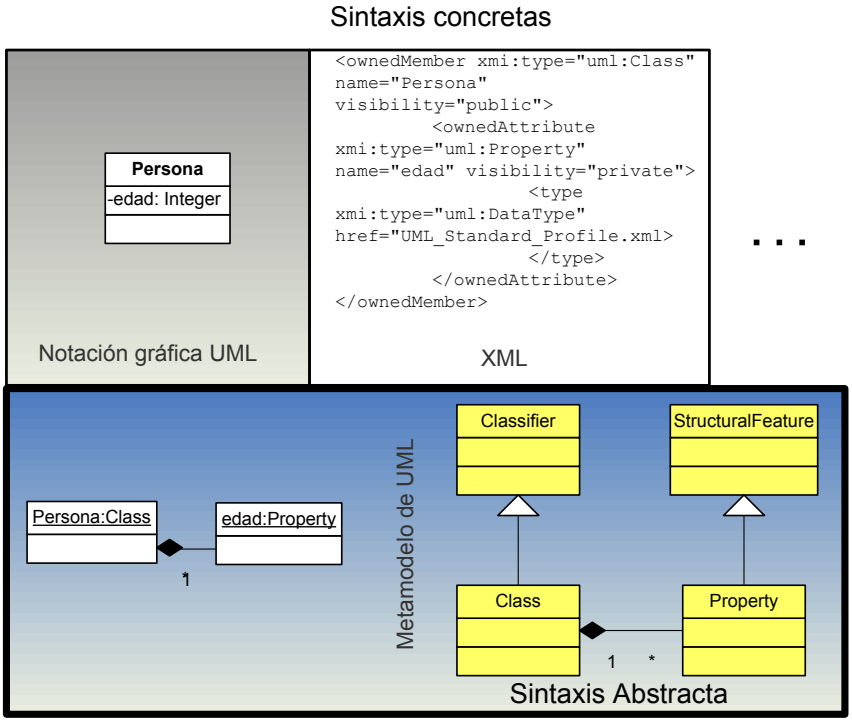

Figura 3.3: Sintaxis abstracta de UML.

La actual especificación de UML 2.0[OMG, 2004b] está dividida en tres grandes partes:

Parte I. *Structure*. Define las construcciones estáticas y estructurales utilizadas en los diagramas de estructura, como los diagramas de clases, de componentes y de despliegue.

Parte II. *Behavior*. Especifica las construcciones dinámicas y de comportamiento utilizadas en los diagramas de comportamiento, como los diagramas de actividad, de secuencia y de estados.

Parte III. ***Supplement.*** Define construcciones auxiliares y los perfiles (*profiles*) que permiten especializar UML en diversos dominios.

Dentro de cada parte, los conceptos se definen a través de su sintaxis abstracta. Se incluyen los diagramas UML de clases y de paquetes que representan los conceptos y relaciones. A continuación se definen formalmente cada uno de los conceptos. Estas definiciones están estructuradas en diversos apartados y son eminentemente textuales, aunque en ocasiones se apoyan en restricciones *Object Constraint Language* (OCL) para definir ciertas propiedades (§ 3.1.5). Uno de los apartados que contienen es la notación propuesta, gráfica o textual, a utilizar para representar gráficamente los conceptos.

3.1.4. Modelado con UML

Las construcciones de UML se dividen en dos grandes grupos según sean utilizadas para modelar aspectos estructurales (estáticos) o de comportamiento (dinámicos) de un sistema. Los modelos de comportamiento especifican como cambian en el tiempo los aspectos estructurales de un sistema [Bock, 2003b].

Con este mismo criterio se dividen los diagramas de UML en dos grandes grupos: estructurales y de comportamiento. En la Figura 3.4 se representan los 13 diagramas de UML 2.0. Los diagramas nuevos en UML 2.0 son el de composición de estructura (*composite structure*), el de *timing* y el de perspectiva de la interacción (*interaction overview*). También se ha renombrado los diagramas de colaboración de UML 1.x como "diagramas de comunicación".

A continuación se realizará un estudio de los diferentes diagramas que componen la notación gráfica de UML bajo la perspectiva de su aplicación en una herramienta MDA. Se analizará cada diagrama entendiendo un uso de UML "como lenguaje de programación" (§ 3.1.2). Bajo este enfoque no todos los diagramas resultan de interés. Existen *diagramas estructurales* cuyos artefactos presentan una semántica de naturaleza distinta a las construcciones lógicas utilizadas utilizadas para desarrollar software. Del mismo modo, existen *diagramas de comportamiento* que no representan ningún comportamiento en tiempo de ejecución. Esto no significa que estos diagramas no sean de utilidad en un proceso que utilice MDA, pero su uso será idéntico al que se les daría en el proceso de desarrollo tradicional.

Se prestará una especial atención a la especificación formal de UML mediante su metamodelo.

3.1.4.1. Modelado Estructural

3.1.4.1.1. Diagrama de Clases

Un diagrama de clases describe el tipo de los objetos de un sistema software, así como las diversas relaciones estáticas que existen entre ellos [Fowler, 2003d]. Los diagramas de clases también muestran las propiedades y operaciones de cada clase, así como las restricciones acerca del modo en el que se conectan los objetos.

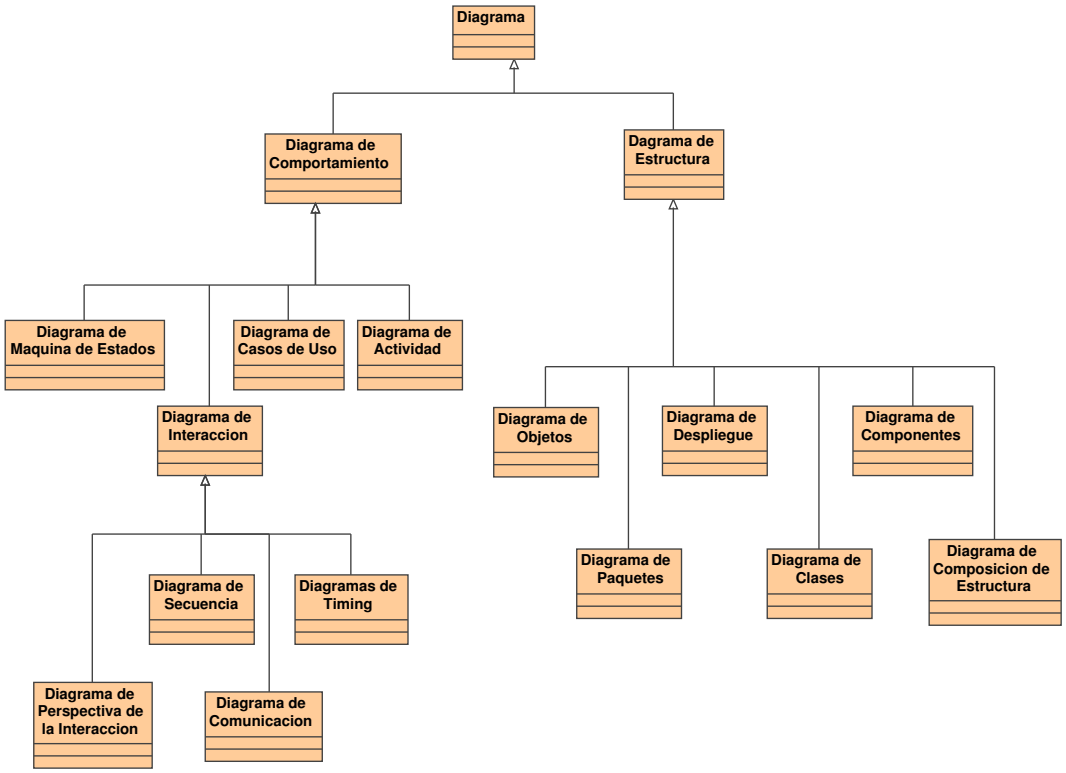

Figura 3.4: Diagramas UML 2.0

Los diagramas de clases son el tipo de diagrama UML más conocido. La razón es que permite representar los conceptos tradicionales básicos de la orientación a objetos: clases, generalización, interfaces, atributos, métodos, asociaciones, etc. Además, es el tipo de diagrama que comprende un mayor número de conceptos de modelado.

Los diagramas de clases se utilizan desde las etapas tempranas de análisis, para representar el modelo de dominio de la aplicación a través de las clases de dominio que se descubren en el análisis. También se utilizan durante el diseño, para detallar la estructura y relaciones de los objetos que conformarán el diseño que se implementará de la aplicación [Larman, 2004].

Los lenguajes de programación orientados a objetos actuales ofrecen construcciones para codificar directamente los conceptos más habituales utilizados en los diagramas de clases. Por ello, los diagramas de clase son utilizados en multitud de herramientas como entrada para mecanismos de generación de código. El carácter estático de las estructuras de modelado permite en muchas ocasiones un mapeo directo con un equivalente en código fuente.

Por ejemplo, conceptos como el nombre de las clases, la lista de atributos y operaciones o las relaciones de generalización tienen su correspondencia directa en el código. En la Figura 3.5 se muestra como puede codificarse un sencillo diagrama UML utilizando Java. Los elementos son una clase de nombre `Persona`, con el

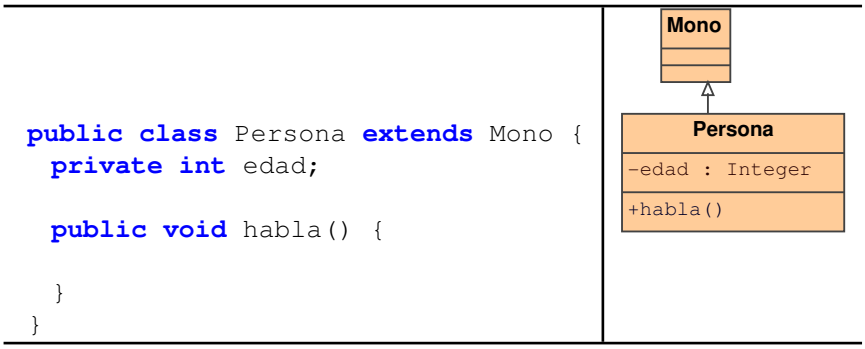

Figura 3.5: Ejemplo de asociación directa de código con el modelo de una clase UML

atributo `edad` y el método `habla()`, que especializa la clase `Mono`. Como se observa en el código, todos estos elementos tienen su representación directa en el código.

Sin embargo existen ciertas ciertas impedancias a la hora de codificar modelos orientados a objetos con los lenguajes de programación existentes. Este problema es abordado por Raúl Izquierdo en [Izquierdo Castanedo, 2002]: a la hora de traducir los elementos[4] que se descubren en el dominio del problema a código fuente, quedan todos incrustados en el concepto de clase del lenguaje de programación utilizado. Esto dificulta afrontar los cambios que se producen en el dominio así como reutilizar los elementos descubiertos en diferentes dominios. Como consecuencia la productividad del desarrollo y la mantenibilidad del software se ven seriamente mermadas.

Este problema repercute en las actividades de ingeniería directa e inversa utilizando o generando diagramas de clases. En el caso de que no existan artefactos en el código fuente que puedan asociarse directamente a los elementos de modelado, cuando se genera código a partir de un diagrama de clases se pierde necesariamente información. Este es el caso de diversos conceptos utilizados habitualmente al modelar la vista estática de clases, como las asociaciones, las relaciones de dependencia o las cardinalidades.

Este hecho supone un problema para realizar ingeniería inversa y obtener un modelo estático de clases a partir de código fuente. En un proceso de desarrollo que utilice ingeniería directa con un enfoque "UML como plano" (§ 3.1.2), es habitual que el código generado necesite modificaciones o que se necesiten modificaciones en el software y éstas se acometan directamente sobre el código por agilidad. La ingeniería inversa en este contexto es una poderosa herramienta, porque evita al desarrollador el tener que mantener sincronizada una información redundante. Por ello, las herramientas UML que permiten ingeniería inversa utilizan diferentes mecanismos que permiten asociar los elementos de codificación a elementos de modelado.

[4]Los elementos señalados en [Izquierdo Castanedo, 2002] son: entidades, relaciones, operaciones y reglas de negocio.

```
public class Persona {
  /**
   * @clientCardinality 0..1
   * @directed true
   * @link aggregationByValue
   */

  private Coche lnkCoche;
}
```

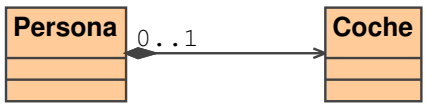

Figura 3.6: Ejemplo de asociaciones UML especificadas como anotaciones en el código. Herramienta: Borland Together Architect 2006

Una solución es introducir metainformación en el código fuente, en forma de comentarios, que incorpore información de modelado a los elementos del código. Esta es el mecanismo utilizado por el entorno de desarrollo Borland Together Architect 2006 [Borland, 2006]. En la Figura 3.6 se muestra un ejemplo. En él se observa una clase `Persona` que tiene una relación de composición, dirigida y con una cardinalidad 0 a 1 en el origen, con una clase `Coche`. A la hora de trasladar a código este modelo se utiliza un convenio de nomenclatura para representar la asociación pura: crear un atributo de nombre `lnk+Clase`. El resto de atributos de la asociación se representan mediante anotaciones previas al atributo utilizando etiquetas JavaDoc [Sun, 2004].

Otro enfoque distinto es el utilizado por MagicDraw [Magic, 2006]. Para poder generar código de las asociaciones, la herramienta obliga a etiquetar el extremo de la relación que hace referencia a la clase asociada. El nombre de dicha etiqueta será la clave que asocie el modelo UML a un atributo del mismo nombre en el código. En el modelo UML, almacenado externamente como XMI, se almacenan el resto de atributos de la relación.

Sea cual sea el enfoque utilizado, tiene que prefijarse un mecanismo que permita asociar la información de modelado al código fuente. En los ejemplos anteriores esta información viene dada como metadatos incrustados en el código fuente o asociada externamente utilizando convenios de nomenclatura. Una consecuencia de este enfoque es que se impide la ingeniería inversa sobre código fuente que no haya sido generado por la propia herramienta, puesto que el mecanismo de asociación es propio de cada fabricante.

Los diagramas de clases son utilizados también para generar código fuente de distinta naturaleza al de un lenguaje de programación tradicional. La razón es que la noción de "clasificación", entendida como la agrupación basada en propiedades comunes [Mellor et al., 2004], y la representación de relaciones entre las entidades agrupadas es una práctica habitual en otros dominios.

Un caso particular es la construcción de modelos de datos. Existen herramientas que permiten generar scripts *Data Definition Language* (DDL) SQL [Chamberlin and Boyce, 1974] a partir de modelos estáticos de clases UML. El script definirá las tablas de la base

de datos, asociando tablas a clases, columnas a atributos y relaciones UML a relaciones entre tablas. Por ejemplo, la herramienta UML MagicDraw [Magic, 2006] permite generar scripts DDL SQL para diferentes dialectos SQL de diferentes gestores de bases de datos relacionales. En la Figura 3.7 se muestra un ejemplo de un modelo de clases a transformar en DDL con esta herramienta. En la Figura 3.8 se muestra el resultado de dicha transformación a tablas, también representado como un modelo de clases UML utilizando un perfil DDL de UML (§ 3.1.6).

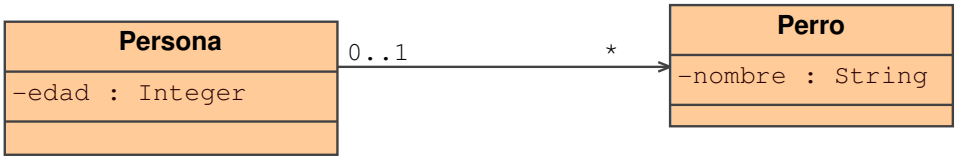

Figura 3.7: Diagrama de clases a transformar en un modelo de tablas relacionales

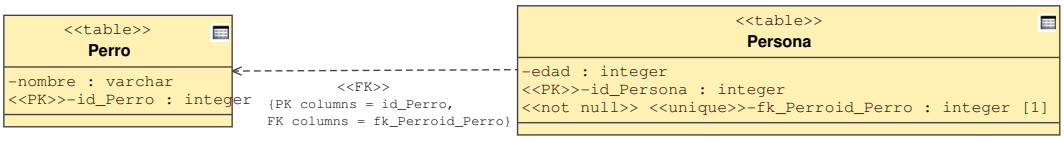

Figura 3.8: Diagrama de clases que representa las tablas de base de datos resultado de la transformación

El código DDL que define las tablas representadas en el diagrama de clases estereotipado se recoge en el Listado 3.1. Utilizando el vocabulario de MDA, el modelo representado en la Figura 3.7 será el PIM. El resultado de la transformación recogido en Figura 3.8 sería un PSM. Este PSM sería finalmente transformado al código recogido en el Listado 3.1.

Listado 3.1: Código DDL SQL generado a partir del diagrama de clases UML

```
CREATE TABLE ddl_model.magicdraw.Perro
(
  nombre varchar,
  id_Perro integer PRIMARY KEY
);

CREATE TABLE ddl_model.magicdraw.Persona
(
  edad integer,
  id_Persona integer PRIMARY KEY,
  fk_Perroid_Perro integer NOT NULL UNIQUE,
  FOREIGN KEY(fk_Perroid_Perro) REFERENCES ddl_model.magicdraw
      .Perro (id_Perro)
);
```

Otro caso de utilización de diagramas de clases son con herramientas de persistencia de datos basadas en el mapeo objeto/relacional. Éstas herramientas necesitan metainformación que especifique cómo se reflejan los elementos del modelo de objetos en la base de datos relacional. UML permite construir modelos de clases que incorporen dicha información, que típicamente hace referencia a atributos, relaciones y cardinalidades. En este sentido, AndroMDA [AndroMDA, 2006] presenta un cartucho que genera automáticamente los datos que configuran cómo realizar la asociación objeto/relacional para Hibernate [Bauer and King, 2004].

Tal y como se ha mostrado en el ejemplo anterior, los modelos de clases UML pueden utilizarse en para construir modelos PIM y PSM. Los PIM y los PSM muestran el nivel adecuado de información para distintos participantes en diferentes etapas del proceso de desarrollo. Pero en MDA es imprescindible que la transformación entre PIM y PSM sea formalizada a través de un conjunto de reglas de transformación [Kleppe et al., 2003].

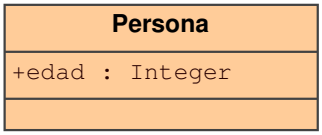

Figura 3.9: Ejemplo de un modelo PIM

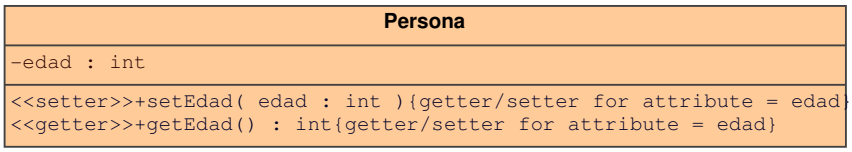

Figura 3.10: Ejemplo de un modelo PSM especializado en la plataforma Java

En la Figura 3.9 se recoge un ejemplo muy sencillo de un modelo PIM. En la Figura 3.10 se recoge dicho modelo PIM transformado a un modelo PSM para la plataforma Java. Las reglas de transformación indicarían que hay que crear un par de métodos de acceso `get()` y `set()` por cada atributo público UML. Que los tipos primitivos UML deben mapearse a los correspondientes tipos primitivos Java. Y que además se especifican con un estereotipo para diferenciarlos de otros métodos que empiecen con `get()` y `set()` y no sean métodos de acceso a propiedades. El PSM resultante sería fácilmente trasladable a un esqueleto de código Java (Listado 3.2).

Listado 3.2: Código Java generado a partir del modelo PSM

```
public class Persona{
  private int edad;

  public void setEdad( int edad ){
```

```
    this.edad=edad;
  }

  public int getEdad( ){
    return edad;
  }
}
```

Existen diversas variantes de los diagramas de clase, dependiendo de los símbolos utilizados y del aspecto cuya estructura se desea reflejar. Los más habituales son los diagramas de estructura, centrados en las clases e interfaces, así como en las relaciones entre ellos. Otras variaciones de los diagramas de clases son los diagramas de paquetes y de objetos [OMG, 2004b].

3.1.4.1.2. Diagrama de Objetos

Los diagramas de objetos representan una fotografía instantánea de los objetos de un sistema en un momento dado [Fowler, 2003d]. A diferencia de los diagramas de estructura, estos diagramas no muestran clases, sino instancias de clases.

Los diagramas de clases no resultan relevantes de cara a MDA, puesto que reflejan la estructura de los objetos en un momento determinado del sistema sin especificar de ninguna manera cómo se llegó a dicha situación. Su función está centrada en la comunicación y clarificación de conceptos, por ejemplo, para ilustrar como se configuran un conjunto de objetos.

3.1.4.1.3. Diagrama de Paquetes

Los diagramas de paquetes son diagramas de estructura UML centrados en representar la estructura de paquetes del modelo UML. Un paquete UML es una construcción que permite agrupar cualquier construcción UML (incluidos otros paquetes) [OMG, 2004b].

Los paquetes UML son utilizados intensivamente en la especificación de UML para agrupar las construcciones UML en unidades de un mayor nivel de abstracción.

Un paquete UML define un espacio de nombres UML en el que los elementos pertenecientes deben ser únicos[5]. Esto permite construir estructuras anidadas de paquetes donde cada elemento tiene un nombre único que lo identifica (*full qualified name*).

Esta idea es utilizada por plataformas como [Gosling et al., 1996] Java y .NET [Thai and Lam, 2002] para la agrupación de clases e interfaces en librerías, por lo que la aplicación de los modelos de paquetes como entrada para una herramienta MDA es directa en estos casos. Si la plataforma destino no soportase el concepto de paquete, esta información podría ignorarse o utilizarse con otros fines, por ejemplo, para agrupar el código fuente en directorios de una forma lógica.

[5]En términos del metamodelo de UML: se define una clase `Package` que especializa `Namespace`.

Un caso particular será la especificación de cómo se configuran las unidades lógicas estableciendo relaciones de dependencia UML entre paquetes. Utilizando una herramienta MDA, los diagramas de paquetes permiten indicar cómo se agrupan los elementos de cada modelo y cómo se relacionan estas agrupaciones entre sí. Esta información podría ser utilizada para establecer las relaciones de configuración en la fase de generación de código. Además, a la hora de construir los PSM, este mecanismo permite especificar qué bibliotecas de la plataforma destino son utilizadas.

3.1.4.1.4. Diagrama de Despliegue

Los diagramas de despliegue muestran un esquema físico del sistema, especificando qué elementos software son ejecutados sobre qué elementos hardware [Fowler, 2003d]. No aportan por lo tanto semántica a la especificación lógica de la estructura del software y no son de aplicación para esta investigación.

3.1.4.1.5. Diagrama de Componentes

Los diagramas de componentes UML permiten representar la estructura de componentes software. Los componentes representan partes de un sistema software que son independientes, intercambiables y que exponen sus servicios proporcionando una serie de interfaces al entorno donde se ejecutan. Se ha establecido un largo debate en torno a qué debe ser considerado un componente, cómo se definen estos formalmente y cuál es la diferencia con respecto a clases normales. Para profundizar en estos aspectos se recomienda la lectura de [Heineman and Councill, 2001].

Desde el punto de vista de UML, los componentes se definen como unidades modulares con interfaces bien definidos que son reemplazables dentro de su entorno [OMG, 2004b][6]. Formalmente serán clases especializadas que definen una especificación externa a través de un conjunto de interfaces proporcionados y requeridos. En su implementación, esta especificación externa (contrato) podrá ser opcionalmente realizada por un conjunto de clasificadores (Figura 3.11).

Los diagramas de componentes tienen aplicación para la generación de código de MDA. Por un lado, permiten especificar qué interfaces expone un componente a su entorno. Estos interfaces podrán ser realizados por el propio componente, o por alguno de los clasificadores que realizan al componente. Esta información puede ser utilizada para la generación de clases: bien para los componentes o para los clasificadores de los cuales el componente es una abstracción. También para la generación de interfaces que representan los servicios que el componente expone en su entorno y de las realizaciones de dichos interfaces por parte de las clases.

Con respecto a la información relativa a qué interfaces requiere un componente, permite generar código relativo a cómo se configuraría el el componente en su en-

[6] En versiones anteriores de UML los componentes se utilizaba para representar estructuras físicas. Para este fin, en la versión actual se utilizan artefactos en los diagramas de despliegue (§ 3.1.4.1.4).

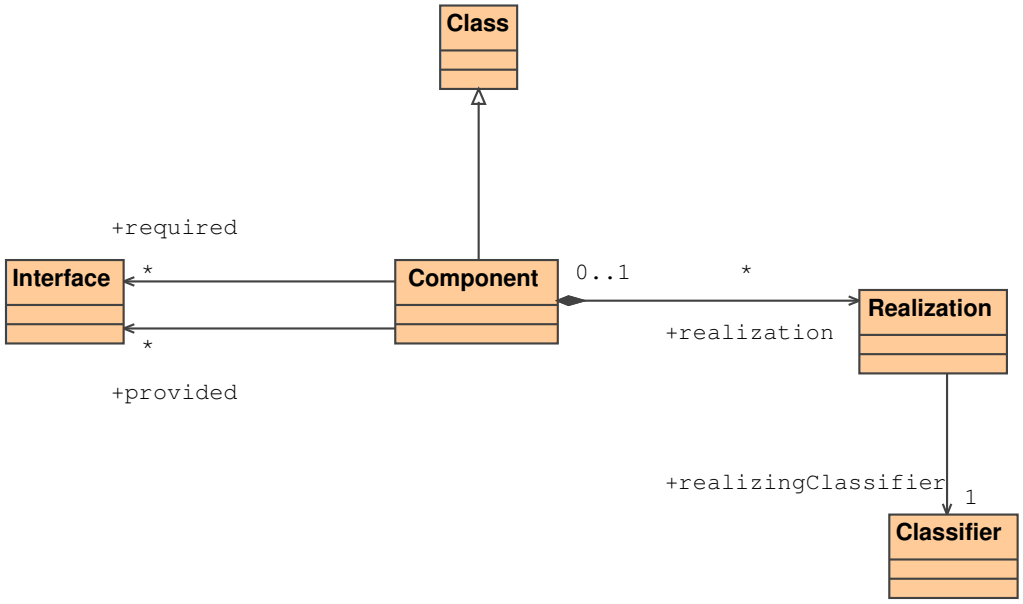

Figura 3.11: Definición de componentes en el metamodelo UML. Un componente especializa una clase que define interfaces requeridos y suministrados. Así mismo, un componente puede ser opcionalmente realizado por un conjunto de clasificadores a través del concepto de `Realization`.

torno. Por ejemplo, podrían generarse propiedades que permitiesen establecer asociaciones con objetos que implementasen dichos interfaces.

En lo relativo a la correspondencia en el código de fuente de un componente con las clases que lo implementan, habitualmente existe una clase que agrupa los elementos del componente. Aunque la especificación de UML establece que un componente puede ser realizado opcionalmente por un conjunto de clases, en la práctica las herramientas suelen adoptar el convenio de generar una clase por componente. En el caso de MagicDraw[Magic, 2006], si se desea generar código fuente para un componente se obliga a que el componente sea realizado, al menos por una clase, y además el nombre de dicha clase debe corresponderse al del componente. Otras herramientas como Rational Software Architect[IBM, 2006] asocian directamente un componente con una clase.

La especificación de componentes UML también permite representar componentes relativos a plataformas tecnológicas concretas que soportan el concepto de componentes, como componentes .NET[Lowy, 2003], EJB [Sun, 2006] o CORBA [OMG, 2004a]. Esta característica puede ser utilizada en MDA a la hora de especificar los modelos PSM. A tal efecto, deberán utilizarse perfiles UML específicos de dichas tecnologías (§ 3.1.6).

3.1.4.1.6. Diagrama de Composición de Estructura

Creados en la versión 2.0 de UML, los diagramas de composición de estructura (*composite structure*) permiten representar a instancias interconectadas colaborando sobre enlaces de comunicación en tiempo de ejecución [Ambler, 2004]. Su motivación radica en que las clases y diferentes tipos de asociaciones UML se muestran insuficientes a la hora de reflejar cómo se asocian los objetos para componer estructuras más complejas [Bock, 2004].

Las instancias interconectadas pueden formar parte de la estructura interna de un clasificador. Por ello, estos diagramas se utilizan habitualmente para descomponer clasificadores UML en su estructura interna, y cómo se agrupan los elementos internos en tiempo de ejecución para conseguir un objetivo común de forma colaborativa [Fowler, 2003d]. También permiten representar con el concepto de "puerto" los puntos de interacción mediante los cuales los clasificadores se relacionan con su entorno. Es por ello que estos diagramas compartan gran parte de notación con los diagramas de componentes (§ 3.1.4.1.5).

Los diagramas de composición de estructura definen también un nuevo tipo de clasificador denominado "colaboración" que recoge los roles y conexiones de diferentes clasificadores en su cooperación para lograr un propósito concreto.

Al igual que los diagramas de objetos (§ 3.1.4.1.2) los diagramas de composición de estructura representa el estado de una estructura de objetos que se produce dentro del sistema en un momento concreto. No obstante, presentan sustanciales diferencias con respecto a aquellos. Por un lado, permiten realizar dicha representación con un nivel de detalle cualitativamente superior. Permiten exponer la estructura interna de los clasificadores a través de las instancias que contienen, y permiten representar como estas instancias se conectan para conformar los servicios que el clasificador proporciona a su entorno. Además, en lugar de la estructura de las asociaciones representa únicamente la capacidad de comunicación entre las instancias conectadas.

En el contexto de una herramienta MDA, este mayor nivel de detalle permite ciertas actividades de generación de código. Al exponer la estructura interna de instancias contenidas por un clasificador, pueden generarse las propiedades correspondientes a dichas instancias. Del mismo modo pueden generarse propiedades para los puertos que representan puntos de interacción del clasificador. Para la generación de este tipo de código, los diagramas de composición de estructura no representan ninguna ventaja con respecto a los diagramas de estructura de clases (§ 3.1.4.1.1).

Con respecto a otros aspectos de estos diagramas, como las colaboraciones o los enlaces de comunicación, no presentan interés de cara a generar código. Las colaboraciones tienen por objeto clarificar la estructura y reparto de roles de determinadas interacciones en el sistema. En el caso de los enlaces de comunicación, no se indican cómo se implementan estos enlaces, sólo que existe una comunicación. Este enlace podría ser algo tan simple como un puntero o una conexión de red [OMG, 2004b]. No especifican formalmente y con completud cómo se resuelven en el sistema las interacciones representadas. Sin embargo, como muchos otras

construcciones de UML, tienen un alto valor de modelado entendido éste como una actividad de documentación, comunicación y clarificación de conceptos.

3.1.4.1.7. Expresiones en UML

UML permite especificar valores asociados a determinados aspectos de los modelos estructurales y de comportamiento. Por ejemplo, para definir el valor por defecto que toma el parámetro de una operación o una propiedad de un objeto, para definir el cuerpo de las restricciones UML (§ 3.1.5) o para establecer las condiciones que deben cumplirse para seguir un flujo de ejecución en un modelo de actividades (§ 3.1.4.2.2). Para especificar estos valores se define la metaclase `ValueSpecification`.

En la Figura 3.12 se recoge el diseño de los valores de especificación propuesto en el metamodelo de UML [OMG, 2004b]. Una especificación de valor (`ValueSpecification`) especifica un conjunto de instancias (tanto objetos como valores de datos). Existen 4 tipos de especificaciones de valor (Figura 3.12):

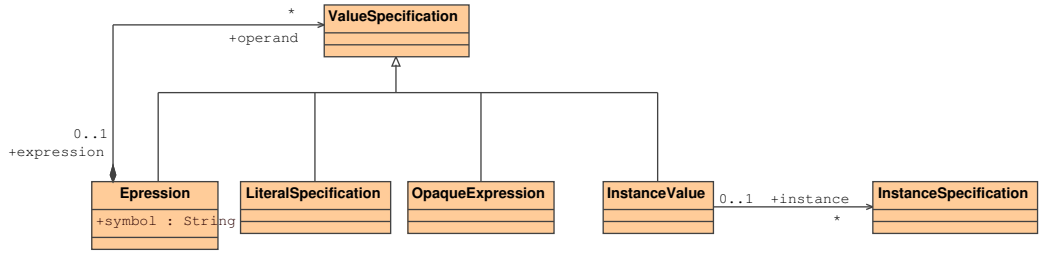

Figura 3.12: Metaclases que representan valores en UML.

Expresiones (`Expression`). Permiten representar un árbol estructurado (opcionalmente vacío) de símbolos que denota un conjunto de valores especificados en un contexto. Tal y como se muestra en la Figura 3.12, el diseño aplica el patrón *composite* [Gamma et al., 1995]: Un objeto `Expression` es un `SpecificationValue` que agrega a su vez otros objetos `SpecificationValue` que representan operandos. Este diseño permite que, recursivamente, otras expresiones sean operandos de la expresión [OMG, 2004b].

Un objeto `Expression` representa un nodo del árbol. Este nodo será terminal si no existen operandos agregados. Si existen operandos, entonces la expresión representa un operador a aplicar sobre ellos. En cualquier caso, la especificación establece que la interpretación del símbolo depende del contexto en el que evalúe la expresión.

Expresiones opacas (`OpaqueExpressions`). Se trata de expresiones en un lenguaje determinado que contienen directamente el código utilizado para describir un valor o conjunto de valores. El lenguaje puede ser natural o un lenguaje

de programación existente. Permite especificar las expresiones en más de un lenguaje y, opcionalmente, asociar a cada expresión una cadena de caracteres que identifique el lenguaje utilizado.

Literales (`LiteralSpecification`). Representan el modelo de una constante literal. Mediante metaclases hijas se representan tipos de literales: enteras (`LiteralInteger`), booleanas (`LiteralBoolean`), cadenas de caracteres (`LiteralString`), etc.

Instancias (`InstanceValue`). Es un valor de especificación que representa a una instancia de un clasificador UML. Esta especificación se recoge en un objeto del tipo `InstanceSpecification`. Además de recoger el tipo del clasificador UML, permite especificar valores para un conjunto de características estructurales del clasificador. Los objetos `InstanceSpecification` también se utilizan para representar los objetos en los diagramas de objetos § 3.1.4.1.2.

La especificación de UML, en su estado actual [OMG, 2004b], no define formalmente qué símbolos pueden utilizarse para formar expresiones, ni tampoco cómo se evalúan estas. Se trata de un aspecto ambiguo que obligaría a una implementación de herramienta MDA a acordar una especificación formal de ambos aspectos.

Las expresiones opacas vienen a ofrecer una solución provisional a este problema, permitiendo utilizar lenguajes de programación existentes. Sin embargo, si se utiliza un lenguaje perteneciente a alguna plataforma existente, si bien se facilitaría la generación de código para dicha plataforma, se violaría el principio fundamental de MDA de independencia respecto a la plataforma. UML adopta como lenguaje predefinido para las expresiones opacas OCL. Se trata de un lenguaje definido por el OMG para el que sí se ha definido un metamodelo formal y una notación concreta. OCL será analizado en OCL § 3.1.5.

3.1.4.2. Modelado de Comportamiento

UML ha adolecido históricamente de una ausencia de formalidad a la hora especificar comportamiento. Aunque ha incluido desde su creación diferentes tipos de diagramas para modelar los aspectos dinámicos de los sistemas, la definición de la semántica de estos diagramas no se realizaba de manera formal.

Por está razón se desarrolló una propuesta denominada *"UML Action Semantics"*[OMG, 2001], que fue aceptada por el OMG y posteriormente incluida como parte del estándar UML a partir de su versión 1.5 [OMG, 2003][7]. La versión 2.0 ha extendido y revisado *Action Semantics* y mejorado su integración en la especificación de UML.

La propuesta *Action Semantics* integra un metamodelo del comportamiento dinámico en el metamodelo de UML, y además proporciona un modelo de ejecución para dicho metamodelo [Sunye et al., 2001]. *Action Semantics* fue inicialmente concebido como una extensión a UML que definía el metamodelo de un lenguaje de

[7]La utilización de modelos UML con la capacidad de ser ejecutados ha sido denominada UML Ejecutable y será analizada en § 3.2.

acción cuyo propósito era conformar los cimientos de la semántica de la dinámica de UML. En UML 2, este metamodelo se integra en el metamodelo de UML y, sobre él, se edifica el metamodelo de los modelos de comportamiento UML. Por esta razón, antes de analizar las construcciones para modelar comportamiento de UML, conviene explicar los fundamentos de la semántica de acción de UML.

3.1.4.2.1. Modelo de Acciones

La unidad mínima de especificación de comportamiento en UML se denomina acción (`Action`) [OMG, 2004b]. De forma general, una acción toma un conjunto de entradas y las convierte en un conjunto de salidas, aunque cualquiera de los dos conjuntos o ambos pueden estar vacíos. UML utiliza el término "pin" para hacer referencia a los valores de entrada y salida (Figura 3.13). En UML se definen formalmente el metamodelo de diversos tipos de acciones, como las correspondientes a la invocación de operaciones (`CallOperationAction`), la creación de objetos (`CreateObjectAction`) o la escritura de atributos (`WriteStructuralFeatureAction`).

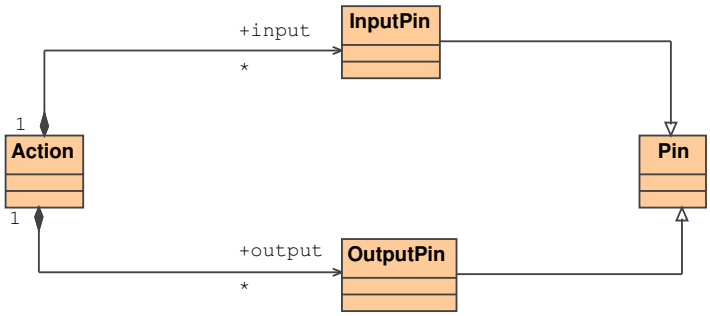

Figura 3.13: Relación de las acciones con los valores de entrada y salida en el metamodelo de UML

Un acción UML, se ejecuta en el contexto de un comportamiento (`Behavior`). Un comportamiento es una metaclase que define un contexto en el que se ejecutan acciones. Establece las restricciones y condiciones de ejecución, como por ejemplo, qué parámetros de entrada toma una acción [OMG, 2004b].

Cualquier comportamiento en UML es consecuencia directa de la acción de al menos un objeto. Un comportamiento describe como los estados de los objetos, descritos por el estado de sus características estructurales (`StructuralFeature`), cambian con el tiempo. Esto significa que los comportamientos no tienen existencia propia, siempre dependen de un objeto anfitrión. Existen dos tipos de comportamiento en UML:

Comportamiento de ejecución (*executing behavior*). Es el realizado por un objeto anfitrión como consecuencia directa de la invocación de una característica de comportamiento (`BehavioralFeature`) de dicho objeto.

Comportamiento emergente (*emergen behavior*). Este comportamiento resulta de la interacción de uno o más objetos participantes. Aunque por tratarse de un objeto está en última instancia referido a un objeto, puede ser especificado por clasificadores no instanciables. Por ejemplo, de un interfaz para describir el comportamiento de las objetos que lo realicen.

La especificación realizada por el comportamiento emergente puede ser utilizada ocasionalmente para generar código. Sin embargo, para generar aplicaciones completas tiene mayor interés la especificación exacta del código que debe contener cada uno de los métodos de cada objeto de la aplicación. Por ello, de cara a esta investigación sobre MDA el tipo de comportamiento más interesante es el comportamiento de ejecución (*executing behavior*), que describe la ejecución de las características de comportamiento de los objetos.

Los diferentes subtipos de la metaclase `Behavior` proporcionarán diferentes mecanismos para especificar comportamiento. UML define define diversos mecanismos de especificación [OMG, 2004b]:

- Grafos de tipo Petri a través de actividades (§ 3.1.4.2.2).
- Autómatas a través de máquinas de estados (§ 3.1.4.2.3).
- Secuencias parcialmente ordenadas de ocurrencias de eventos (interacciones) (§ 3.1.4.2.4).
- Descripciones informales mediante *Casos de Uso* (§ 3.1.4.2.5).

También se podrán definir nuevos estilos de especificación de comportamiento a través de perfiles UML (§ 3.1.6). No todos los mecanismos para especificar comportamiento tienen la misma naturaleza ni el mismo poder expresivo. Por ejemplo, una descripción podría consistir en describir la ocurrencia de eventos que se desprende de la ejecución de comportamientos y otra en describir la máquina que induciría dichos eventos.

Los comportamientos son invocados en UML mediante la invocación de acciones especializadas. En la Figura 3.14 se recoge el metamodelo de las dos acciones de invocación fundamentales: `CallBehaviorAction`, que permite invocar directamente comportamientos y `CallOperationAction`, que permite invocar operaciones de objetos desencadenando la invocación de los comportamientos asociados a éstas. El disponer de ambas posibilidades de invocación de acciones facilita la utilización de UML en entornos no orientados a objetos, resolviendo uno de los problemas planteados por Conrad Bock en [Bock, 1999a].

Un aspecto interesante de cara a MDA de *Action Semantics*, es que permite definir formalmente qué se ejecuta dentro del cuerpo de los métodos de las clases, unos de los problemas tradicionales existentes con UML. La especificación de UML [OMG, 2004b] establece que, cuando se asocia un comportamiento (`Behavior`) con una característica de comportamiento (`Behavioral Feature`), éste define la implementación de dicha característica, es decir, la computación que genera los efectos de la característica de comportamiento.

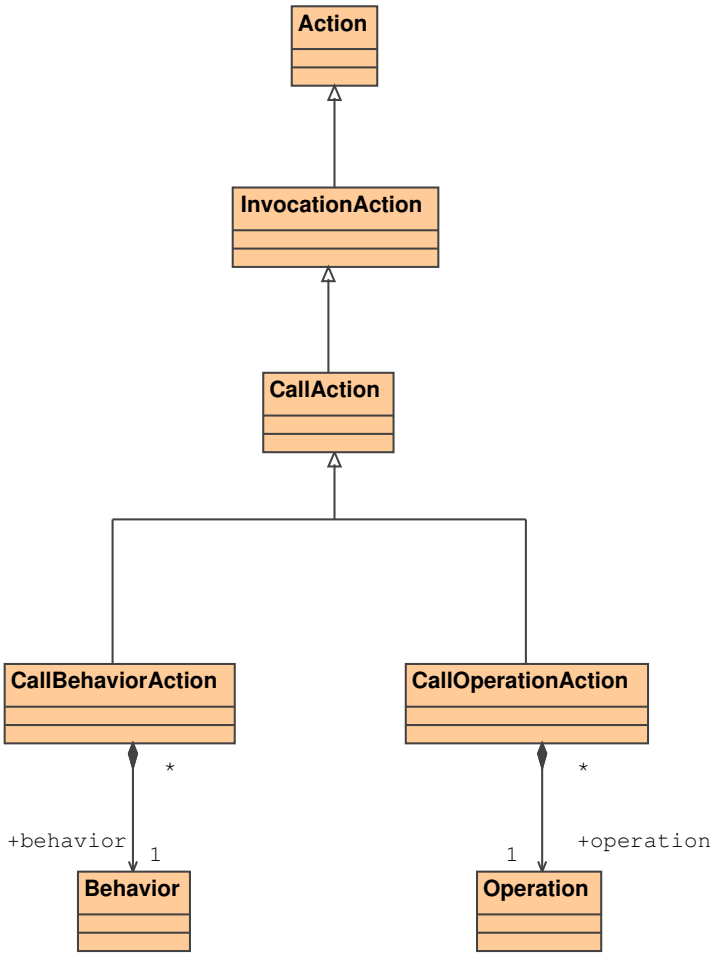

Figura 3.14: Metamodelo de acciones de invocación de comportamiento

Para ilustrar esta especificación se expondrá un ejemplo de como se relacionan en el metamodelo de UML operaciones y comportamientos. Lo que hace *Action Semantics* es extender la metaclase `BehavioralFeature`, padre de la metaclase `Operation`. En dicha superclase define una asociación etiquetada como `method` con una instancia de la metaclase `Behavior`. De este modo, asocia a cada operación un objeto del tipo `Behavior` que especifica un contexto de ejecución de acciones. Además, se extiende la metaclase `Classifier` y se define `BehavioredClassifier` : clasificador que puede contener diversos objetos `Behavior` uno de los cuales se invocará cada vez que se cree una instancia del clasificador[8]. En general, cualquier objeto que puede alojar comportamiento se especifica mediante un subtipo concreto de `BehavioredClassifier`. El diagrama de clases de este diseño se recoge en la Figura 3.15.

[8]Se correspondería con un constructor en los lenguajes de programación orientados a objetos actuales.

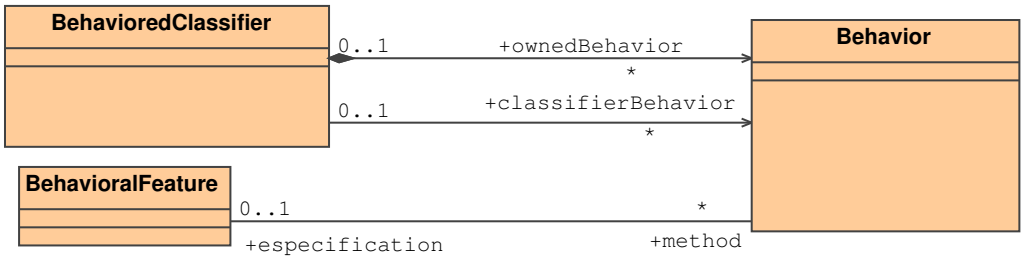

Figura 3.15: Asociación de Comportamiento a Clasificadores y Operaciones en el metamodelo de Action Semantics

Una consecuencia de este diseño es que las operaciones pueden ligarse a cualquier tipo de comportamiento definido en UML.

3.1.4.2.2. Diagrama de Actividad

Los diagramas de actividad UML permiten describir lógica procedural, procesos de negocio y flujos de trabajos (*workflow*) [Fowler, 2003d]. Este tipo de diagramas UML se inspira en los diagramas de flujo (*flowcharts*) [IBM, 1970], tradicionalmente utilizado para representar lógica condicional en los programas de ordenador.

En versiones anteriores de UML, los diagramas de actividad se consideraban un caso especial de los diagramas de estados. En la versión 1.5 de UML se definían los diagramas de actividad como una "vista extendida" de los diagramas de estados UML [OMG, 2003]. El considerar los diagramas de actividad como un sistema de transición de estados dificultaba la utilización de los mismos para modelar flujos de trabajo. En UML 2 esta restricción fue eliminada, y los diagramas de actividad tienen su propia semántica de ejecución, que está integrada con el modelo de acciones de *Action Semantics*.

El modelo de actividades de UML 2 sigue el enfoque tradicional de flujo de información de control y de datos mediante el cual se inician determinados comportamientos cundo otros finalizan y cuando se disponen de las entradas necesarias. La especificación de UML utiliza el término general *"token"* (señal) para designar valores de datos y de control [OMG, 2004b]. Se sigue una semántica de flujo de *tokens* inspirada en las redes de Petri [Peterson, 1981].

Para el modelo de actividades, UML 2 define lo que Conrad Bock denomina máquinas virtuales "intuitivas" [Bock, 2003a]. Su objetivo es permitir a usuarios y fabricantes predecir cuáles serán los efectos en tiempo de ejecución de sus modelos. Se trata de una máquina virtual basada en el encaminamiento de datos y control a través de un grafo de nodos conectados por los extremos. Cada nodo y extremo define cuándo la información de control y los valores de datos se mueven a través de ellos. Combinando todos estos movimientos de puede predecirse el comportamiento del grafo completo.

El único objetivo de las reglas que determinan el movimiento de datos e información de control es predecir el efecto en tiempo de ejecución, es decir, cuándo se desencadenarán comportamientos y con qué entradas. En concreto, estas reglas no implican que los modelos de actividades tengan que implementarse como una máquina virtual que responda al modelo de *tokens*, de paso de mensajes o cualquier otro mecanismo. El único requisito es que los efectos en tiempo de ejecución que predice la máquina virtual realmente ocurran en la implementación [Bock, 2003a].

Una actividad UML especifica la secuencia y condiciones necesarias para coordinar comportamientos de bajo nivel definidos por acciones UML (§ 3.1.4.2). Las actividades son el único comportamiento definido en UML que puede contener directamente acciones. Al igual que las acciones, las actividades pueden ejecutarse, lo que implicará en última instancia la ejecución de todas las acciones que contengan [OMG, 2004b]. UML contempla un tipo de acción que puede invocar directamente a comportamientos UML (`CallBehaviorAction`). Como una actividad es un tipo de comportamiento, este diseño permite descomponer las acciones que coordina una actividad como subactividades que las especifican.

En la Figura 3.16 se recoge la definición de la metaclase `Activity` en el metamodelo de UML. Una actividad es un comportamiento UML (`Behavior`) que agrega nodos (`ActivityNode`). Las acciones UML (`Action`) son nodos. El modelo de actividades redefine además la metaclase `Behavior` para que acepte parámetros (`Parameter`).

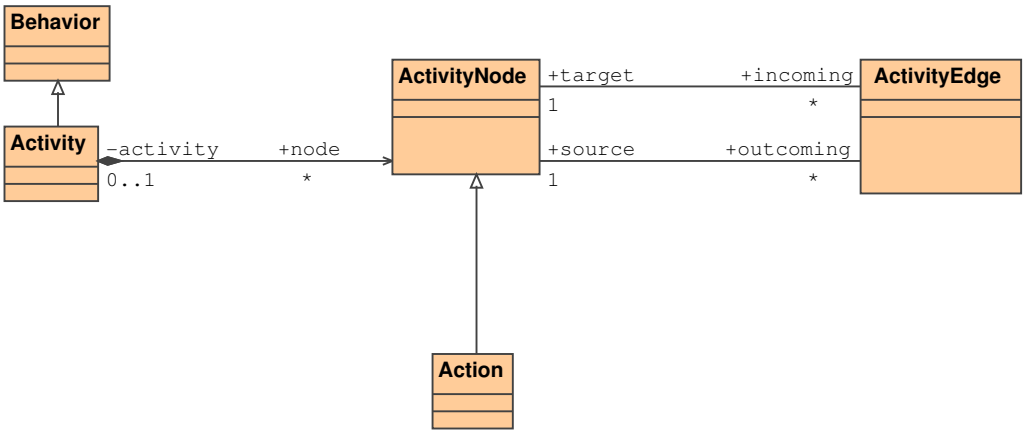

Figura 3.16: Actividades en el metamodelo UML

Los nodos de las actividades estarán conectados por extremos (`ActivityEdge`) para conformar un grafo de flujos. Cada extremo representa una conexión dirigida entre dos nodos. Los datos de control y de información fluyen a lo largo de los extremos y los nodos operan sobre ellos. Existen tres tipos de nodos en los modelos de actividades [Bock, 2003a]: nodos de acción, nodos de control y nodos objeto.

Nodos de control (`ControlNode`). Comprenden construcciones destinadas a controlar el flujo de ejecución de nodos. Permiten coordinar los flujos en una actividad. Existen diferentes tipos de nodos de control: nodos inicial (`InitialNode`) y final (`FinalNode`), nodos de bifurcación (`ForkNode`) y de unión (`JoinNode`), nodos que aceptan múltiples flujos de entrada (`MergeNode`) o nodos que permiten elegir entre diferentes bifurcaciones salientes (`DecisionNode`) (Figura 3.17).

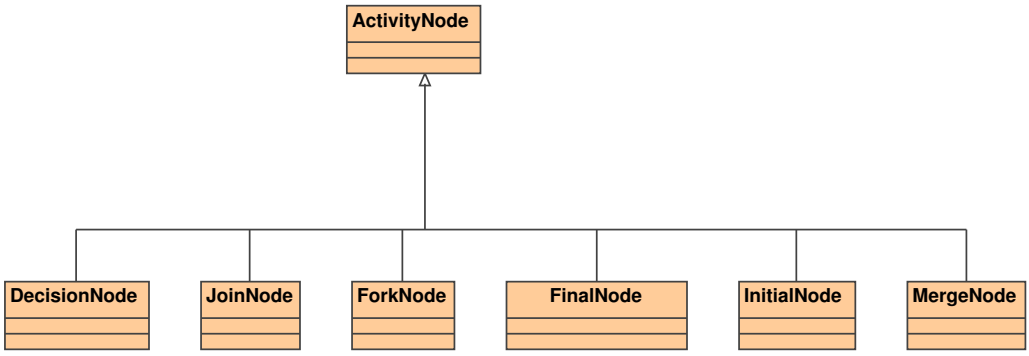

Figura 3.17: Especificación de Nodos de control de actividad

Resulta interesante estudiar cómo especifica UML formalmente las condiciones que deben satisfacerse para tomar un flujo u otro en los nodos de control de decisión. De cada a una herramienta MDA, se necesita poder especificar las condiciones de los flujos de una manera completa y sin ambigüedad. En la especificación UML estas condiciones se modelan mediante guardas (*guards*) en los extremos que conectan los nodos [OMG, 2004b]. Estas guardas son valores de especificación (§ 3.1.4.1.7) booleanos que, evaluados en tiempo de ejecución, determinan si la información de control y los datos pueden atravesar el extremo.

En la práctica, para representar condiciones se utilizan habitualmente expresiones informales que son interpretadas a la hora de implementar el diagrama. Este enfoque, recogido en el metamodelo de UML por las "expresiones opacas" (§ 3.1.4.1.7), no es válido para MDA por su ambigüedad.

Un nodo de decisión además puede definir un comportamiento que define la "entrada de la decisión" (asociación `decisionInput`) [OMG, 2004b]. En tal caso, cada *token* de datos que llega al nodo será pasado a dicho comportamiento antes de que se evalúen las guardas en los extremos salientes. La salida del comportamiento estará disponible para cada guarda. Puede utilizarse este comportamiento para implementar el proceso de decisión, cuando los valores de especificación UML se muestren insuficientes.

Nodos objeto (`ObjectNode`). Almacenan temporalmente datos mientras que éstos fluyen por el grafo. Su objetivo es es modelar los parámetros que se pasan

entre acciones dentro de una actividad o que la actividad recibe externamente. Para ello, el metamodelo de actividades de UML redefine la metaclase `Pin` para que especialice `ObjectNode`.

Nodos de acción (`ActionNode`). Estos nodos operan en los datos de información y control que reciben, y a la vez pueden proporcionar datos a otras acciones. Los nodos de acción en el metamodelo de UML se modelan haciendo que la metaclase `Action` especialice la clase `ActivityNode`. Además, otra clase padre de `Action` será la metaclase `ExecutableNode`. Ésta es una clase abstracta para nodos de actividad que pueden ser ejecutados. Un nodo ejecutable puede agregar con un conjunto de manejadores de excepciones (`ExceptionHandler`) que capturarán las excepciones que se propaguen al nivel contenedor del nodo ejecutable.

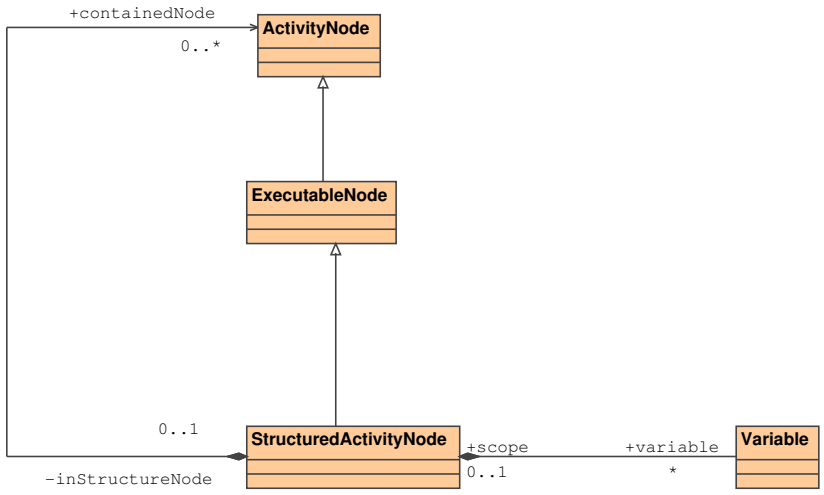

Figura 3.18: Actividades estructuradas en el metamodelo de UML

Además de las acciones, el otro tipo de nodos que pueden ser ejecutados son los nodos de actividades estructuradas (`StructuredActivityNode`). Los nodos de actividades estructuradas son nodos de actividades ejecutables que pueden expandirse en actividades subordinadas [OMG, 2004b]. Son el único tipo de nodo definido en la especificación que puede contener otros nodos no pudiendo los nodos contenidos formar parte de ningún otro nodo contenedor, salvo en nodos estructurados anidadas[9]. Dentro de un nodo estructurado pueden declararse variables para las cuales el nodo establecerá su ámbito de existencia.

En la Figura 3.18 se muestran los aspectos básicos del metamodelo de los nodos estructurados UML. El anidamiento de nodos que a su vez pueden ser es-

[9]Dicho de otra forma, una acción puede estar contenida por varios nodos estructurados, pero sólo habrá uno que inmediatamente la contenga.

tructurados se consigue aplicando el patrón *Composite* [Gamma et al., 1995]. Bajo un punto de vista de generación de código, los nodos estructurados son un modelo natural para los bloques en los lenguajes de programación habituales, donde existen delimitadores de apertura y cierre que casan de una forma bien anidada y sin ambigüedades [Bock, 2005].

Existen diferentes tipos de nodos estructurados representados mediante metaclases especializadas. Tres tipos de nodos que tendrán su correspondencia directa en los lenguajes de programación habituales son (Figura 3.19):

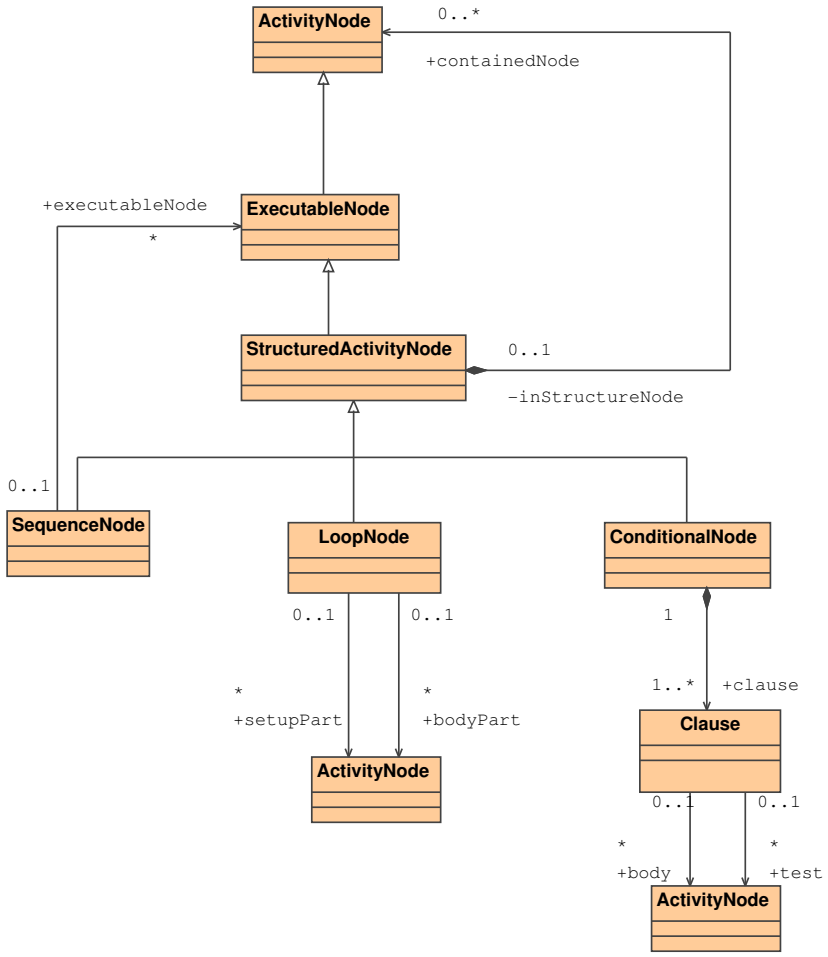

Figura 3.19: Tipos de actividades estructuradas

Nodos de secuencia (`SequenceNode`). Permiten agrupar un conjunto de nodos ejecutables que serán ejecutados en orden.

Nodos bucle (`LoopNode`). Representan bucles compuestos por tres secciones modeladas a su vez como nodos de actividad:

Configuración (*setup*). Contiene acciones a ejecutar al comienzo del bucle, típicamente dedicadas a inicializar valores o ejecutar otras tareas de configuración inicial.

Examen (*test*). Comprende acciones que se ejecutan al comienzo o al final del bucle (según determine una variable booleana). Estas acciones establecerán el valor de una variable booleana que determinará si se debe ejecutar el cuerpo del bucle.

Cuerpo (*body*). Contiene las acciones a realizar en cada iteración del bucle.

Nodos condicionales (`ConditionalNode`). Los nodos condicionales representan una elección entre un determinado número de alternativas. Se modelan mediante una agregación de cláusulas (`Clause`). Cada cláusula contiene una sección de examen (*test*) y un cuerpo (*body*). Del mismo modo que con los nodos bucle, las acciones contenidas en la sección de examen determinan si se ejecutan las acciones contenidas en el cuerpo. Las cláusulas se ejecutarán en estricto orden. Las cláusulas `else` de los lenguajes de programación habituales se modelarán mediante una cláusula sin sucesor y con una condición que siempre es verdadera.

Mientras que los nodos de control y los nodos objeto están orientados a lenguajes de flujos gráficos (modelos de flujos), los nodos estructurados describen modelos apropiados para lenguajes que habitualmente tienen una presentación textual (modelos estructurados). Ambos modelos son distintos pero no independientes, ya que el modelo estructurado hace uso en muchas ocasiones del flujo de objetos para pasar información entre acciones [Bock, 2005]. Este hecho hace que ambas notaciones no representen formalmente una vista del mismo modelo lo que dificulta la transformación entre ellas.

Por otra parte UML no define una notación estándar para la mayor parte de modelos del nivel estructurado. La razón es que estos modelos están pensados para ser generados a partir de lenguajes de programación o lenguajes de acción específicos [Bock, 2005]. Esto hace que el metamodelo de los nodos estructurados sea muy interesante de cara a esta investigación: define modelos para artefactos habituales del código fuente tales como bucles o condiciones. Estos modelos son apropiados para su transformación sobre lenguajes de programación concretos. En particular, elimina las ambigüedades que surgían con los nodos de decisión y las expresiones vistas con los nodos de control. Una conclusión interesante de cara a esta investigación, es que la semántica del modelo estructurado de las actividades UML es lo suficientemente potente como para describir la lógica interna de operaciones complejas.

Analizando el uso de los diagramas de actividades la construcción de modelos ejecutables o transformables en código se detecta que las herramientas UML actuales más populares no contemplan la generación de código a partir de diagramas de actividad. Este es el caso de Borland Together 2006 [Borland, 2006] o MagicDraw 11 [Magic, 2006]. Los diagramas de actividades sí han sido utilizados para modelado formal de comportamiento en dominios concretos. Así, se han pro-

puesto extensiones a los mismos para el modelado del comportamiento de agentes [Odell et al., 2000] y para sistemas de tiempo real [Douglass, 1999, Eshuis and Wieringa, 2001].

AndroMDA [AndroMDA, 2006] utiliza diagramas de actividad extendidos para la generación de aplicaciones Web utilizando Struts [Apache, 2006a]. Son el mecanismo utilizado para especificar la implementación de los modelos de casos de uso utilizados en el cartucho de Struts de AndroMDA para organizar lógicamente las vistas de la aplicación (§ 3.1.4.2.5). AndroMDA propone una serie de estereotipos (§ 3.1.6) con los que extender los diagramas de actividad. Las acciones de los diagramas de actividad se usan para representar tanto controladores (acciones Struts) como vistas (páginas JSP)[10], diferenciadas por la utilización de un estereotipo `FronEndView`. El flujo de cómo los controladores redirigen a las vistas, y como las vistas se enlazan entre sí se modela mediante transiciones[11] entre acciones. Las transiciones también pueden ir estereotipadas, por ejemplo, para mostrar mensajes de advertencia en la página dirigida. Otro tipo de estereotipo en las transiciones, utilizados cuando éstas salen de una vista, es utilizado para indicar parámetros que se envían a un controlador, típicamente obtenidos de un formulario.

Debe señalarse que en los ejemplos comentados de utilización y extensión de los diagramas de actividad para modelado formal de comportamiento, se parte de la antigua especificación de los diagramas de actividad como una vista extendida de los diagramas de estados, drásticamente modificada en UML 2. Antes de la integración del modelo de actividades con *Action Semantics* hubo intentos de formalizar la semántica de ejecución de los diagramas de actividad entendidos como máquinas de estados [Börger et al., 2000].

También se han utilizado diagramas de actividad para modelado de flujos de trabajo (*workflow*) y procesos de negocio[12] (*Business Process Modeling* o BPM). En [Dumas and ter Hofstede, 2001] se analizan los aspectos positivos y carencias de este tipo de diagramas para el modelado de flujos de trabajos. En este sentido, AndroMDA [AndroMDA, 2006] ofrece un cartucho que permite generar definiciones de procesos para el motor de *workflow* de JBoss (jBPM) [JBoss, 2006].

3.1.4.2.3. Diagrama de Máquina de Estados

Los diagramas de máquinas de estados UML representan los estados en los que un objeto puede estar así como las transiciones entre dichos estado [Ambler, 2004].

[10] Struts ofrece una implementación parcial del patrón Modelo Vista Controlador (MVC) [Buschmann et al., 1996].

[11] En UML 2, los extremos que salen y entran de los nodos (`ActivityEdge`) sustituyeron las transiciones de las versiones anteriores de UML. AndroMDA utiliza el metamodelo de UML versión 1.5, pues es el máximo que ofrece la implementación de MOF que utiliza: Netbeans MDR [NetBeans, 2006a].

[12] Los diagramas de actividad son una de las múltiples notaciones estándar existentes para especificar procesos de negocio. Ante la multitud de notaciones y consorcios de estandarización existentes, se desarrolló una notación estándar "unificada" denominada Business Process Modeling Notation (BPMN) que actualmente mantiene el OMG [OMG, 2006a], por lo que no se descarta su integración con los diagramas de actividad UML en el futuro.

Son una técnica común a otros lenguajes de modelado y también una de las primeras adoptadas por las notaciones orientadas a objetos para especificar comportamiento. Por esta razón, fue la primera técnica que contemplaron los creadores de UML para modelar comportamiento [Bock, 1999b].

Las construcciones manejadas en los diagramas de estados UML permiten el modelado de comportamiento discreto mediante sistemas de transición de estados finitos. La especificación de UML contempla dos tipos de máquinas de estados [OMG, 2004b]:

Máquinas de estados de comportamiento (*behavioral state machines*). Permiten especificar el comportamiento de diversos elementos de modelado. Por ejemplo, de instancias de clases. UML formaliza para estas máquinas de estados una variante orientada a objetos de los diagramas de estados definidos por David Harel [Harel, 1987].

Máquinas de estados de protocolo (*Protocol State Machines*). Se usan para expresar protocolos de uso de partes del sistema. En esta caso, representan las transiciones legales que puede disparar un clasificador dado. Este tipo de máquinas de estado son adecuadas para representar el ciclo de vida de un objeto, sin incluir ningún tipo de información acerca de la implementación de su comportamiento.

Las máquinas de estados modelan el comportamiento como el recorrido de un grafo de nodos de estado interconectados por uno o más arcos de transición. Las transiciones se disparan por la ocurrencia de eventos. Durante el recorrido, la máquina de estados ejecuta una serie de actividades UML (§ 3.1.4.2.2) que pueden asociarse a diversos elementos de la máquina de estados.

En el metamodelo de UML una máquina de estados (`StateMachine`) es un comportamiento UML (`Behavior`). Una máquina agrega regiones (`Region`) las cuales a su vez contienen estados (`State`) y transiciones (`Transition`). En la Figura 3.20 se recoge el diseño realizado en el metamodelo de UML. Los nodos conectados por transiciones en una máquina de estados son vértices (`Vertex`) y un estado es un tipo de vértice. Un estado además puede contener a su vez regiones, siendo en tal caso un estado compuesto (*composite state*). Un estado compuesto es por lo tanto un superestado que puede descomponerse en otros estados y transiciones.

Las *máquinas de estados de comportamiento* se representan como instancias de la metaclase `StateMachine`. El clasificador (`BehavioredClassifier`) que defina el contexto del comportamiento especificará qué disparos de señales y de llamadas se definen para la máquina de estados, y qué atributos y operaciones estarán disponibles para las actividades de la máquina de estados. Estos disparos se definirán de acuerdo con las operaciones del clasificador [OMG, 2004b]. Como cualquier comportamiento UML, la máquina de estados puede asociarse a una característica de comportamiento (`BehavioralFeature`). En este caso, define el comportamiento de esta característica de comportamiento, coincidiendo en este caso los parámetros de

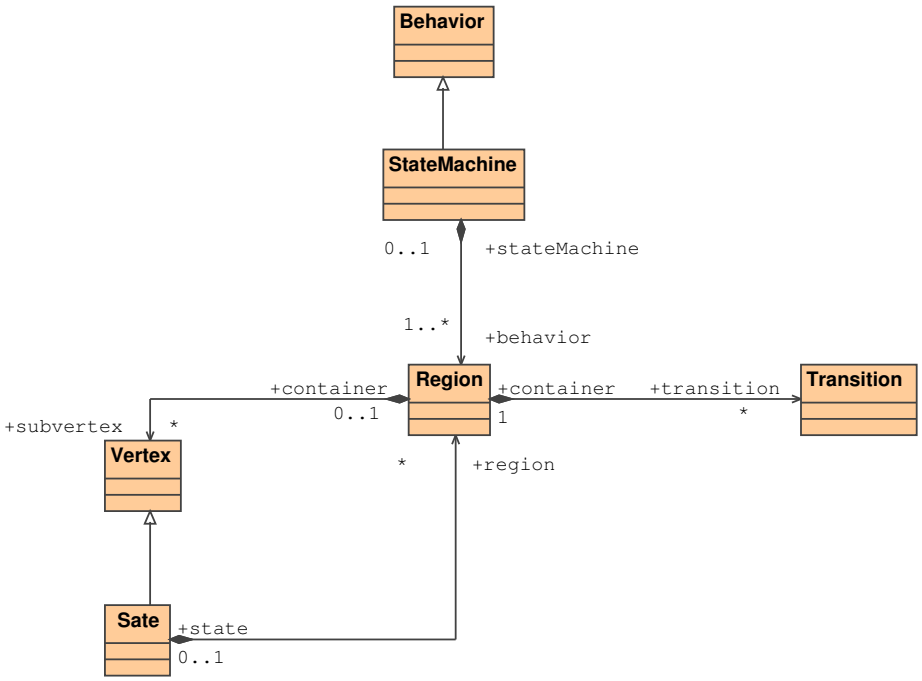

Figura 3.20: Máquinas de estados definidas en el metamodelo de UML

la máquina de estados con los de la característica de comportamiento. Si la máquina de estados no tiene un clasificador que especifique su contexto puede utilizar disparos que sean independientes de las operaciones de un clasificador.

Los estados representan situaciones durante las cuales se mantiene algún tipo de condición invariante (habitualmente implícitas aunque pueden hacerse explícitas mediante restricciones § 3.1.5). Los estados pueden llevar opcionalmente asociados tres comportamientos Figura 3.21:

1. Entrada (*entry*). Comportamiento a ejecutar cuando se llega al estado.
2. Salida (*exit*). Comportamiento a ejecutar cuando se sale del estado.
3. Actividad a realizar (*do activity*). Comportamiento que se ejecuta mientras se está en el estado. Comienza a ejecutarse cuando se entra en el estado y se mantienen ejecutándose hasta que finaliza por sí mismo o hasta que se sale del estado.

Existen un tipo especial de vértices denominados pseudoestados (`Pseudoestate`). Los pseudoestados tienen un atributo `kind` que indican de qué tipo se tratan. Por ejemplo, los estados iniciales y los estados elección de alternativa (*choice*) son pseudoestados. Los estados finales se representan con una metaclase que especializa `State`: `FinaState`. Cuando se alcanza un estado final se completa el comportamiento representado en la región que lo contiene.

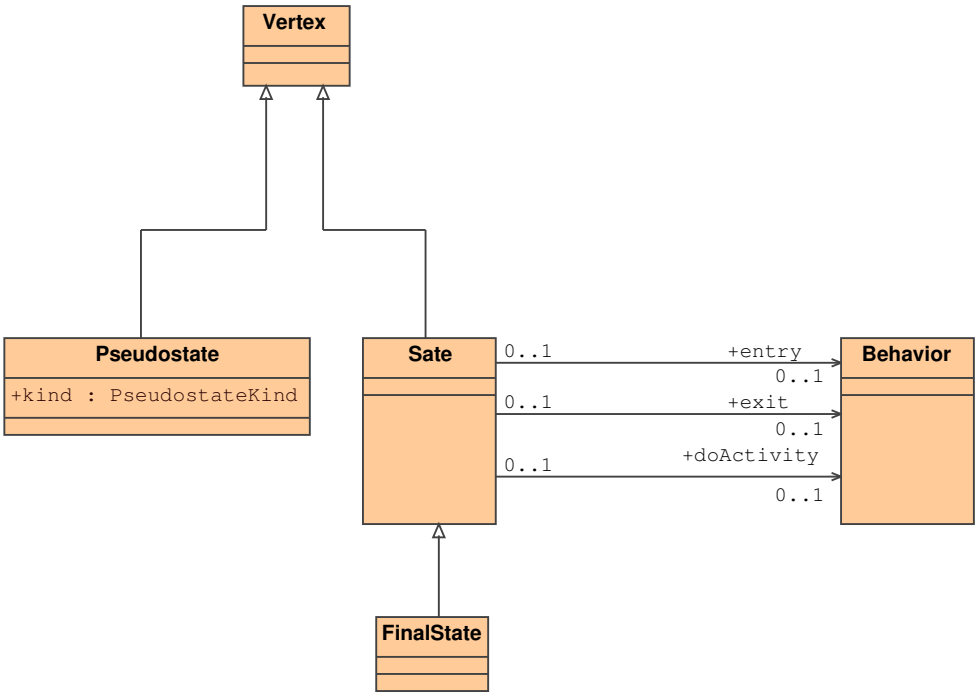

Figura 3.21: Comportamientos asociados a un estado

Junto con los estados, el otro elemento básico de los diagramas de máquina de estados son las transiciones. Una transición es una relación dirigida entre un vértice origen y un vértice destino [OMG, 2004b]. En la Figura 3.22 se recoge el diseño de los aspectos básicos del metamodelo de transiciones. Una transición especifica qué disparadores (`Trigger`) pueden dispararla. Un disparador especifica un evento (`Event`) que puede causar la ejecución del comportamiento asociado.

Una transición puede especificar también una guarda (*guard*). Una guarda es una restricción (`Constraint`) que es evaluada cuando se despacha un evento, y que debe ser verificada para que pueda habilitarse la transición. El efecto de una transición se modela como un comportamiento (`Behavior`). Este comportamiento será ejecutado cuando la transición se dispare.

Para que una transición se habilite debe verificarse que [OMG, 2004b]:

- Todos los estados origen de la transición se encuentren activados.
- Se satisfaga uno de los disparadores de la transición. Esto se produce si sucede un evento cuyo tipo coincida con el que el disparador especifica.

El modelo de comportamiento de UML define una metaclase `Event` padre de todos los eventos que pueden producirse. Un tipo es el evento que se produce cuando un objeto recibe un mensaje que invoca una llamada a una operación (`CallEvent`).

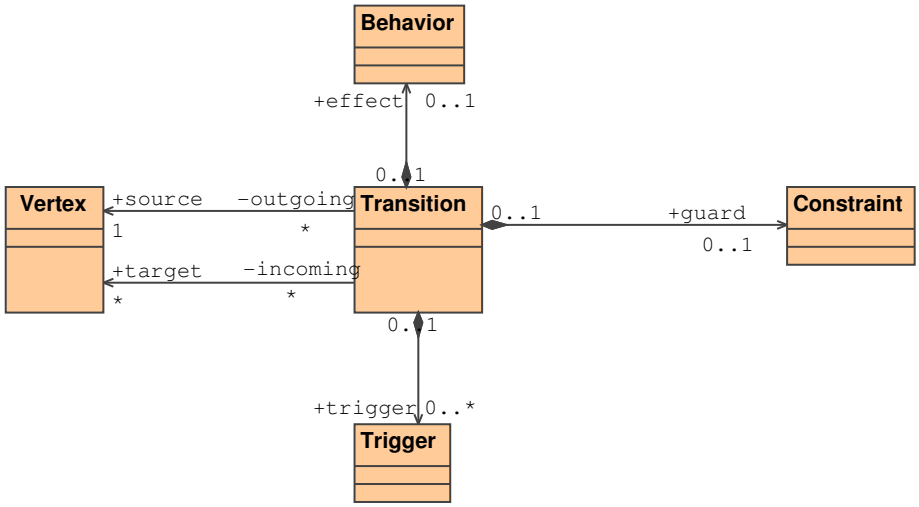

Figura 3.22: Transiciones de estados en el metamodelo de UML

Otro tipo de evento se produce cuando un objeto recibe una señal UML (`SignalEvent`)[13].

Una vez que una transición se dispara se produce una secuencia de pasos:

1. Se deja de estar en el estado origen de la transición y se ejecuta el comportamiento correspondiente.
2. Se ejecutan los efectos de la transición.
3. Se entra en el estado destino de la transición y se ejecuta el comportamiento correspondiente.
4. Poner ejemplo de las implementaciones.

A la hora de implementar un diagrama de estados existen diferentes técnicas. Las más habituales son 3 [Fowler, 2003d]:

***Switch* anidados.** Consiste en utilizar la construcción `switch` presente en la mayoría de lenguajes de programación actuales, o una equivalente[14]. Mediante este enfoque se comprobaría una variable que indicase en qué estado se encuentra el objeto y, en función de su valor, se ejecutarían las actividades a realizar en el estado activo. Dentro de cada estado, se comprobaría la llegada de eventos que pudiesen provocar la transición hacia otro estado. El problema de esta

[13]Una señal dispara una reacción en el objeto que la recibe. A diferencia de las operaciones, la reacción se produce asíncronamente y sin respuesta, sin que el emisor de la señal se quede bloqueado esperando que finalice [OMG, 2004b].

[14]Una sentencia `switch` puede ser construida utilizando únicamente sentencias condicionales `if`.

implementación es que las estructuras `switch` tienden a crecer rápidamente y volverse difíciles de manejar y de entender [Gorlen and Plexico, 1990][15].

Patrón de diseño *State*. Este enfoque se basa en utilizar el patrón *State* [Gamma et al., 1995] para implementar la máquina de estados. Esta patrón propone modelar cada estado como un objeto independiente. Los estados implementan un interfaz común que contiene las operaciones cuya ejecución depende del estado. En la implementación de una máquina de estados, un objeto controlador seleccionará el objeto estado adecuado cada vez que se produzca una transición. Cuando se produzca un evento, este objeto simplemente delegará el el objeto estado correspondiente la reacción adecuada.

Tablas de estados. Las tablas de estados [Samek, 2002] representan la información del diagrama de máquina de estados como datos. Un convenio utilizado es que cada file represente un estado y cada columna represente un evento [Mellor and Balcer, 2002, Raistrick et al., 2004]. El contenido de las filas especificará qué ocurre cuando un objeto en un estado dado detecta un evento particular. Las tablas de datos ofrecen una vista distinta de los diagramas de estados pero contienen la misma información. Si como resultado de un proceso de generación de código se generase una tabla de estados, habría que desarrollar un intérprete que la ejecutase. El intérprete podría reutilizarse con diferentes tablas de estados y la tabla de estados podría modificarse en tiempo de ejecución, pero se seguiría necesitando desarrollar el intérprete, tarea para la cual serían de aplicación las dos técnicas anteriores.

Analizando la utilización de diagramas de máquinas de estados para la construcción de modelos formales, se observa que los diagramas de estados has sido utilizados intensivamente para la construcción de autómatas ejecutables en el ámbito de sistemas en tiempo real, sistemas reactivos y sistemas empotrados [Selic et al., 1994][16], [Selic, 1998, Harel and Politi, 1998, Samek, 2002]. Se han utilizado también para modelado de flujo de trabajos [Mok and Paper, 2002], aunque en este área se han mostrado más adecuados los diagramas de actividades (§ 3.1.4.2.2).

Por otra parte, se ha propuesto utilizar modelos de máquinas de estados dominios más específicos. En [Horrocks, 1999] se propone utilizar máquinas de estados para especificar formalmente los controladores de las vistas a la hora de construir interfaces gráficos de usuario. En [Hartmann et al., 2004] se plantea utilizar diagramas de estados para especificar los escenarios de interacción de cada componente software, con el fin de generar y ejecutar sus tests funcionales automáticamente.

Fuera del ámbito de dominios específicos, los diagramas de máquinas de estados UML son utilizados intensivamente en UML Ejecutable. Como se verá en

[15] En este libro se llega a utilizar la expresión "switch statement considered harmful" en una referencia al título de famoso artículo de Edsger W. Dijkstra acerca de las sentencias `goto` [Dijkstra, 1968].

[16] Aunque en 1994 no se había creado UML, en esta obra se propone hacer uso intensivo de modelos de máquinas de estados de Harel [Harel, 1987], en los que se basan las máquinas de estados UML, para modelado de sistemas en tiempo real.

§ 3.2, UML Ejecutable utiliza los diagramas de estados UML como mecanismo principal para modelar el comportamiento de las clases [Mellor and Balcer, 2002]. Mediante diagramas de máquinas de estados se modela el ciclo de vida de las clases cuyo comportamiento varía con el tiempo y mediante un lenguaje de acción que implemente Actions Semantics se modelan tanto las actividades a realizar cuando se activa un determinado estado en un objeto, como las operaciones de las clases cuyo comportamiento no varía con el tiempo.

3.1.4.2.4. Diagramas de Interacción

Los diagramas de interacción describen como colaboran grupos de objetos en un determinado comportamiento [Fowler, 2003d]. En UML 2 existen 4 tipos diferentes de diagramas de interacción. Los más conocidos son el diagrama de secuencia y el de comunicación (antiguamente denominado "de colaboración"), existentes en versiones previas de UML. Los dos diagramas nuevos en UML 2 son el de *timing* y el de perspectiva de la interacción:

Diagrama de Secuencia. Muestra el paso de mensajes entre un conjunto de objetos ordenados temporalmente. El orden temporal se obtiene al mostrar la línea de vida (*lifeline*) de cada objeto verticalmente y observando los mensajes que salen y llegan de ella de arriba hacia abajo.

Diagrama de Comunicación. Este tipo de diagramas muestra el intercambio de mensajes entre un conjunto de objetos, pero en lugar de hacer énfasis en el orden temporal de los mensajes, se centra en los enlaces de intercambio de datos que se establecen entre los diversos participantes en la interacción.

Diagrama de *Timing*. Se trata de un tipo de diagrama de interacción centrado en representar las restricciones temporales existentes, tanto para objetos aislados como para conjuntos de objetos. Son útiles para mostrar restricciones temporales entre los cambios de estado de los objetos involucrados [Fowler, 2003d].

Diagrama de Perspectiva de la interacción (*interaction overview*). Estos diagramas son una mezcla de los diagramas de actividad (§ 3.1.4.2.2) y los diagramas de secuencia. Se utiliza la notación de los diagramas de actividad para mostrar el control del flujo y las actividades son explotadas por diagramas de secuencia.

Todos estos diagramas representan la notación gráfica de un metamodelo de interacciones común. A continuación analizaremos los aspectos de este metamodelo centrado en los diagramas de secuencia y de comunicación. No se analizará los aspectos del metamodelo específicos a los diagramas de *timing*, de aplicación en dominios muy reducidos, habitualmente relacionados con sistemas en tiempo real o sistemas electrónicos de control. Los diagramas de Perspectiva de la Interacción comparten metamodelo con los diagramas de actividad y de secuencia.

En la Figura 3.23 se recogen los aspectos básicos del metamodelo de interacciones. Una interacción (`Interaction`) es una unidad de comportamiento UML (`Behavior`) que se centra en el intercambio de información observable entre instancias de clasificadores [OMG, 2004b]. Una interacción es y, a la vez, está compuesta de fragmentos de interacción (`InteractionFragment`). Un fragmento de interacción es una clase abstracta que denota un "trozo" de interacción. Se trata de un artefacto de diseño aplicando el patrón *Composite* [Gamma et al., 1995]. Una interacción puede estar compuesta de otras interacciones y de otros elementos que especialicen `InteractionFragment`.

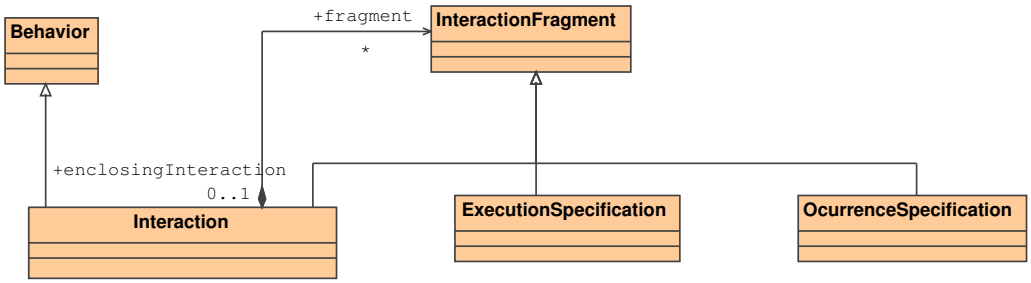

Figura 3.23: Aspectos básicos del metamodelo de Interacciones

El modelo de interacción de UML define una semántica de traza (*trace*) diferente al modelo de ejecución de UML pero relacionada con éste. Una traza en el contexto de interacciones UML significa una secuencia de ocurrencia de eventos [OMG, 2004b]. En relación con el modelo de ejecución de UML, una traza es un comportamiento emergente (§ 3.1.4.2.1). Cada vez que se invoca un comportamiento en el modelo de ejecución se produce una ocurrencia de evento en el modelo de traza y esta ocurrencia se representa con la metaclase `OcurrenceSpecification`.

En la Figura 3.24 se recoge la relación entre interacciones y ocurrencias de eventos en el metamodelo de UML. Una interacción está compuesta por, al menos, una línea de vida (`Lifeline`). Una línea de vida representa participante individual en la interacción. La línea de vida describe al clasificador al cual está asociada la interacción que la contiene. También puede hacer referencia a elementos de la estructura interna del clasificador a través de una referencia `ConnectableElement` (§ 3.1.4.1.6). La unidad semántica básica de las interacciones viene representada por objetos `OcurrenceSpecification`. La secuencia de ocurrencias que especifican estos modelan el significado de las interacciones. Estos objetos mantienen una referencia a la especificación de la ocurrencia de un evento (`Event`).

Las peticiones (*request*) que se producen en el modelo de ejecución cuando se invoca una característica de comportamiento UML, se modela en el modelo de interacciones mediante mensajes (`Message`). Con respecto a las ejecuciones de comportamiento en el modelo de ejecución, en el modelo de trazas se corresponden con dos ocurrencias: una al comienzo del comportamiento y otra al final.

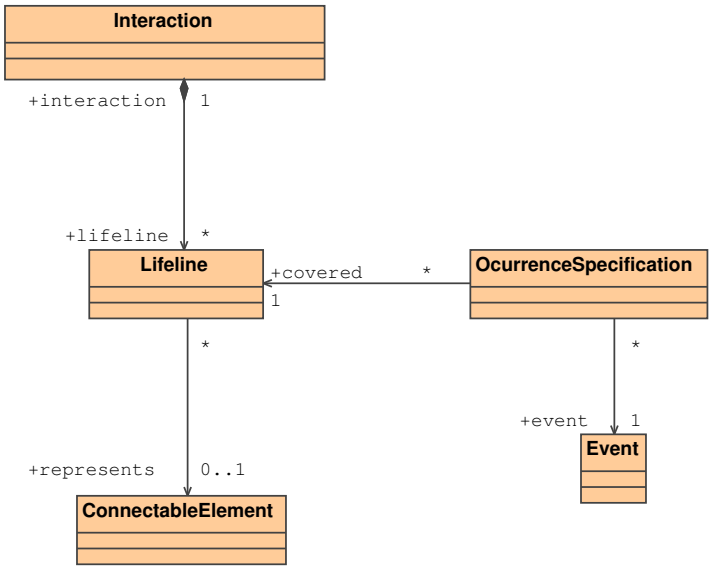

Figura 3.24: Relación entre interacciones, líneas de vida y ocurrencias de eventos en el metamodelo de UML

En la Figura 3.25 se recoge la especificación de los mensajes de las interacciones en el metamodelo de UML. Una interacción UML contiene mensajes (`Message`). Un mensaje define un tipo específico de comunicación entre líneas de vida dentro de una interacción. Una comunicación puede ser, por ejemplo, la invocación de una operación o la creación o destrucción de una instancia. Un mensaje puede hacer referencia un conjunto de valores de especificación (§ 3.1.4.1.7) que representan el valor de los argumentos que se pasan al invocar el mensaje.

Para modelar lo que sucede al principio y al final de la invocación de un mensaje se establecen dos asociaciones con la metaclase `MessageEnd`. `MessageEnd` especifica lo que sucede en uno de los extremos de un mensaje. La subclase `MessageOcurrence Specification` especifica la ocurrencia de eventos tales como el envío o recepción de señales asíncronas (`SendSignalEvent`) o la invocación o recepción de operaciones síncronas (`SendOperationEvent`).

Normalmente, en una interacción no se representan las acciones que produjeron la invocación. En caso de desearse tal nivel de detalle, puede asociarse un comportamiento UML a un objeto `OcurrenceSpecification`. Para ello, se define una subclase `ExecutionOccurrence Specification` que representa un momento en el tiempo en el cual comienza o finaliza la ejecución de acciones o comportamientos UML. La especificación de la ejecución la realiza la metaclase `ExecutionSpecification` cuya duración se representa con dos `ExecutionOccurrence Specification`, la inicial y la final. Para la ejecución de acciones y de comportamientos se definen dos metaclases hijas: `ActionExecutionSpecification` y `BehaviorExecution Specification`. En la Figura 3.26 se representa este diseño en el metamodelo.

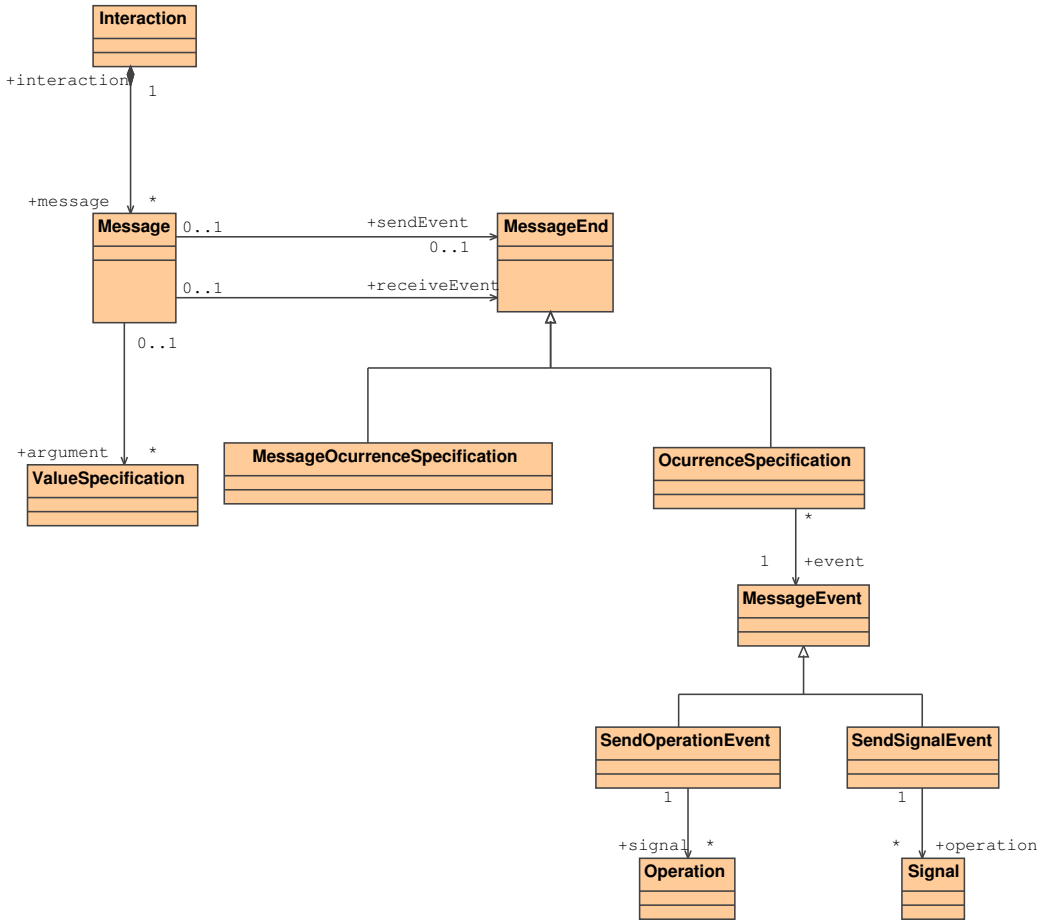

Figura 3.25: Diseño del modelo de mensajes en las interacciones en el metamodelo de UML

Para aumentar el poder expresivo de los diagramas de interacción, UML 2 definió un nuevo tipo de interacción que permitía definir expresiones combinando otras interacciones [OMG, 2004b]. Formalmente, se definió una nueva metaclase hija de `InteractionFragment` denominada `CombinedFragment` (Figura 3.27). Un fragmento de interacción combinado queda definido por un operador de interacción (InteractionOperator) y un conjunto de operandos de interacción (`InteractionOperand`).

Un fragmento combinado puede tener varios operandos y, al mismo tiempo, un operando puede contener varios fragmentos de interacción. `InteractionOperator` es una enumeración de los diferentes tipos de operadores que puede utilizar un fragmento combinado. Ejemplos de tipos de operadores serían `neg` para designar que una traza es invalida o `loop` para construir bucles. Los posibles parámetros que puede necesitar el operador, por ejemplo, el valor máximo y mínimo de iteraciones de un bucle, se recogen en una restricción especial (`InteractionConstraint`)

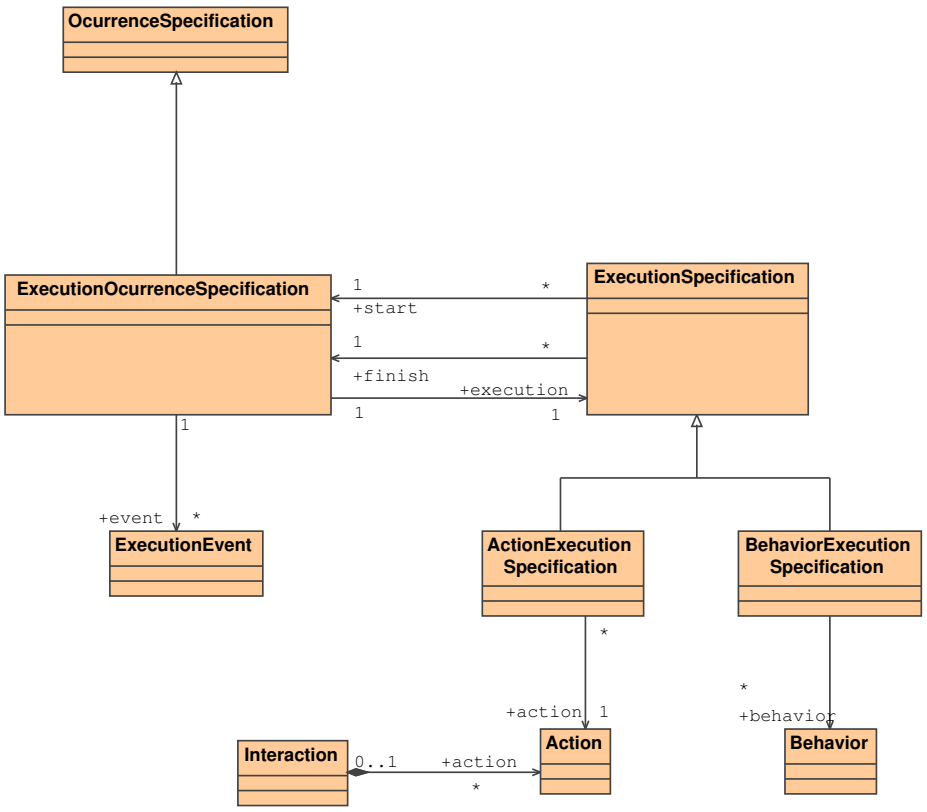

Figura 3.26: Ejecución de acciones o comportamientos UML en el modelo de interacción

asociada a un operando.

Debe señalarse que el objetivo de los fragmentos combinados es aumentar la riqueza expresiva de las interacciones, no ofrecer construcciones que encajen con artefactos habituales de programación. En el ejemplo de los bucles, al especificar su semántica la especificación [OMG, 2004b], se limita a señalar que el bucle iterará como mínimo el valor mínimo especificado en el atributo `maxint` del objeto `InteractionConstraint` y como máximo el valor especificado por `maxint`. No se trata por tanto de estructuras equivalentes a las contenidas en el nivel estructurado del modelo de actividades (§ 3.1.4.2.2).

Los diagramas de interacción UML representan distintas notaciones del metamodelo presentado. Los más utilizados son los diagramas de secuencia [Fowler, 2003d], los cuales hacen uso de la totalidad del metamodelo de interacciones de UML. Los diagramas de comunicación son diagramas de secuencia simplificados. En lugar en una disposición geométrica vertical para representar el orden temporal de los mensajes, permiten numerar la secuencia de mensajes. Por otra parte, no permiten utilizar los mecanismos de estructura de los diagramas de interacción, como los fragmentos combinados.

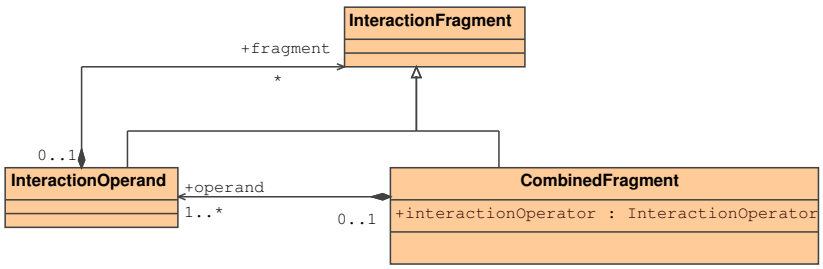

Figura 3.27: Expresiones combinadas en el metamodelo de UML

Los diagramas de secuencia y de colaboración UML son utilizados intensivamente en los métodos de desarrollo orientados a objetos actuales. Su objetivo es describir el comportamiento que se produce cuando varios objetos interactúan, no siendo adecuados para definir con precisión el comportamiento de un objeto [Fowler, 2003d]. En UP se utilizan diagramas de secuencia para describir como interactúan los objetos del sistema para resolver los escenarios de los casos de uso durante el análisis [Jacobson et al., 1999]. Son una técnica utilizada tanto en el análisis, para descubrir qué objetos hacen falta para satisfacer las operaciones, como en el diseño, para especificar los detalles de las interacciones. En este sentido, en [Larman, 2004] se recomienda crear para cada escenario principal de cada caso de uso un "diagrama de secuencia de sistema" que contendrá únicamente las interacciones entre los actores y un objeto `Sistema` y las interacciones del sistema sobre sí mismo.

Analizando las operaciones que permiten realizar las herramientas UML con los diagramas de secuencia, MagicDraw 11 [Magic, 2006] no permite generar código a partir de ellos. Esta herramienta sí permite generar diagramas de secuencia mediante ingeniería inversa a partir del código fuente, utilizando como punto de partida una operación de una clase dada. Dado el código Java del Listado 3.3, en la Figura 3.28 se recoge el diagrama de secuencia generado por MagicDraw a partir de la operación `mensajeA()` de la clase `A`. La notación utilizada para representar el bucle `for` no es UML estándar. Utiliza un perfil propio de MagicDraw por el que extiende la metaclase `Message` para representar sentencias Java.

Listado 3.3: Código Java para generar el diagrama de secuencia

```
pulic class A{
  B b;
  public void mensajeA( ){
    for (int i=1; i<10; i++){
      b.mensajeB();
    }
  }
}

public class B{
  public void mensajeB( ){}
```

```
    }
```

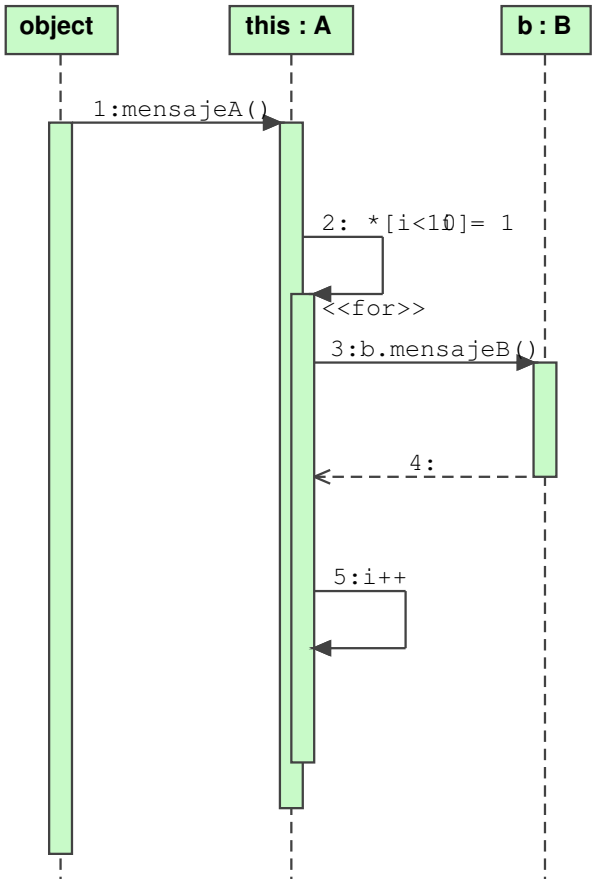

Figura 3.28: Diagrama de secuencia generado por MagicDraw mediante ingeniería inversa

La herramienta Borland Together Architect 2006 [Borland, 2006] ofrece de forma similar a MagicDraw la generación de diagramas de secuencia mediante mecanismos de ingeniería inversa. Al igual que MagicDraw, utiliza extensiones propias en la notación para representar elementos del código fuente, tales como bucles y sentencias condicionales. Ofrece además la posibilidad de generar código fuente a partir de diagramas de secuencia. Ofrece además mecanismos de generación de código a partir de los diagramas de secuencia. Estos mecanismos se limitan a generar el código fuente relativo a las invocaciones de operaciones sobre los objetos.

Al igual que los diagramas de actividad (§ 3.1.4.2.2) y los diagramas de máquinas de estados (§ 3.1.4.2.3), se han utilizado diagrama de interacción para modelado formal de comportamiento en dominios específicos. En [Bauer, 1999, Odell et al., 2000] se propone una extensión de UML para modelar comportamiento de agentes. En esta extensión, denominada *Agent UML*, se propone utilizar diagramas de secuencia UML para modelar protocolos de interacción multiagente. En [Abdurazik and Offutt, 2000]

se propone utilizar diagramas de colaboración para la generación automática de test de integración, analizando cómo los diferentes objetos se comunican unos con otros en los diagramas de colaboración.

Un aspecto a destacar, desde la perspectiva de una herramienta MDA, es que el fin de los diagramas de interacción no es detallar el comportamiento de los objetos, sino representar escenarios de colaboración de objetos. Aunque esta representación puede hacerse con gran nivel de detalle, en su forma estándar no contienen suficiente información como para generar el código fuente de los métodos de las clases por completo. En este sentido, en [Engels et al., 1999] se proponen unas directrices a la hora de utilizar diagramas de colaboración para modelar comportamiento y se define un algoritmo de transformación para generar código Java. Mediante este algoritmo se genera una parte sustancial del comportamiento de las clases haciendo el proceso de transformación más predecible, pero no automático.

En UML ejecutable (§ 3.2) no se utilizan diagramas de interacción para construir modelos transformables en código. Se utilizan los diagramas de comunicación como un mecanismo para visualizar los patrones de comunicación entre clases y los de secuencia para clarificar la ejecución de escenarios sencillos [Mellor and Balcer, 2002]. En [Raistrick et al., 2004] se mantiene este planteamiento recomendando, como paso previo al modelado detallado del comportamiento, la creación de diagramas de comunicación para cada dominio[17]. En [Raistrick et al., 2004] se propone también obtener diagramas de secuencia automáticamente mediante una traza de la ejecución de los modelos. Estos diagramas podrían ser utilizados para validar los modelos inicialmente construidos mediante una comparación manual.

3.1.4.2.5. Diagrama de Casos de Uso

Un caso de uso es una descripción del comportamiento de una parte de un sistema en términos de la interacción que se produce entre dicha parte y actores externos al sistema [Cockburn, 2000].

Los casos de uso no se utilizan para descubrir y modelar los requisitos funcionales del sistema. Su creador fue Ivar Jacobson, el cuál los incorporó a UML [Booch et al., 1999] y al Proceso Unificado [Rumbaugh et al., 1998], proceso de desarrollo que se define como "dirigido por casos de uso".

Es importante señalar que, los diagramas de casos de uso UML únicamente representan una perspectiva de las relaciones entre los casos de uso y los actores, pero no su contenido. Como señala Craig Larman en [Larman, 2004] la tarea de modelar casos de uso trata sobre escribir texto, no sobre dibujar diagramas.

En el metamodelo de UML un caso de uso (`UseCase`) es un `BehavioredClassifier` . Esto significa que pueden asociarse formalmente comportamientos UML, tales como interacciones, a un caso de uso para detallar cómo colaboran los objetos del sistema para resolverlo.

[17]En esta obra se denominan CCD (*Class Collaboration Diagram*) porque las líneas de vida se limitan a clases UML. Los diagramas de comunicación de UML 2.0 se denominaban "de colaboración" en versiones anteriores.

Los casos de uso también pueden definirse como un conjunto de escenarios [Larman, 2004]. Cada escenario describe un uso concreto de un aspecto determinado del sistema. Los casos de uso se describen con lenguaje natural, por lo tanto no pueden ser la entrada de una herramienta MDA que especifique un programa. El mismo razonamiento es aplicable a las "historias de usuario" (*user stories*) propuestas en XP [Beck, 1999].

No obstante los diagramas de casos de uso pueden ser utilizados en determinados contextos para generar código fuente, o esqueletos de código fuente. En AndroMDA se utilizan los casos de uso contenidos en un diagrama UML de casos de uso para dividir los archivos generados para cada diagrama de actividad que los explote [AndroMDA, 2006]. También para modelar los perfiles de autorización a los servicios, utilizando para ello actores UML y creando dependencias entre ellos y los métodos que representan los servicios.

3.1.5. Aumento de la Semántica de Operación mediante Restricciones Explícitas

La semántica del lenguaje UML define implícitamente diversas restricciones. Por ejemplo, al utilizar una asociación de composición se establece que la eliminación del objeto compuesto debe implicar la eliminación de los objetos que lo componen [OMG, 2004b]. Otro ejemplo son las cardinalidades de las asociaciones, que establecen el número de ocurrencias máximas y mínimas de cada instancia participante en las mismas.

Sin embargo existen circunstancias en las se necesita establecer explícitamente restricciones sobre el modelo, y la semántica de las construcciones UML se muestra insuficiente. Para ello, UML define en su metamodelo el concepto de restricción (`Constraint`). Una restricción en UML se define una condición acerca de la semántica de un elemento UML (`Elemento`). `Element` es la metaclase padre de todos los elementos del metamodelo de UML, luego pueden establecerse restricciones sobre cualquier elemento del lenguaje. La restricción puede especificarse mediante un valor de especificación UML (`SpecificationValue`). Como se vio en § 3.1.4.1.7, en UML pueden utilizarse diferentes tipos de expresiones para representar valores de especificación: expresiones en lenguaje natural o en algún lenguaje de programación. En la Figura 3.29 se muestra este diseño en el metamodelo de UML.

Para entender qué aportan las restricciones a los modelos construidos con UML conviene revisar lo que se conoce como Diseño por Contratos (*Design by Contract*). Es una metodología de diseño de software basada en especificar explícitamente los contratos de uso de los módulos software de una manera formal, más allá de la signatura de las operaciones que soportan [Meyer, 2000]. Para realizar esta especificación, se proponen dos tipos de restricciones formales:

Precondiciones y postcondiciones. Una precondición establece una afirmación (*assertion*) que debe satisfacerse en el momento en el que una operación comienza su ejecución. Una postcondición establece una restricción que debe

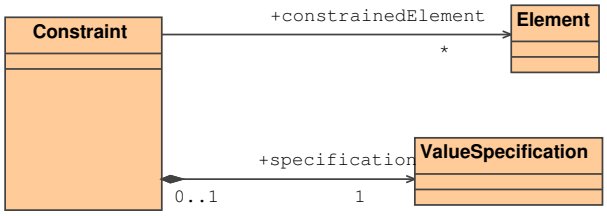

Figura 3.29: Restricciones en el metamodelo de UML

satisfacerse cuando la operación finaliza su ejecución.

Invariantes. Una invariante define una afirmación acerca del estado de un sistema o módulo que siempre debe ser verdadera.

El objetivo en última instancia del diseño por contratos es aumentar la corrección del software. Sin importar como se implementen los módulos software, si se especifican las condiciones que éstos deben satisfacer y las satisfacen, entonces es de esperar un funcionamiento correcto del sistema global.

El lenguaje de programación que soporta diseño por contratos más conocidos es Eiffel [Meyer, 1991]. Aunque los lenguajes más utilizados en la actualidad no soportan la programación por contratos, existen librerías y productos de terceros que permiten su implementación[18].

UML ofrece soporte directo para el diseño por contratos. Una operación[19] UML puede presentar precondiciones y postcondiciones, modeladas con dos asociaciones entre las metaclase `Operation` y `Constraint`. Para definir invariantes pueden establecerse restricciones sobre las clases o sobre propiedades de las mismas.

Una herramienta MDA podría generar el código relativo a las restricciones establecidas sobre los modelos automáticamente. Para ello, el lenguaje utilizado para definir restricciones debe ser un lenguaje formal procesable por una máquina. El lenguaje de especificación de restricciones predefinido en UML es OCL (§ 3.1.5.1), aunque se considera válido utilizar cualquier lenguaje de programación existente o diseñado al efecto. La especificación de UML señala al respecto que la sintaxis e interpretación del lenguaje utilizado es responsabilidad de las herramientas.

3.1.5.1. Object Constraint Language (OCL)

Object Constraint Language (OCL) es un lenguaje de especificación para definir restricciones sobre modelos UML [OMG, 2005e]. Las raíces de OCL radican en el lenguaje de modelado Syntropy [Cook and Daniels, 1994] y al trabajo realizado

[18]Habitualmente en forma de preprocesadores. Aunque la programación por contratos es un ejemplo habitual de aplicación de la programación orientada a aspectos. Existe una herramienta para Java que utiliza este enfoque: [Contract4J, 2006]

[19]Nótese que una operación UML sólo define una signatura. Su implementación vendrá dado por un comportamiento UML (método) tal y como se vio en § 3.1.4.2.1.

por Steve Cook y Jos Warmer en 1995 en un proyecto de IBM. OCL fue diseñado para ser al mismo tiempo formal y simple. Buscaba evitar la complejidad de los lenguajes formales, que tenían un fuerte contenido matemático y, a la vez, evitar la ambigüedad del lenguaje natural [Kleppe et al., 2003]. OCL fue utilizado desde las primeras revisiones de UML para dotar de precisión a la definición del lenguaje, inicialmente descrito únicamente textualmente. Como consecuencia, se incluyó también OCL en el estándar UML, para que los usuarios del lenguaje pudieran utilizarlos con los modelos que construyesen.

OCL es un lenguaje de especificación puro. Esto significa que una expresión OCL no puede tener efectos laterales. Las expresiones OCL se limitan a devolver valores. El estado de un sistema nunca podrá ser modificado por la evaluación de una expresión OCL, lo cual no significa que no puedan utilizarse expresiones OCL para expresar cambios de estado [OMG, 2005e] (por ejemplo, una postcondición). OCL no es un lenguaje de programación. No puede utilizarse para programar la lógica de control de flujo de los programas. Tampoco pueden invocarse procedimientos u operaciones que no sean de consulta.

OCL, en su versión actual 2.0 [OMG, 2005e], queda caracterizado por tres aspectos fundamentales [Warmer and Kleppe, 2003]:

Lenguaje de consulta y de restricciones. Una de las novedades de UML 2 es que puede utilizarse OCL no sólo para especificar restricciones, sino también cualquier tipo de expresión en los elementos del lenguaje. Toda expresión OCL indica un valor dentro del sistema. Por ejemplo, la expresión $2 + 3$ es una expresión OCL válida que representa el valor `Integer` UML 4.

Ejemplos donde pueden utilizarse expresiones OCL son la especificación de cómo se deriva el valor de un determinado atributo o la especificación de los valores iniciales de los atributos de las clases.

Los valores expresados con OCL pueden ser colecciones de valores, por ejemplo, colecciones de referencias a objetos. En [Akehurst and Bordbar, 2001] se demuestra que OCL tiene las mismas capacidades que SQL [Chamberlin and Boyce, 1974]. Esto significa que el cuerpo de una operación de consulta puede ser completamente especificado por una expresión OCL.

Lenguaje fuertemente tipado. Toda expresión OCL tiene un tipo. Para que una expresión del lenguaje esté bien formada, ésta debe estar de acuerdo con las reglas de concordancia de tipos del lenguaje [OMG, 2005e]. Como consecuencia pueden validarse las expresiones OCL en tiempo de modelado, sin tener que obtener una versión ejecutable del sistema [Warmer and Kleppe, 2003].

Lenguaje declarativo. OCL es un lenguaje declarativo. A diferencia de los lenguajes procedurales, donde las expresiones son descripciones de las acciones que deben realizarse, en un lenguaje declarativo una expresión establece *qué* debería realizarse, pero no *cómo*. Esta es la razón por la que las expresiones OCL no tienen efectos laterales, esto es, no cambian el estado del sistema.

A la hora de especificar la versión 2.0 de OCL [OMG, 2005e] se ha seguido un enfoque similar al visto para UML (§ 3.1.3). Se especifica una sintaxis abstracta del lenguaje dada por el metamodelo del mismo definido utilizando *Meta Object Facility* (MOF) 2.0 (§ 3.3). También se proporciona una sintaxis concreta del lenguaje utilizando una gramática atribuida [Ortín Soler et al., 2004] expresada mediante notación EBNF [Cueva Lovelle, 1998].

Toda expresión OCL está relacionada con una instancia de un tipo UML específico [OMG, 2005e]. Ésta definirá el contexto de la expresión OCL. La palabra reservada `self` sirve para hacer referencia esta instancia contextual. Por ejemplo, en el Listado 3.4 se define una invariante OCL asociada a todas las instancias de la clase `Empresa`: el valor del atributo `numeroDeEmpleados` debe ser siempre mayor que 0.

Listado 3.4: Ejemplo de definición de una invariante OCL

```
context Empresa inv:
  self.numeroDeEmpleados > 0
```

OCL también ofrece soporte explícito para definir precondiciones y postcondiciones en operaciones UML. En el Listado 3.5 se muestra un ejemplo. Para invocar la operación `estaVacia()` que indica si un objeto cuenta de la clase `Cuenta` está vacía, debe verificarse que la cuenta esté habilitada. Además, después de invocar la operación, debe verificarse que el resultado de la misma, referenciado con la palabra reservada `result`, coincida con la comprobación acerca de si existe dinero en la cuenta.

Listado 3.5: Ejemplo de definición de precondiciones y postcondiciones de operaciones con OCL

```
context Cuenta::estaVacia(): Boolean
  pre: estaHabilitada()
  post: result = (dinero = 0)
```

OCL permite definir los valores iniciales de atributos y las expresiones que indican como se calcula el valor de un atributo derivado. En el Listado 3.1.5.1 se muestra un ejemplo de ambas utilizaciones. En el contexto del atributo `ingresos` de la clase `Persona`, se define que su valor inicial es 0. Y su valor derivado depende de si la persona es menor de edad o no. Si lo es, entonces el valor es el 1 % del salario de sus padres. Si no lo es, entonces sus ingresos conciciden con su salario.

```
context Person::ingresos : Integer
init: 0
derive:
  if menorEdad
    then parents.income->sum() * 0.01
    else trabajo.salario
  endif
```

Como se ha señalado, OCL puede utilizarse para definir el cuerpo de las operaciones de consulta. Una operación de consulta es una operación que no cambia el estado del sistema, limitándose únicamente a devolver un valor o conjunto de

valores [Warmer and Kleppe, 2003]. El tipo de la expresión utilizada debe coincidir con el tipo de retorno de la operación. En el Listado 3.6 se muestra un ejemplo. Se muestra una expresión OCL en el contexto de la operación `getEsposaActual()` de la clase `Persona`. Esto código especifica qué debe devolver la operación: de todos los matrimonios en los haya participado la persona, se selecciona aquél que no haya finalizado, y se selecciona la esposa de dicho matrimonio. El objeto seleccionado será del tipo `Persona` coincidiendo con el tipo de retorno de la operación. Nótese que OCL permite utilizar construcciones `select` similares a las permitidas en SQL [Chamberlin and Boyce, 1974].

Listado 3.6: Ejemplo de definición del cuerpo de operaciones de consulta con OCL

```
context Persona::getEsposaActual() : Person
  pre: self.estaCasado = true
  body: self.matrimonios->select( matrimonio | matrimonio.
      ended = false ).esposa
```

OCL se utiliza dentro de la especificación de UML cuando es posible especificar formalmente lo que se describe textualmente. Por ejemplo, en la especificación de UML 2, se establece que, en una asociación tipo composición, el límite superior de la cardinalidad en el extremo que agrega no puede ser mayor que 1. En el Listado 3.7 se muestra el código OCL correspondiente a esta especificación.

Listado 3.7: Especificación de una restricción sobre el metamodelo de UML con OCL

```
--A multiplicity on an aggregate end of a composite
    aggregation must not have an upper bound greater than 1.

isComposite implies
  (upperBound()->isEmpty() or upperBound() <= 1)
```

En OCL 2 se ha definido un metamodelo del lenguaje que define su sintaxis abstracta. Además, este metamodelo se alinea con el metamodelo de UML 2 al estar definido utilizando MOF. De este modo se ha solucionado uno de los problemas señalados por David Frankel en [Frankel, 2003]: la dificultad para las herramientas UML de construir analizadores OCL que permitan manejar sus construcciones de una manera estándar. El diagrama de clases del metamodelo de la sintaxis abstracta de OCL se recoge en la Figura 3.30. También permite definir diferentes notaciones para expresar restricciones que utilicen el mismo metamodelo, dando respuesta a la propuesta realizada en [Mellor and Balcer, 2002] relativa a poder expresar restricciones utilizando lenguajes de acción UML que implementen *Action Semantics*.

No existen en el momento de escribir estas líneas muchas herramientas UML que soporten OCL de una manera formal. Una de las herramientas que sí lo soportan es Borland Together Architect 2006 [Borland, 2006]. Además de ofrecer facilidades para la edición de expresiones OCL, permite la generación automática de código Java a partir de ellas. En la Figura 3.31 se recoge un ejemplo con el fin de mostrar el código generado por la herramienta. Sobre una clase `Persona` que tiene un atributo `edad` de tipo entero, con un método `getEdad()` y un método `nace()` se definen 2

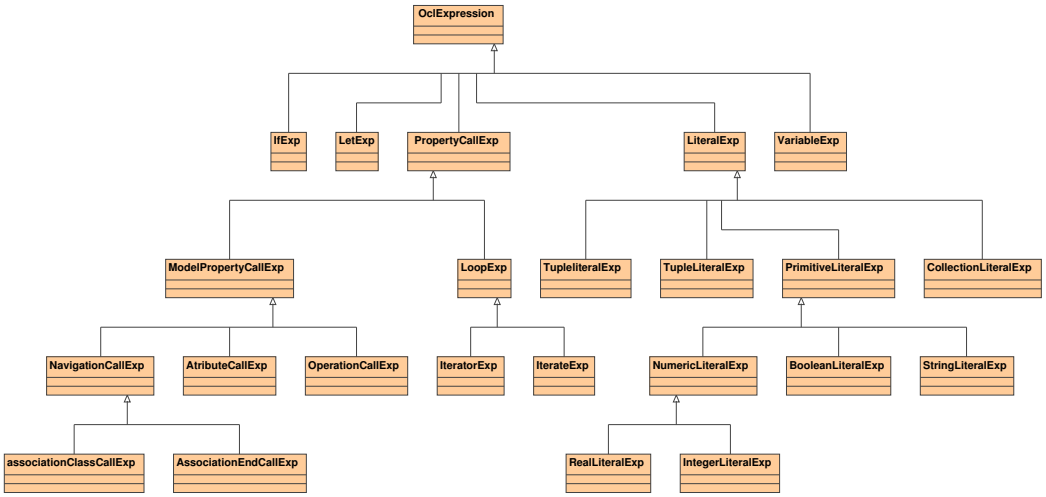

Figura 3.30: Sintaxis abstracta de OCL

restricciones: una invariante, en el contexto[20] de la clase, que indica que el atributo `edad` siempre debe ser mayor o igual que 0; y una postcondición sobre la operación `nace()`, que indica que tras ejecutarse dicha operación el valor del atributo edad debe ser 0. Además, se utiliza OCL para representar el cuerpo de la operación de consulta `getEdad()`, que se limita a devolver el valor del atributo `edad`.

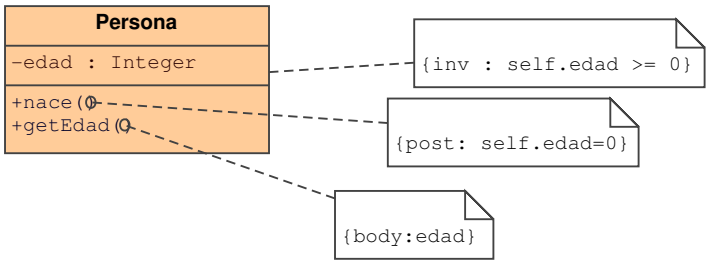

Figura 3.31: Dos restricciones y una operación de consulta definidas sobre una clase UML utilizando OCL

En el Listado 3.8 se recoge el código generado por la herramienta Borland Together Architect a partir de las restricciones mostradas en el diagrama de la Figura 3.31. Dentro de la clase afectada por las restricciones, se crea una clase interna que define las operaciones relativa a la postcondición y a la invariante, además de un método de conveniencia que validaría todas las invariantes si existiesen más de una. El tipo de retorno de todas estas operaciones Java es booleano. Para ejecutar las

[20]En la notación gráfica de UML el contenido de las restricciones se representa entre llaves. Se acepta utilizar los símbolos de comentarios UML para representar las restricciones y los enlaces de comentarios permiten unir las restricciones con los elementos UML que representan su contexto.

comprobaciones, utiliza el sistema de asertos introducido en la versión 1.4 de Java [Sun, 2003]. En cada método genera el código que comprueba si se verifican las invariantes de la clase. En el método `nace()` comprueba, además la postcondición especificada para él. También genera el código que implementa la consulta en la operación `getEdad()`.

Listado 3.8: Código Java generado a partir de las restricciones OCL

```
/**
 * @model.uin <code>design:node:::-tdsqjuep9wc4ql-bbh861</
    code>
 */
public class Persona {
  public int edad;

  public void nace() {
    /* default generated stub */;
    assert (OCL.postNace(this) && OCL.allInvariants(this));

  }

  public int getEdad() {
    pruebaOCL.Persona self = this;
    int tmpOclRet = self.edad;
    assert (OCL.allInvariants(this));
    return tmpOclRet;
  }

  /** OCl generated class DO NOT MODIFY*/
  protected static class OCL {
    private static final boolean isEnabled = Persona.class
        .desiredAssertionStatus();

    private OCL() {
    }

    static protected boolean postNace(final Persona self) {
      java.lang.Boolean bool1 = java.lang.Boolean.valueOf(self.
          edad == 0);
      return bool1.booleanValue();
    }

    static boolean inv$0(Persona self) {
      java.lang.Boolean bool2 = java.lang.Boolean.valueOf(self.
          edad >= 0);

      return (bool2 != null ? (bool2).booleanValue() : false);
    }

    static boolean allInvariants(Persona self) {
      return inv$0(self);
    }
  }
```

```
    }
```

Existe un consenso en la comunidad MDA acerca de los beneficios de utilizar OCL para enriquecer los modelos UML: permite construir modelos más precisos, con mayor capacidad de comunicación y documentación. Sin embargo, en lo relativo a su papel a la hora de modelar comportamiento formalmente y procesos de generación de código fuente existen diferentes aproximaciones. David Frankel señala en [Frankel, 2003] que la posibilidad de generar automáticamente el código de las restricciones a partir del análisis de expresiones OCL facilitará la adopción por parte de los desarrolladores del diseño por contratos [Meyer, 2000]. También señala que la mayor parte de los sistemas no pueden ser completamente generados a partir de un modelo declarativo de los mismos, y apunta a *Action Semantics* como solución para definir este comportamiento.

Una aproximación distinta se encuentra en [Kleppe et al., 2003], donde se apuesta por la combinación UML y OCL como la mejor aproximación a la hora de especificar los PIM en MDA. En opinión de sus autores, la propuesta de UML Ejecutable (§ 3.2) y *Action Semantics* supone descender a un nivel de abstracción tan bajo que no es aceptable. Por ello proponen utilizar OCL intensivamente, aunque también reconocen que la mayor parte de las veces habrá que especificar el código de las operaciones directamente en los PSM y que todavía no es posible generar los aspectos dinámicos del sistema con una combinación UML y OCL.

Así pues, una pregunta interesante de cara a esta investigación es ¿qué partes de la aplicación final pueden generarse a partir de expresiones OCL? Con respecto a OCL como un lenguaje de especificación de restricciones, UML no especifica cuando o ni siquiera cómo deben verificarse las restricciones. Por ejemplo, pueden ser comprobadas en tiempo de diseño, o durante la fase de transformación del modelo a una plataforma concreta. UML tampoco define qué efecto en tiempo de ejecución debe producirse si la condición de una restricción falla [Bock, 2003b], por ejemplo, si se detiene la ejecución del sistema. Como con el resto de elementos del lenguaje, UML no dice nada acerca de cómo deben implementarse en un lenguaje de programación. En la práctica suelen utilizarse asertos, pero no es obligatorio.

OCL es un lenguaje declarativo que no indica *cómo* deben realizarse las operaciones, sólo *qué* restricciones deben verificar o *qué* valores se desean obtener. En determinados casos, esta declaración es suficiente como para generar automáticamente el código que la implemente. En caso contrario, deberán encontrarse otros mecanismos para especificar la funcionalidad que verifique las restricciones.

En [McNeile and Simons, 2004] se indica que la generación de código será posible siempre y cuando coincidan los niveles de abstracción correspondientes a la definición del contrato y a la implementación de la funcionalidad que lo verifique. Sin embargo, señala que precisamente cuando los niveles coinciden es cuando peor funcionan el diseño por contratos, puesto que los contratos especifican condiciones sin importar cómo se implementen estas. En este artículo también se indica que el uso de contratos para especificar comportamiento supone fragmentar la descripción del comportamiento de los objetos haciendo el sistema más difícil de entender.

Lo anterior no significa que las precondiciones, postcondiciones e invariantes no

sean útiles a la hora de modelar. El mejor uso que se puede hacer de ellas es indicado precisamente en el diseño por contratos [Meyer, 2000]: verificar que los modelos se comportan correctamente, especificando con otros mecanismos la funcionalidad que define dicho comportamiento. Otro área donde OCL resulta adecuado, es para definir y utilizar expresiones sobre las construcciones UML de modelado. Por ejemplo, para especificar los valores iniciales de los atributos, o las condiciones que debe verificar un estado para poder activarse en una máquina de estados UML. Con respecto a OCL como lenguaje de especificación de consultas, aunque presenta la potencia para especificar cualquier consulta que pueda ser resuelta navegando por el metamodelo UML y realizando comparaciones, puede no resultar un mecanismo natural desde el punto de vista del desarrollador. Especialmente si se está utilizando un lenguaje de programación imperativo para especificar el cuerpo de los métodos, tales como los lenguajes de acción basados en *Action Semantics*.

3.1.6. Mecanismos de Extensión de UML

A partir de la versión 1.4 de UML se incluyó en el estándar el concepto de "perfil" como mecanismo general de especialización. Un perfil UML define un modo específico de utilizar UML [Kleppe et al., 2003]. Permite adaptar el metamodelo de UML con construcciones que son específicas a un dominio, plataforma o metodología particular [Mellor et al., 2004].

Un perfil comprende una combinación de tres mecanismos: estereotipos, restricciones y valores etiquetados (*tagged values*).

Estereotipos. Un estereotipo define cómo puede extenderse una metaclase existente y, además, permite utilizar una notación o terminología específica a un dominio o plataforma en lugar de, o además de, las que sean utilizadas para la metaclase que es extendido [OMG, 2004b]. Básicamente, un estereotipo se adjunta a un elemento de modelado para indicar que dicho elemento es de algún modo diferente del resto. Por ejemplo, el estereotipo `<<metaclass>>` adjuntado a una clase UML especifica que dicha clase es una metaclase.

Restricciones. Pueden adjuntarse restricciones[21] a la definición de un estereotipo. Describirán las restricciones que se aplican a los elementos de modelado sobre los que se apliquen el estereotipo. Por ejemplo, el estereotipo `<<JavaClass>>`, definido en el perfil EJB definido en *Java Community Process* (JCP) [Sun, 2001], especifica la restricción de que la clase sobre la que se aplique únicamente puede tener una superclase (restricción impuesta por la arquitectura del lenguaje Java [Gosling et al., 1996]).

Valores etiquetados. Los valores etiquetados permiten añadir meta-atributos a una metaclase del metamodelo de UML [Kleppe et al., 2003]. Un valor etiquetado tiene un nombre y un tipo y se adjunta a un estereotipo concreto. Dentro de

[21] El concepto de restricción y la manera de expresar restricciones se revisa en § 3.1.5

los modelos que instancien el metamodelo, se les puede asignar un valor, pero siempre que el respectivo elemento del metamodelo tenga el estereotipo asociado.

La sintaxis abstracta del sistema de perfiles de UML se define en la especificación UML *Infrastructure* [OMG, 2005k], donde junto con el paquete *Core*, conforman la biblioteca *Infrastructure*. En concreto, el paquete *Profile* depende del paquete *Constructs* contenido en el paquete *Core* (§ 3.3.3). La razón es que el sistema de perfiles es compatible con MOF, permitiendo extender metaclases de metamodelos definidos utilizando MOF. El metamodelo de UML es por lo tanto un escenario particular de extensión, aunque sin duda el más utilizado.

En la Figura 3.32 se recogen los aspectos básicos del metamodelo de perfiles de UML. Un perfil (`Profile`) es un paquete UML (`Package`) que agrega estereotipos (`Stereotype`). Un estereotipo es una clase (`Class`) especializada. Los valores etiquetados se representarán como atributos de esta clase especializada.

La metaclase `Extension` se utiliza para indicar que ciertas propiedades de una metaclase se extienden mediante un prototipo. Esta metaclase especializa una asociación (`Association`). Un extremo de la extensión es una propiedad UML ordinaria (`Property`) de la clase a la que se adjunta el estereotipo; el otro extremo es un objeto (`ExtensionEnd`) que hace referencia al estereotipo que extiende la clase.

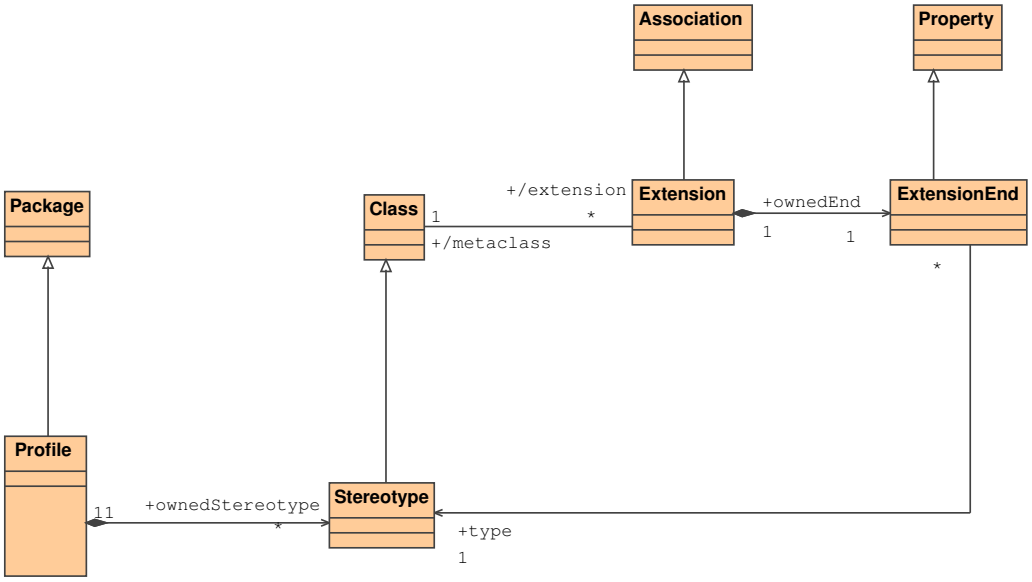

Figura 3.32: Metamodelo del sistema de Perfiles UML

Se han definido diversos perfiles dentro del OMG para diversa tecnologías y dominios: CORBA [OMG, 2002c]; CORBA Component Model (CCM) [OMG, 2005h]; *Enterprise Application Integration* (EAI) [OMG, 2004a]; *Enterprise Distributed Object Computing* (EDOC) [OMG, 2004b]; calidad de servicio y tolerancia a fallos [OMG, 2006]; para pruebas (*testing*) [OMG, 2005i].

3.1.7. Aportaciones y Carencias para la Investigación

UML es el estándar del OMG más extendido y con mayor soporte en el momento actual. Está destinado a jugar un papel importante en la iniciativa MDA, sin embargo, existe un debate en torno a la utilización de UML en el desarrollo dirigido por modelos donde surgen dos preguntas recurrentes:

1. ¿Para qué debe utilizarse UML? UML se define como un lenguaje de modelado de propósito general: ¿signfica esto que deben modelarse todos los aspectos del sistema utilizando UML?

2. ¿Cómo debe utilizarse UML? UML es un lenguaje de gran tamaño que ofrece diferentes alternativas para modelar diferentes aspectos. En concreto, para modelado de comportamiento permite diferentes aproximaciones con características particulares cada una de ellas. ¿Qué partes del lenguaje deben utilizarse y cómo para realizar un modelado formal?

Con respecto a la primera cuestión, se han realizado fuertes críticas a la aspiración inicial del OMG de hacer de UML un lenguaje para modelar casi todos los aspectos de un sistema. Con este fin, se han realizado drásticas modificaciones a la arquitectura interna del lenguaje, que ha ido viendo incrementarse el número de conceptos que permite manejar. Como se vio en § 3.1.4, estos conceptos van en la actualidad desde el concepto de "clase" hasta el concepto de "bucle" o de "acción de modificar un atributo".

Gracias a la separación realizada de la sintaxis abstracta del lenguaje, su representación no se restringe a una notación gráfica[22], pero ésto no evita el hecho de que UML se haya convertido en un lenguaje muy amplio y por lo tanto más difícil de entender y manejar. Martin Fowler critica en [Fowler, 2003d] que la evolución reciente de UML ha ignorado a los usuarios que utilizan UML en su versión "como boceto", la mayoría en la actualidad, y en un artículo titulado "The unwanted modelling language" [Fowler, 2003f] argumenta que este hecho está alejando cada vez más a los usuarios de UML del estándar.

Existen dominios a los que UML se adapta bien utilizando los mecanismos de extensión predefinidos. Un ejemplo es el modelado de datos, como se vio en § 3.1.4.1.1. Sin embargo existen otros dominios para los que UML simplemente no encaja. Un ejemplo típico es la especificación de interfaces gráficas de usuario, que no puede realizarse con UML estándar [Raistrick et al., 2004]. Se han definido perfiles de UML para esta labor [Hennicker and Koch, 2001, Blankenhorn and Jeckle, 2004] pero su uso no ha salido del ámbito académico.

En [Thomas, 2003] en un artículo titulado *"Unified or Universal Modeling Language?"* Dave Thomas cuestiona si el enorme número de abstracciones que recoge

[22]Lo que hace difícil de entender porqué Martin Fowler, un experto conocedor de UML y su especificación, argumenta en contra de UML que una notación gráfica no tiene que ser más productiva que una textual [Fowler, 2003b].

UML no es únicamente un mero ejercicio académico sin utilidad real. En particular señala que diferentes problemas requieren diferentes abstracciones las cuales necesitan diferentes lenguajes.

Lo que se está definiendo en el artículo anterior es la necesidad de lenguajes DSL, lenguajes específicos de dominio [Fowler, 2005b]: un enfoque distinto a utilizar UML para especificar todos los aspectos del sistema es diseñar lenguajes específicos al dominio de cada aspecto a especificar. En la actualidad, existe un consenso acerca de la necesidad de utilizar *Domain Specific Language* (DSL) en MDA [Booch et al., 2004], la cuestión es *cómo* se especifican los DSL. En § 3.3.6 se analizará la utilidad de MOF para definir DSL.

En cuanto a la utilización de UML para especificar comportamiento se ha observado como existen diferentes aproximaciones. El enfoque de utilizar OCL como único mecanismo junto con UML estructural para especificar los PIM, defendido por algunos autores como Anneke Kleppe [Kleppe et al., 2003], queda descartado para esta investigación. OCL no es suficiente para poder generar el código necesario que implemente la funcionalidad que valide las restricciones (§ 3.1.5). El comportamiento emergente UML, y en particular el modelo de interacciones, también quedan descartados como mecanismos de especificación de comportamiento para una herramienta MDA. Su objetivo es hacer énfasis en cómo se producen las interacciones entre varios objetos en un escenario, no especificar cómo se comporta cada uno de esos objetos.

Los diagramas de estados UML se muestran adecuados para indicar cómo evolucionan las instancias de una clase a lo largo del tiempo y de diferentes escenarios, pero se necesita un mecanismo adicional que permita especificar qué sucede en dichas instancias cada vez que se produce un cambio de estado. Es en este punto donde resulta interesante *Action Semantics* al definir un lenguaje de acción. Aunque esta manera de enfoque de uso de UML (UML Ejecutable) será analizado en el § 3.2, debe destacarse el intento de convertir UML en un verdadero lenguaje de programación, a pesar de que este objetivo estaba explícitamente excluido por sus creadores [Rumbaugh et al., 1998]. Mediante el modelo de acciones UML se especifican los conceptos de ejecución mínimos, tales como crear un objeto o modificar un atributo, y mediante el modelo de actividades UML, y en concreto mediante las actividades estructuradas, se permite coordinar el flujo de ejecución de dichas actividades. En el § 3.2 se discutirán las ventajas y desventajas de este enfoque.

De cara a esta investigación, mucho más importante que la notación UML resulta la especificación de su metamodelo. El disponer de una especificación formal y orientada a objetos de cada concepto de UML permite construir software que puedan manipular los conceptos del lenguaje de forma automatizada. Un enfoque lógico para cualquier herramienta que manipule modelos UML es utilizar su arquitectura basada en una sintaxis abstracta para construir un único repositorio de instancias del metamodelo de UML y, sobre dicho repositorio, edificar tantas "vistas" como se deseen (por ejemplo, una representación gráfica en diagramas UML). Conrad Bock analiza las ventajas de este enfoque en [Bock, 2003c]. Estas ventajas, o necesidades, coinciden con la utilización de una tabla de símbolos en un compi-

lador tradicional [Juan Fuente et al., 2006]. La manipulación de metamodelos será analizada en § 3.3.

3.2. UML Ejecutable

En el Capítulo 3 se analizaron las diferentes aproximaciones que permite UML para modelar comportamiento. Como se vio, en UML 2 existe un modelo de acciones y un modelo de ejecución sobre el que se edifican las construcciones que especifican dinámica de comportamiento. En este capítulo se analiza una aproximación para utilizar UML que persigue un ambicioso objetivo: permitir que los modelos UML puedan ejecutarse.

3.2.1. Introducción

Los orígenes de lo que hoy se conoce como UML ejecutable son anteriores al propio lenguaje. Se remontan a los trabajos realizados por Shlaer y Mellor y su propuesta para construir modelos de análisis orientados a objetos especificados formalmente [Shlaer and Mellor, 1988, Shlaer and Mellor, 1991]. Una característica de sus modelos es que tenían una semántica bien definida. En teoría estos modelos eran potencialmente ejecutables, aunque no existían herramientas que permitiesen ejecutarse.

No fue hasta 1993 cuando IBM definió *Process Specification Language* (PSL) y construyó un prototipo Smalltalk que permitía ejecutar modelos Shlaer-Mellor. Influenciados por este prototipo, en la empresa Kennedy Carter comenzaron a desarrollar la versión inicial de lo que posteriormente se convertiría uno de los primeros lenguajes de acción UML: *Action Specification Language* (ASL).

Con la adopción de UML por parte del OMG en 1997 como lenguaje estándar de modelado Software, Steve Mellor junto con miembros de Kennedy Carter desarrollaron una propuesta para adaptar el proceso Shlaer-Mellor y formalizar su uso con UML [Wilkie and Mellor, 1999]. Se detectó que era necesario aplicar varias restricciones y añadir determinadas características a los modelos UML para hacerlos ejecutables. Esto llevó a formar un consorcio en el seno del OMG denominado *Action Semantics* que trabajó desde 1998 a 2001 para desarrollar la especificación del mismo nombre: *Action Semantics* [OMG, 2001]. Esta especificación fue incluida dentro del estándar UML en su versión 1.5 [OMG, 2003]. Como se vio en § 3.1, en UML 2 se mejoró la integración de *Action Semantics* en el metamodelo del lenguaje.

UML Ejecutable puede entenderse como una manera de utilizar UML, como un perfil del lenguaje o como la manera formal en la cual pueden especificarse un conjunto de reglas que indican cómo los elementos de UML encajan unos con otros con un propósito concreto: definir la semántica de los modelos construidos con precisión [Mellor and Balcer, 2002]. Para ello, no sólo utiliza *Action Semantics* para especificar con precisión la semántica de acción de los modelos, sino que utiliza un

subconjunto de UML eliminando las construcciones que restan precisión al lenguaje. Desde el punto de vista de MDA, UML ejecutable es la única aproximación en la actualidad que hace uso de *Action Semantics* para modelar el comportamiento en los PIM.

3.2.2. La Propuesta de UML Ejecutable

UML Ejecutable se define como un perfil de UML que define una semántica de ejecución para un subconjunto de UML [Mellor et al., 2004]. Para ello lo que hace, es tomar un subconjunto del lenguaje UML que elimina las construcciones semánticamente ambiguas y añadirle la semántica de ejecución a través de *Action Semantics* (Figura 3.33). La propuesta resultante es *computacionalmente completa*. Esto signfica que es lo suficientemente expresiva para la definición de modelos que pueden ser ejecutados en un ordenador, bien a través de su interpretación o como programas equivalentes generados a partir de los modelos a través de algún tipo de transformaciones automatizadas [OMG, 2005g].

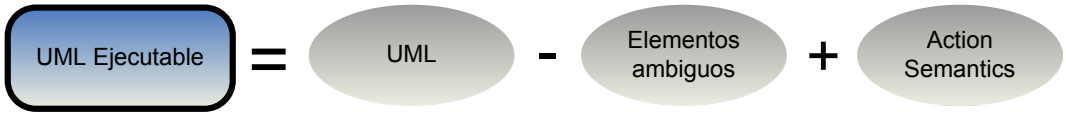

Figura 3.33: Planteamiento básico de UML Ejecutable

Dado que no todos los elementos de UML tienen semántica de ejecución no todos son utilizados en UML ejecutable. Esto no significa que dichos elementos y los diagramas que los representan no tengan valor de cara a construir modelos en UML ejecutable, su valor será el mismo que con UML convencional: artefactos que ayudan a la comprensión de los sistemas, que lo documentan y que facilitan la comunicación. Sin embargo, al no ser adecuados para la especificación de modelos ejecutables, no resultan de interés en el estudio de la propuesta de UML Ejecutable.

Este es el caso de los los diagramas de componentes (§ 3.1.4.1.5) y de despliegue (§ 3.1.4.1.4). Los únicos diagramas estructurales UML utilizados son los diagramas de clases (§ 3.1.4.1.1) para especificar las características estructurales estáticas de los conceptos que se capturan en los dominios. Los diagramas de casos (§ 3.1.4.2.5) de uso sí son utilizados en UML ejecutable. Aunque los casos de uso se modelan con una notación informal sirven para enlazar los modelos ejecutables que realizan cada escenario con los requisitos funcionales que representan. Son por lo tanto valiosos en UML ejecutable para trazar los requisitos a través de los modelos ejecutables [Raistrick et al., 2004].

Los diagramas de actividad (§ 3.1.4.2.2) tampoco son utilizados para modelado formal de comportamiento. Los diagramas de actividad tampoco son considerados en las propuestas de UML Ejecutable de Steve Mellor [Mellor and Balcer, 2002] y de Kennedy Carter [Raistrick et al., 2004]. La razón es que antes de UML 2.0 los

diagramas de actividad eran considerados como una vista extendida de los diagramas de estados y en UML ejecutable se prefieren éstos últimos para representar el comportamiento de los objetos. No obstante, como se explicó en § 3.1.4.2.2, en UML 2 el modelo de actividades sufrió importantes modificaciones. En lugar de ser un caso especial de máquinas de estados, las actividades UML 2 presentan un modelo de ejecución propio y son el mecanismo elegido para coordinar acciones. En versiones anteriores de *Action Semantics*, las acciones eran agregadas por una metaclase `Procedure` que representaba el cuerpo de las operaciones [OMG, 2001]. En UML 2 las acciones son contenidas en comportamientos (`Behavior`) [OMG, 2004b]. El principal mecanismo para agrupar acciones son las actividades (`Activity`), que son un tipo de comportamiento. Para representar el cuerpo de las operaciones en UML 2 se asocian comportamientos a las mismas.

Los diagramas UML utilizados en UML Ejecutable para modelar comportamiento son los diagramas de estados (§ 3.1.4.2.3). El enfoque utilizado es: para modelar el comportamiento de los objetos que cambia con el tiempo, se utilizan diagramas de estados combinados con un lenguaje de acción. Para el comportamiento que no es dependiente del estado de los objetos, se utiliza únicamente el lenguaje de acción [Raistrick et al., 2004].

Los diagramas de interacción (§ 3.1.4.2.4) no son utilizados en UML Ejecutable para modelar comportamiento de una manera formal. No obstante, los diagramas de interacción sí son considerados importantes artefactos de modelado para visualizar la dinámica de los dominios modelados [Mellor and Balcer, 2002]. Se utilizan diagramas de comunicación[23] para describir las comunicaciones existentes entre instancias de máquinas de estados y los diagramas de secuencia para describir las secuencia temporal de interacciones entre objetos.

En la Tabla 3.2.2 se recogen las técnicas de modelado utilizadas en UML Ejecutable para especificar los diferentes aspectos de un sistema. A continuación se analizan las diferentes etapas que propone UML Ejecutable para el desarrollo de un sistema, analizándolas bajo el punto de vista de MDA.

3.2.2.1. Identificar los Dominios del Sistema

El primer paso propuesto en UML Ejecutable es identificar los diversos dominios que conforman el sistema. Un dominio es un mundo autónomo, real, hipotético o abstracto habitado por un conjunto de entidades conceptuales que se comportan de acuerdo a unas reglas y políticas características [Mellor and Balcer, 2002]. En [Raistrick et al., 2004] se clasifican en 2 los tipos de dominios:

Dominios de aplicación. Dominios relacionados directamente con la funcionalidad de la aplicación.

[23] Los diagramas de colaboración en UML 2 se corresponden con los diagramas de colaboración en UML 1.X.

Tabla 3.1: Resumen de técnicas de modelado de conceptos en UML Ejecutable

Conceptos	**Categoría**	**Elementos de Modelado**	**Diagrama UML**
Elementos del dominio del problema	Datos	Clases, atributos, asociaciones, restricciones	Diagrama de clases UML
Ciclo de vida de los elementos	Control	Estados, eventos, transiciones, procedimientos	Diagrama de máquina de estados UML
Tareas a realizar en cada etapa	Algoritmo	Acciones	Lenguaje de acción

Dominios de servicios. Dominios de elementos que ofrecen el soporte técnico para construir la aplicación. Por ejemplo, el dominio del interfaz de usuario o de las alarmas gestionadas por el sistema.

El conocimiento experto de cada dominio se capturará en un PIM expresado mediante UML Ejecutable. Se utiliza el término "tema" (*subject matter*) para indicar el tipo de contenido capturado en cada dominio. El objetivo es evitar que unos dominios estén contaminados con conceptos de otros dominios permitiendo que diferentes expertos modelen diferentes aspectos del sistema. Por otra parte las ventajas de este enfoque coinciden con las ya perseguidas en el diseño estructurado al buscar maximizar la cohesión y minimizar el acoplamiento [Pressman, 1987].

Para representar los dominios se utilizan diagramas de paquetes UML (§ 3.1.4.1.3) y son denominados "diagramas de dominio". Cada paquete del diagrama representa un dominio. Aunque los dominios son autónomos, un dominio puede hacer uso de los servicios ofrecidos por otro dominio. Para modelar estas relaciones se utilizan dependencias UML entre los paquetes que representan los dominios.

3.2.2.2. Definir los Casos de Uso

Como ya se ha mencionado, los casos de uso son utilizados en UML Ejecutable como punto de partida a la hora de construir los PIM formales, si bien los casos de uso por su naturaleza no permiten especificar formalmente comportamiento. En [Mellor and Balcer, 2002] se propone el uso tradicional de los casos de uso (§ 3.1.4.2.5) para capturar los escenarios de interacción dentro de cada dominio en que se ha dividido el sistema. También propone el uso de diagramas de actividad (§ 3.1.4.2.2) para clarificar el comportamiento del sistema en casos de uso complejos, aunque no para especificar formalmente su comportamiento.

En [Raistrick et al., 2004] se propone detallar los escenarios de los casos de uso utilizando diagramas de secuencia UML a nivel de dominio. En estos diagramas, además de los actores que interactúan con el sistema, se describen como interactúan los dominios del sistema entre sí para realizar el escenario. Los dominios son representados en el diagrama como instancias de clasificadores de alto nivel.

3.2.2.3. Construir los Modelos Independientes de la Plataforma

UML Ejecutable no es diferente a otras aproximaciones de desarrollo orientado a objetos en lo que se refiere a cómo se descubren los objetos que realizan cada caso de uso y sus responsabilidades. Para esta tarea se utilizarían las técnicas de análisis y diseño orientadas a objetos propuestas en un método de desarrollo concreto, como por ejemplo UP [Jacobson et al., 1999]. En lo que sí se introducen importantes modificaciones es en cómo se especifican los modelos manejados durante el proceso, esto es, en términos de MDA, cómo se elaboran los PIM.

En la construcción del PIM correspondiente a cada dominio del sistema se identifican tres etapas [Raistrick et al., 2004]:

- Elaboración del modelo de clases.
- Especificación de las máquinas de estado y de las operaciones.
- Definición del comportamiento de acción.

3.2.2.3.1. Elaboración del Modelo de Clases

El modelo de clases construido en UML Ejecutable se corresponde con el modelo de clases de dominio en el análisis orientado a objetos tradicional [Larman, 2004]. Se identifican las entidades conceptuales que intervienen en cada caso de uso, y se crean clases que se corresponden con abstracciones de estos grupos de conceptos. Estas clases se describen desde un punto de vista estructural, esto es, en términos de sus características y asociaciones con otras clases. El modelo de clases constituirá los cimientos para edificar los PIM de cada dominio, ya que el resto de elementos de modelado se basan en él. Para su modelado se utilizan los diagramas de clases vistos en § 3.1.4.1.1.

3.2.2.3.2. Especificación de las Máquinas de Estado y Operaciones

En esta etapa se define parte del comportamiento dinámico del dominio. En concreto, se define el comportamiento de los objetos que cambia a lo largo de su ciclo de vida. Es decir, se describe el modo en el que un objeto reacciona a la recepción de un mensaje cuando su reacción depende de su estado.

Para modelar los modelos de estados de los objetos se utilizan diagramas de máquinas de estados y tablas de estado[24]. Una clase puede recibir estímulos asíncronos (señales) o invocaciones síncronas a operaciones de la misma. En este último caso, se capturarán como operaciones UML de la clase en el modelo de clases (§ 3.2.2.3.1). Debe señalarse que las máquinas de estado utilizadas para modelar la dinámica de los objetos cuyo comportamiento no depende del estado, dan como resultado diagramas muy complejos y poco legibles [Raistrick et al., 2004]. Por ello para modelar el comportamiento independiente del estado UML ejecutable propone

[24]Como se vio en § 3.1.4.2.3 ambas representaciones contienen la misma información.

especificar el cuerpo de los métodos directamente utilizando un lenguaje de acción (§ 3.2.2.3.3).

En UML Ejecutable se recomienda utilizar modelos de estados Moore [Moore, 1956] en los cuales se especifican las acciones a realizar únicamente al entrar en un estado y, además, un evento individual solo puede causar una única transición de salida de un estado dado. Este enfoque proporciona una semántica de ejecución sin ambigüedad y a la vez permite modelar cualquier problema basado en estados [Raistrick et al., 2004].

3.2.2.3.3. Definición del Comportamiento de Acción

En esta etapa se describen las acciones a realizar en detalle. En concreto, estas acciones se definen en dos lugares:

- En los diagramas de estados como respuesta a señales.
- En los métodos que implementan las operaciones.

Para realizar esta descripción UML Ejecutable propone la utilización de un lenguaje de acción que represente una sintaxis concreta de la sintaxis abstracta definida por *Action Semantics* [OMG, 2001]. Los lenguajes de acción permiten especificar las acciones a realizar en términos de la manipulación de los elementos de modelado. Como ejemplos de acciones que pueden ser especificadas pueden señalarse la creación y el borrado de objetos, la búsqueda de objetos que cumplan un criterio determinado, el establecimiento y borrado de asociaciones entre objetos, la navegación a través de asociaciones, la modificación de atributos, la modificación de operaciones, etc.

Como se analizó en § 3.1.4.2, *Action Semantics* forma parte del metamodelo de UML y con la versión 2.0 ha sufrido modificaciones respecto a su planteamiento original. En especial en lo referente a su integración con la nueva semántica de comportamiento definida en UML 2. No obstante, la esencia de esta especificación se mantiene. Se trata de un metamodelo de un lenguaje de acciones que permite especificar unidades mínimas de comportamiento (acciones). El objetivo es poder representar las construcciones de un lenguaje imperativo[25] computacionalmente completo. *Action Semantics* representa por lo tanto una notación abstracta que permite especificar la semántica de ejecución de los modelos con la precisión de un lenguaje de programación.

Sobre la sintaxis abstracta definida en *Action Semantics* se han definido varios lenguajes de acción que implementan una sintaxis concreta. El más conocido es *Action Specification Language* (ASL), desarrollado por Kennedy Carter [Wilkie et al., 2002]. Otros lenguajes de acción que pueden citarse son *Bridge Point Action Language* (AL) [Projtech-Technology, 2006] y *Specification and Design Language* (SDL) [ITU, 1992, ITU, 2000].

[25] De este modo se da respuesta a las limitaciones de los lenguajes puramente declarativos como OCL (§ 3.1.5).

3.2.3. Aportaciones y Carencias para la Investigación

Analizando las posiciones adoptadas en el ámbito del desarrollo dirigido por modelos frente a la propuesta de UML Ejecutable se encuentran resultados un tanto sorprendentes. En un ámbito donde existen posiciones fuertemente encontradas y donde los partidarios de un enfoque realizan fuertes críticas al resto de enfoques, resulta realmente difícil encontrar argumentos razonables en contra o a favor de esta propuesta.

Anneke Kepple, partidaria de utilizar UML junto con OCL para especificar los PIM en MDA, considera que existen dos problemas básicos con UML Ejecutable [Kleppe et al., 2003]:

- El primero es que supone descender al nivel de abstracción de un lenguaje de progamación. Cuestiona si es mejor utilizar un lenguaje de acción que escribir el código fuente directamente en el PSM.

- El segundo problema es que action Semantics no especifica una sintaxis concreta a utilizar y no existe ningún lenguaje de acción estándar.

El primer punto plantea una interesante cuestión: Teniendo en cuenta que MDA plantea elevar el nivel de abstracción a la hora de desarrollar software ¿supone un incremento en el nivel de abstracción el utilizar un lenguaje de acción? Realmente sí se produce un incremento en el nivel de abstracción, puesto que únicamente se manipulan elementos del modelo UML, pero este incremento es muy pequeño con respecto al que se produce al utilizar otras construcciones UML. Al fin y al cabo, en su intento de definir una sintaxis abstracta común para especificar "unidades mínimas de ejecución", *Action Semantics* define metaclases que representan construcciones que se encuentran en los lenguajes de programación actuales: por ejemplo, instrucciones de paso de mensajes o de creación de objetos. Así pues, la siguiente pregunta que surge es ¿es rentable el coste de obtener un incremento pequeño en el nivel de abstracción?

En este sentido, el coste de implementar soluciones basadas en un lenguaje de acción *Action Semantics* presenta dos caras:

- El coste para el usuario, que tiene que aprender un nuevo lenguaje.

- El coste para los desarrolladores de herramientas, que tienen que implementar los procesadores necesarios para generar instancias del metamodelo de *Action Semantics* a partir de sintaxis concretas específicas.

El problema de una falta de sintaxis estandarizada para los lenguajes de acción también ha sido señalado como una debilidad por David Frankel [Frankel, 2003]. Este problema acentúa los dos aspectos negativos señalados anteriormente. La existencia de varias sintaxis concretas que implementan el estándar dificulta su utilización por parte de los usuarios, y su implementación por parte de las herramientas.

Una conclusión importante de cara al futuro es la integración en el metamodelo de UML de *Action Semantics*. Este hecho, junto con las importantes modificaciones

realizadas en el metamodelo de UML 2 para mejorar su integración, indican que el OMG ha acabado reconociendo la necesidad de disponer de lenguajes imperativos como mecanismos de especificación de comportamiento. Este hecho contrasta sin embargo con el nulo soporte que ofrecen en la actualidad las herramientas UML de propósito general más prestigiosas tales como IBM Rational Rose [IBM, 2006], Magic Draw 11 [Magic, 2006] o Borland Together 2006 [Borland, 2006].

Sthephen J. Mellor asocia su propuesta de UML Ejecutable al desarrollo ágil. En [Mellor, 2004], se discute está asociación, argumentando que el poder desarrollar modelos testeables encaja perfectamente con los ciclos cortos de desarrollo y testing propuestos en las metodologías ágiles [Larman, 2003].

Dentro del ámbito del desarrollo dirigido por modelos, debe analizarse la postura de dos partidarios de utilizar DSL específicos a cada problema a resolver, definidos al margen de los estándares del OMG[26]. Una es la de Martin Fowler que deja UML Ejecutable al margen en la fuerte crítica a MDA que realiza en [Fowler, 2005a], indicando que sus partidarios son los más sensatos de entre los defensores de utilizar UML en MDA.

Por otra parte, Steve Cook, también partidario de utilizar DSL diseñados al efecto en la propuesta de Microsoft de *Software Factories* [Greenfield et al., 2004], ha mostrado una opinión también poco argumentada, esta vez negativa, acerca de UML Ejecutable. Dentro del feroz debate que se produjo en *The MDA Journal* en torno a las posturas del OMG y Microsoft, al analizar los enfoque que proponen utilizar UML como un lenguaje de programación, se limita a señalar que se trata de una postura soportada por una comunidad muy pequeña y que no parece que vaya a ganar aceptación comercial en el futuro[27] [Cook, 2004].

Finalmente, cabe destacar un aspecto fundamental de *Action Semantics*. Permite especificar la semántica de ejecución de los modelos con la precisión necesaria para hacerlos ejecutables. Este es un requisito fundamental para MDA y es algo que ni UML ni UML con OCL permiten conseguir[28].

3.3. Meta Object Facility (MOF)

3.3.1. Introducción

En § 3.1 se ha analizado UML, un lenguaje de modelado software de propósito general propuesto por el OMG. Como se analizó en § 3.1.7 existen diversos aspectos de un sistema software que no pueden ser modelados con UML. *Meta Object Facility* (MOF) pretende dar respuesta a este problema. En su propuesta inicial de 1997 [OMG, 1997], MOF especificaba una arquitectura compatible con CORBA para definir y compartir metadatos en entornos heterogéneos distribuidos. En las

[26]MOF puede ser considerado un lenguaje de definición de DSL (§ 3.3).

[27]Precisamente esta afirmación es puesta como ejemplo de falta de argumentos en la respuesta, no menos agresiva, de Michael Guttman [Guttman, 2004].

[28]Incluso los partidarios de una combinación UML+OCL reconocen este hecho [Kleppe et al., 2003].

versiones siguientes, MOF evolucionó hacia un *framework* que permite definir y manipular metadatos. Un ejemplo de metadatos son las construcciones que pueden ser utilizadas en un lenguaje y que constituyen su metamodelo.

Por lo tanto, MOF puede ser utilizado para formalizar lenguajes a través de su metamodelo y hacer que dichos lenguajes se beneficien de los servicios proporcionados junto con el *framework*. Tal y como se vio en § 3.1, UML es un lenguaje definido formalmente por una sintaxis abstracta expresada en términos de su metamodelo. Debido a circunstancias relacionadas con el origen y evolución de ambos estándares, en versiones anteriores a la 2.0, MOF y el metamodelo de UML [OMG, 2004b] estaban desalineados. Por ello, se han realizado substanciales modificaciones en la versión 2.0 de UML y de MOF [OMG, 2006b] con el objetivo de alinear los conceptos de modelado manejados en ambos estándares.

3.3.2. Metaniveles MOF

La arquitectura de MOF suele describirse en base a cuatro metaniveles (Tabla 3.3.2) [Frankel, 2003]:

Nivel M3. Comprende las construcciones proporcionadas por MOF para definir metamodelos[29]. Éstas construcciones son básicamente un conjunto reducido de las construcciones que constituyen el metamodelo de UML, y que en la versión 2.0 de ambas especificaciones se han definido en un documento aparte para permitir su reutilización [OMG, 2005k].

Nivel M2. Comprende los metamodelos definidos mediante construcciones MOF. Comprende metamodelos estandarizados, como el de UML, y otros metamodelos de lenguajes definidos por los usuarios de MOF. Las construcciones utilizadas en M2 son realmente instancias de las construcciones de M3.

Nivel M1. Comprende modelos formados por construcciones de M2. Es decir, modelos definidos con lenguajes cuyas sintaxis abstractas se definen en términos de metamodelos M2.

Nivel M0. Comprende objetos y datos que son instancias de elementos M1. Estos elementos resultan de la instanciación de los elementos de los lenguajes definidos en M1.

En la Figura 3.34 se muestra un un ejemplo que, a través de un diagrama de clases UML, busca ilustrar qué construcciones se manejarían en cada metanivel y como éstas se relacionan entre sí. En el nivel M3 se define el concepto de clase a través de la construcción (`Class`). Utilizando esta metaclase, se definen en el nivel M2 las clases correspondientes a determinados conceptos de UML: atributo (`Attribute`), clase (`Class`) e instancia (`Instance`). En el nivel M1, el usuario utilizaría estas construcciones para definir su modelo: una clase `Persona` con un atributo `nombre`,

[29] Por esta razón en ocasiones se denomina a MOF meta-metamodelo.

Tabla 3.2: Metaniveles MOF

Metanivel	Descripción	Ejemplo
M3	Construcciones utilizadas para describir metamodelos	Clases MOF, Asociaciones MOF , Atributos MOF
M2	Metamodelos que comprenden construcciones instancias de M3	Clases UML, Asociaciones UML, Atributos UML
M1	Modelos instancias de las construcciones M2	Clase "Cliente", atributo "nombre"
M0	Objetos y datos instancias de las construcciones de M1	Cliente de nombre "Pepe" y edad 40

y también modela una instancia de dicha clase con el valor "`pepe`" para el atributo `nombre`. Finalmente, en el nivel M0 se representa una instancia de la clase `Persona` en tiempo de ejecución.

Realmente, no debe entenderse esta enumeración de niveles como algo rígido. Los dos conceptos claves de modelado en MOF son *Clasificador* (Clase) e *Instancia* (Objeto) y la posibilidad de navegar desde una instancia hasta su metaobjeto (su clasificador) [OMG, 2006b]. Este enfoque puede utilizarse para manejar cualquier número de metaniveles mayor o igual que dos. En la práctica, los sistemas más utilizados se basan en los cuatro metaniveles expuestos. Como ejemplo de sistema con un número inferior de niveles puede señalarse los sistemas reflectivos genéricos con dos niveles (construcciones manejadas en el intérprete y construcciones manejadas en el intérprete del intérprete) [Ortín Soler, 2001].

3.3.3. Especificación de MOF

La especificación de MOF 2.0 [OMG, 2006b] supone una revisión profunda de la especificación anterior MOF 1.4 [OMG, 2002a]. El objetivo ha sido alinear MOF con UML, de tal manera que las construcciones definidas por MOF coincidan plenamente con las utilizadas para definir el metamodelo de UML.

El enfoque utilizado para conseguir este alineamiento ha sido definir un núcleo de elementos de metamodelado reutilizable. La idea es sencilla: si tanto en MOF como en UML se maneja los conceptos de "clase" y de "generalización" ¿por qué no reutilizarlos en ambas especificaciones?. Este núcleo de metalenguaje común se define en la especificación *UML Infrastructure* [OMG, 2005k], en la biblioteca denominada *Core* (Núcleo).

La biblioteca *Core* de *UML Infrastructure* se considera el núcleo arquitectónico de MDA [OMG, 2005k]. Su objetivo es que el metamodelo de UML y otros metamodelos de MDA reutilicen total o parcialmente las construcciones de esta biblioteca. Dado que MOF es el lenguaje utilizado para definir metamodelos en MDA, MOF se edifica sobre este núcleo. Como el resto de metamodelos se especifican utilizando MOF, utilizan directamente las construcciones de esta biblioteca.

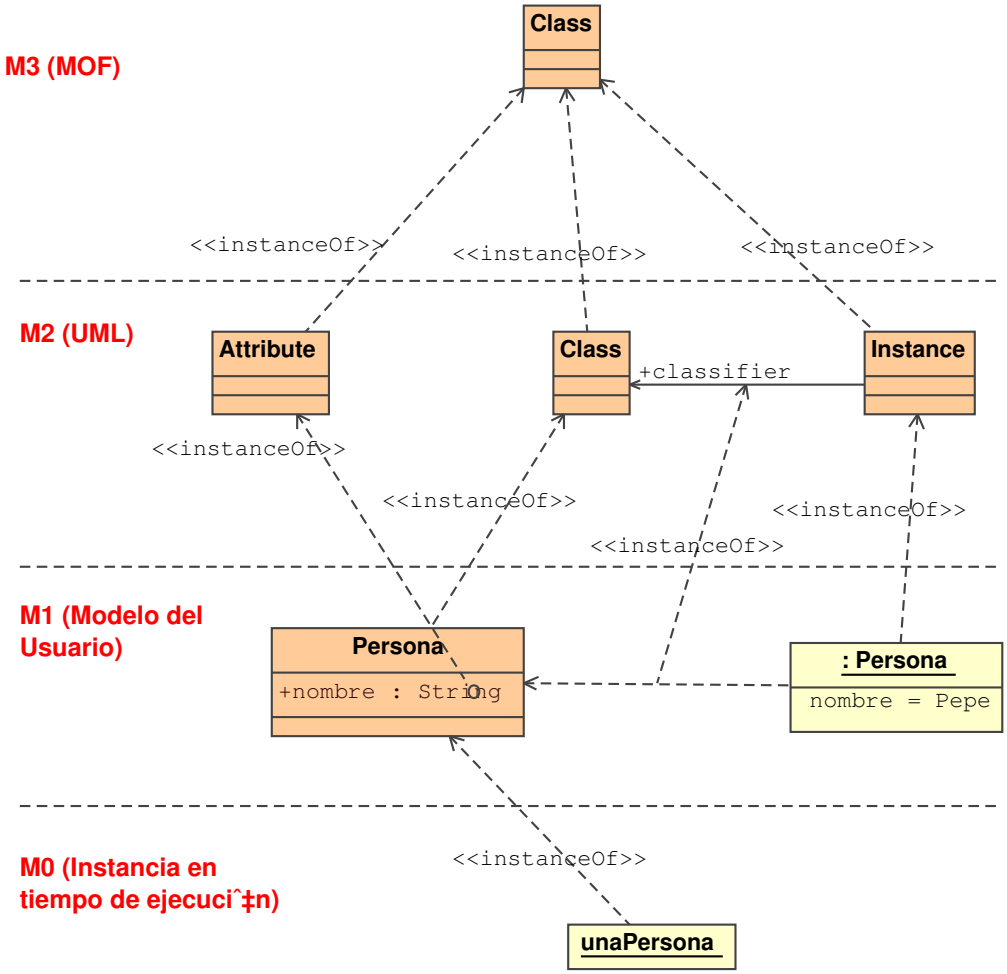

Figura 3.34: Ejemplo de construcciones utilizadas en los cuatro metaniveles MOF

Las ventajas de este enfoque son las derivadas de la reutilización: los metamodelos se benefician de la sintaxis abstracta y semántica que ya han sido definidas.

En la Figura 3.35 se muestra la estructura de paquetes interna de la biblioteca *Core* así como otros estándares del OMG que utilizan las construcciones definidas en ella. Además de UML y de MOF, se recoge el estándar *Common Warehouse Metamodel* (CWM). Se trata de un estándar para el intercambio de metadatos en aplicaciones de almacén de datos e inteligencia de negocio (*data warehouse*). El metamodelo de dicho estándar se define utilizando MOF. Por otra parte, el paquete de Perfiles (*Profiles*) permite definir extensiones a metamodelos existentes, y será analizado en § 3.1.6.

Tal y como se muestra en la Figura 3.35, la biblioteca *Core* está compuesta de cuatro paquetes [OMG, 2005k]:

***PrimitiveTypes*.** Contiene un conjunto de tipos predefinidos que son habitualmente

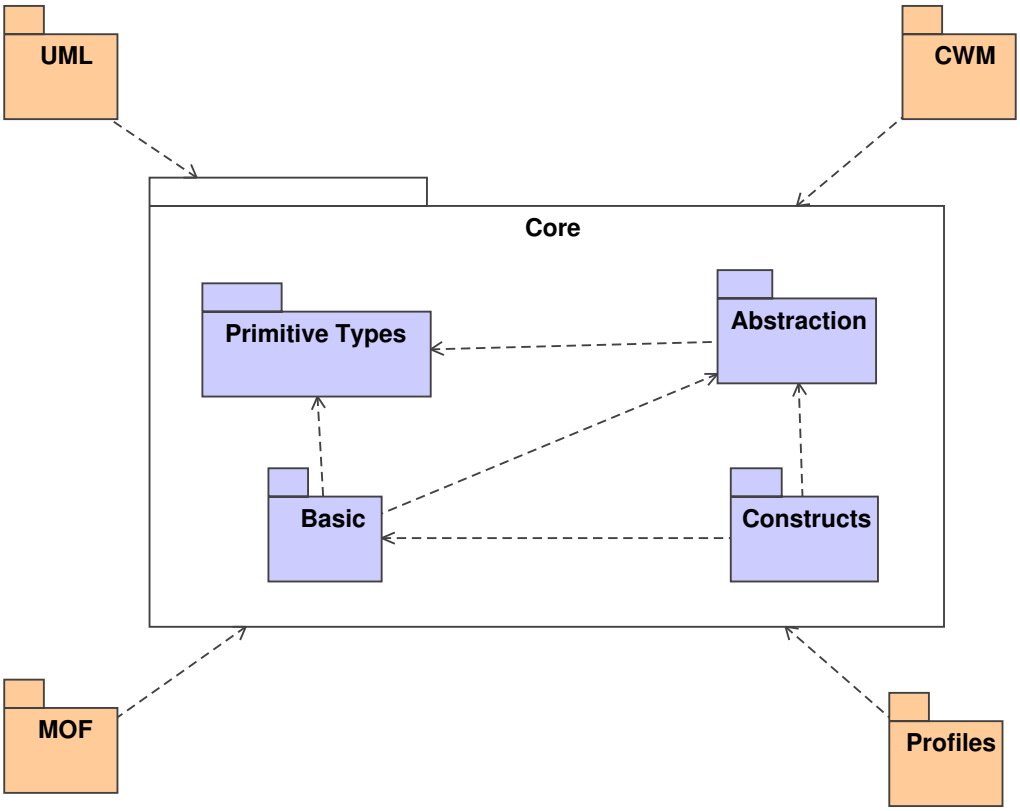

Figura 3.35: Biblioteca Core de UML Infrastructure

utilizados al definir metamodelos. Ha sido diseñado teniendo en cuenta específicamente las necesidades de MOF y de UML, por lo que otros metamodelos puede que tengan que solapar determinados conjuntos de tipos primitivos. Los tipos primitivos predefinidos son únicamente cuatro: booleanos (`Boolean`), enteros (`Integer`), naturales ilimitados (`UnlimitedNatural`) y cadenas de caracteres (`String`).

Abstractions. Contiene metaclases abstractas pensadas para ser especializadas o de utilización común por muchos metamodelos. Un ejemplo es la construcción utilizada para definir espacios de nombre (`Namespace`). Es habitual en lenguajes de modelado que existan construcciones que contengan a otras y que definan un espacio de nombres que permita identificar a las construcciones contenidas unívocamente. Por ejemplo, en UML, un paquete define un espacio de nombre sobre los elementos UML que contiene; una clase UML define un espacio de nombres sobre las operaciones que contiene. Esta clase será por lo tanto una clase habitualmente especializada al definir metamodelos de lenguajes.

Constructs. Contiene principalmente metaclases concretas que se consideran fun-

damentales para el modelado orientado a objetos. Este paquete en concreto es reutilizado completamente tanto por MOF como por UML y representa una parte significativa del trabajo realizado de alineamiento de ambos modelos. en este paquete se definen conceptos clave como el de clasificador (`Classifier`), clase (`Class`), operación (`Operation`), relación de asociación (`Association`).

***Basics*.** Define las construcciones de un lenguaje de modelado basado en clases mínimo. Se ha diseñado para ser utilizado en la Capa Esencial (*Essential Layer*) de MOF. Tal y como se muestra en la Figura 3.35, el paquete *Basics* depende de otros paquetes de la biblioteca: importa elementos del paquete *PrimitiveTypes*, contiene metaclases derivadas de las contenidas en *Abstractions* y define construcciones básicas que son reutilizadas por el paquete *Constructs*. Las construcciones que contiene son utilizadas también como la base para la producción de XMI (§ 3.3.4.1) para UML, MOF y otros metamodelos basados en la biblioteca *Core*.

MOF 2 se edifica sobre un subconjunto de las construcciones definidas en la biblioteca *Core*. El Modelo de MOF 2 comprende dos paquetes principales: *Essential MOF* (EMOF) y *Complete MOF* (CMOF) [OMG, 2005k].

EMOF. Define un subconjunto de MOF que se corresponde con los servicios habitualmente encontrados en lenguajes de programación orientados a objetos y XML. Su principal aportación es que proporciona un *framework* sencillo para trasladar modelos MOF de metamodelos sencillos a implementaciones como *Java Metadata Interface* (JMI) (§ 3.3.4.3) y XMI (§ 3.3.4.1). Este paquete se basa en el paquete *Basics* de la biblioteca *Core* de la Infraestructura de UML, al que impone ciertas restricciones y extensiones.

CMOF. Extiende los conceptos sencillos de modelado de EMOF para permitir la elaboración de metamodelos más sofisticados. Esta extensión la realiza utilizando la generalización de clases. Esta paquete se construye sobre EMOF y el paquete *Constructs* de la biblioteca *Core*. No define ninguna clase nueva, sino que mezcla dichos paquetes con extensiones que, juntos, definen las capacidades de metamodelado básicas.

Además de estos dos subsistemas, el modelo de MOF incluye capacidades adicionales definidas en paquetes separados, como por ejemplo, tipos primitivos adicionales, la capacidad de descubrir y manipular metadatos y metaobjetos (reflexión), o un sencillo mecanismo de extensión de los elementos de modelado utilizando pares atributo–valor (*tags*) [OMG, 2005k].

3.3.4. El Papel de MOF en MDA

MOF juega un papel fundamental en MDA. De este estándar se ha dicho que es la tecnología central que va a hacer posible la realización de la propuesta MDA

[Kleppe et al., 2003], y la tecnología clave con la que se definen los cimientos de la propuesta [Frankel, 2003]. Así pues, conviene analizar qué beneficios ofrece MOF a la propuesta MDA.

Tal y como se ha explicado en los apartados anteriores, MOF permite especificar, utilizando un conjunto de construcciones comunes, el metamodelo que define la sintaxis abstracta de un lenguaje de modelado. Así, pueden especificarse formalmente diferentes lenguajes de modelado para modelar diferentes aspectos del sistema. Esta característica se considera esencial para que la propuesta de MDA pueda ser factible [Frankel, 2004a].

Las características de las especificaciones de metamodelos realizadas utilizando MOF ya se han estudiado al revisar el metamodelo de UML (§ 3.1). Se define una sintaxis abstracta formal de los lenguajes de modelado. Esta sintaxis refleja las construcciones, esto es, los aspectos estructurales de dichos lenguajes. También permite definir aspectos semánticos del lenguaje. Al igual que sucedía con UML, existe una semántica implícita en las construcciones de MOF utilizadas, por ejemplo la semántica de una asociación de composición o una generalización. También pueden definirse restricciones explícitas formalmente utilizando OCL (§ 3.1.5). Sin embargo, al igual que ocurría en la especificación de UML, existe una gran parte de aspectos semánticos del lenguaje que no pueden ser expresados utilizando OCL. En tal caso la única alternativa existente es expresarlos textualmente.

Así pues MOF permite definir lenguajes utilizando metamodelos en base a construcciones que tienen una sintaxis, semántica y estructura común. La principal ventaja de este enfoque la obtienen las herramientas informáticas que manipulan modelos, dentro de las cuales una herramienta MDA es un caso particular. Como se señala en [OMG, 2004b], el hecho de definir un lenguaje de modelado en términos de su metamodelo no responde a criterios de formalidad, sino al intento de que sea fácilmente comprensible por desarrolladores de herramientas. Si las construcciones del metamodelo se corresponden con conceptos básicos de la orientación a objetos, entonces pueden implementarse dichas construcciones utilizando un lenguaje de programación. El disponer de un metamodelo implementado como software, permite su manipulación automatizada.

Esta es la razón por la que es tan importante MOF para MDA. Habilita el desarrollo de servicios informáticos que manipulen modelos de una manera automatizada. Si dichos servicios manipulan lenguajes en base a los elementos de su metamodelo MOF, y a la vez MOF permite definir distintos lenguajes de modelado, entonces se obtienen servicios que pueden reutilizarse con diferentes lenguajes modelado.

Un caso particular de servicios automatizados que pueden construirse sobre MOF son las traslaciones (*mappings*) que pueden realizarse entre la sintaxis abstracta definida en términos de MOF y diferentes sintaxis concretas para diferentes propósitos. Se han definido diferentes traslaciones estándar a diferentes representaciones, dentro de las cuales la más conocida es XMI [OMG, 2005c], que permite representar modelos MOF mediante XML. A continuación se analizan varias de éstas representaciones.

3.3.4.1. XMI

XML Metadata Interchange (XMI) es un estándar del OMG que permite expresar mediante XML modelos escritos en un lenguaje especificado por un metamodelo MOF. Para ello, la versión actual de XMI 2.1 [OMG, 2005c] ofrece dos mecanismos:

- Un conjunto de reglas que definen como generar un *XML Schema Definition* (XSD) [W3C, 2004] a partir de un metamodelo basado en MOF. Los esquemas generados permiten validar las instancias de dicho metamodelo expresadas como XML.

- El proceso que debe seguirse para generar un documento XML a partir de un modelo basado en la librería *Core* sobre la que se edifica MOF 2.0 (§ 3.3.3). El proceso a seguir queda definido por un conjunto de reglas de producción. El resultado de aplicar dichas reglas a un modelo o a un fragmento de un modelo es un documento XML.

Para la definición de las reglas de producción en ambos casos se utiliza la notación EBNF junto con anotaciones textuales. De este modo se define como se generan los fragmentos de XML relativos a la definición del esquema y la definición del documento XML propiamente dicho .

XMI define su principal objetivo en sus iniciales: habilitar el intercambio de metadatos entre herramientas. Por ello utiliza XML, un lenguaje optimizado para facilitar su manipulación automatizada [Harold and Means, 2004] ampliamente extendido en la actualidad. Aunque nada impide a un usuario generar un documento XMI manualmente, su uso lógico es por parte de una herramienta que genere o utilice modelos. XMI permite el intercambio de modelos basados en MOF entre herramientas. En concreto, permite:

- Una herramienta de modelado que permita al usuario definir el modelo MOF de un conjunto de metadatos, podrá generar el *XML Schema Definition* (XSD) que permita validar instancias de dichos metadatos.

- Una herramienta que permita modelar utilizando un lenguaje definido en base a un metamodelo MOF, podrá exportar los modelos creados utilizando dicho lenguaje a un formato XML estándar.

- Una herramienta que acepte como entrada modelos definidos en un lenguaje con un metamodelo MOF, podrá leer dichos modelos en formato XML definido por XMI.

Un error habitual acerca de XMI es concebirlo como un formato XML para el intercambio de modelos UML. La razón, es que UML es con diferencia el lenguaje con un metamodelo basado en MOF más conocido. Se va a mostrar un ejemplo de cómo un sencillo modelo UML puede ser representado con XMI. En la Figura

3.36 se representa una clase UML `Persona` con un atributo `edad` de tipo entero y visibilidad privada. En el Listado 3.9 se muestra la representación XMI de esta clase, generada por la herramienta UML MagicDraw [Magic, 2006][30].

En la línea 4 se define la etiqueta raíz del modelo UML. No se corresponde con ninguna construcción del metamodelo de UML, es un artefacto XMI para declarar la etiqueta contenedora del modelo[31]. En la línea 5 se define la clase `Persona` mediante la etiqueta `ownedMember`. Con el atributo `uml:Class` se define la metaclase del objeto: en este caso la metaclase UML `Class`. El metamodelo de UML define el espacio de nombres XML "uml", donde se definen todas sus construcciones. El resto de atributos de la etiqueta `ownedMember` son utilizados para asignar valores a las propiedades de la instancia de la metaclase. En el ejemplo, el nombre de la clase (`Persona`) y la visibilidad de la clase dentro del paquete (`public`). Mediante una etiqueta contenida declara la propiedad `edad` dentro de la clase `Persona`. El tipo de edad es un tipo `Integer`. `Integer` en el metamodelo de UML [OMG, 2004b] es una instancia de un tipo primitivo (`PrimitiveType`) que a su vez es una instancia de `DataType`. Esta instancia se representa mediante el atributo `href` dentro de la etiqueta `type`. El valor de `href` es un identificador que identifica a la instancia `Integer` dentro del metamodelo de UML 2 que implementa la herramienta. Este metamodelo expresado en XMI es importando como un paquete UML más en la línea 10.

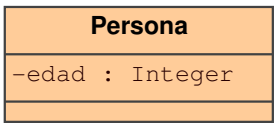

Figura 3.36: Clase Persona a representar con XMI

Listado 3.9: Fichero XMI que describe la clase Persona

```
1  <xmi:XMI xmi:version="2.1" timestamp="Fri Jul 07 17:47:12
       CEST 2006" xmlns:uml="http://schema.omg.org/spec/UML/2.0"
2  xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
3    <xmi:Documentation xmi:Exporter="MagicDraw UML" xmi:
       ExporterVersion="11.0"/>
4    <uml:Model xmi:id="eee_1045467100313_135436_1" name="Data"
       visibility="public">
5      <ownedMember xmi:type="uml:Class" xmi:id="
         _11_0_1_65c0218_1152211954734_844307_2" name="Persona"
         visibility="public">
6        <ownedAttribute xmi:type="uml:Property" xmi:id="
           _11_0_1_65c0218_1152287219074_911880_1" name="edad"
           visibility="private">
```

[30]Se han eliminado diversos fragmentos correspondientes a extensiones de la herramienta. XMI soporta extensiones propias de las herramientas que lo utilicen a través de la etiqueta `Extension`.

[31]En el caso de Magic Draw, el paquete raíz del proyecto se denomina `data`.

```
7         <type xmi:type="uml:DataType" href="
              UML_Standard_Profile.xml|
              donce_1051693917650_319078_0"/>
8       </ownedAttribute>
9     </ownedMember>
10    <ownedMember xmi:type="uml:Package" href="
          UML_Standard_Profile.xml|
          magicdraw_uml_standard_profile_v_0001"/>
11  </uml:Model>
12 </xmi:XMI>
```

Como se ha visto en el ejemplo anterior el propio metamodelo de UML puede especificarse mediante XMI al estar basado en MOF. Es general, XMI permite no sólo el intercambio de instancias de metamodelos MOF, sino también de los propios metamodelos. Esto es posible debido a la arquitectura vista en § 3.3.3: el metamodelo de MOF (es decir, el meta-metamodelo de UML) se modela también con MOF.

Finalmente debe señalarse que la especificación de XMI [OMG, 2005c] contempla la generación de las definiciones de objetos MOF a partir de su representación como XML. En ella se describen, de una manera informal, cómo pueden generarse definiciones MOF a partir de especificaciones XML. En concreto se contemplan tres algoritmos:

- Producción de MOF a partir de DTD.
- Producción de MOF a partir de XML.
- Producción de MOF a partir de XML Schema.

En los tres casos existen varias posibilidades de traslación (*mapping*), debido a que las representaciones XML y DTD no son lo suficientemente ricas como para producir una única representación MOF sin ambigüedad [OMG, 2005c]. Lo que se hace en la práctica para poder generar una representación MOF automáticamente a partir de su especificación XMI es enriquecer ésta con información adicional y utilizar convenios predefinidos (un caso concreto puede verse en § 3.3.5.1).

3.3.4.2. HUTN

Human-Usable Textual Notation (HUTN) es una especificación del OMG [OMG, 2004c] que define una solución genérica para generar lenguajes textuales a partir de modelos MOF. A diferencia de XMI (§ 3.3.4.1), que también representa un formato de serialización genérico para modelos y metamodelos MOF, el objetivo de HUTN es que las representaciones sean fácilmente entendibles a la hora de ser leídas y escritas por usuarios humanos. Como se explicó en § 3.3.4.1, XMI está basado en XML, que es un lenguaje optimizado para su manejo por parte de software. Como puede observarse en el Listado 3.9, XMI no resulta apropiado para su escritura y lectura manual.

La utilidad de HUTN queda caracterizada por tres propiedades [OMG, 2004c]:

Genérico . La especificación define un conjunto de reglas sintácticas que cubren completamente todos los conceptos de modelado MOF. De esta manera, HUTN puede utilizarse para definir un lenguaje para cualquier modelo especificado mediante MOF.

Automatizado. La generación de la especificación HUTN a partir de un modelo MOF, así como la generación del sistema analizador (*parser*) que permita procesar usos de la notación generada, pueden ser totalmente automatizadas. Esto permite absorber rápidamente los cambios producidos tanto en el modelo MOF que especifica el metamodelo del lenguaje utilizado, como cambios en la notación HUTN equivalentes.

Usable. El conjunto de convenios para definir lenguajes HUTN tiene como objetivo fundamental el lograr la usabilidad de los lenguajes generados. En particular, se acordó que la audiencia de los lenguajes HUTN generados estaría en cierta medida familiarizada con los lenguajes de programación tradicionales, aunque no se requeriría un conocimiento avanzado de los mismos en ningún caso. En este sentido, para su especificación se basaron en dos trabajos sobre usabilidad de lenguajes de programación [McIver and Conway, 1996, Richard and Ledgard, 1977].

El uso de HUTN va a ilustrarse con un ejemplo. En la parte superior Figura 3.37 se recoge la definición de un sencillo metamodelo MOF. En el modelo existen personas (`Persona`), coches (`Coche`) y perros (`Perro`). Una persona posee coches, definido mediante una relación de agregación. El tipo de trabajo que puede tener una persona se representa mediante una enumeración `Trabajo`. En la parte inferior de la figura se recoge una configuración de objetos instanciada a partir de dicho metamodelo: Se definen dos personas con dos coches y un perro.

En el Listado 3.10 se recoge la representación HUTN del modelo de objetos representado en Figura 3.37. HUTN permite utilizar una serie de atajos y convenios que hacen más sencillas de leer y escribir sus representaciones textuales. Por ejemplo, cuando un atributo de un objeto es booleano o de tipo enumeración, permite considerarlo un "adjetivo" de la instancia y representarlo al principio de la declaración, caracterizando a la misma. Este es el caso de la profesión de las personas o la agresividad de los perros. Por otra parte, las instancias deben ser unívocamente identificables, por ejemplo, para poderles hacer referencia en otras construcciones tales como asociaciones. El identificador de los objetos puede ser un valor arbitrario, como en el caso de los atributos `id` en XMI (ver Listado 3.9), aunque se permite que sea un atributo del clasificador, habitualmente mucho más sencillo de leer. En el ejemplo del Listado 3.10, el nombre de las personas[32] y la matrícula de los coches juegan el papel de este identificador único. HUTN permite omitir la especificación de las referencias a objetos, cuándo estas pueden deducirse del contexto, como es el caso de la asociación `poseeCoche`. Obsérvese también como la relación de agre-

[32]En un ejemplo real el nombre de una persona no valdría por no garantizar la unicidad.

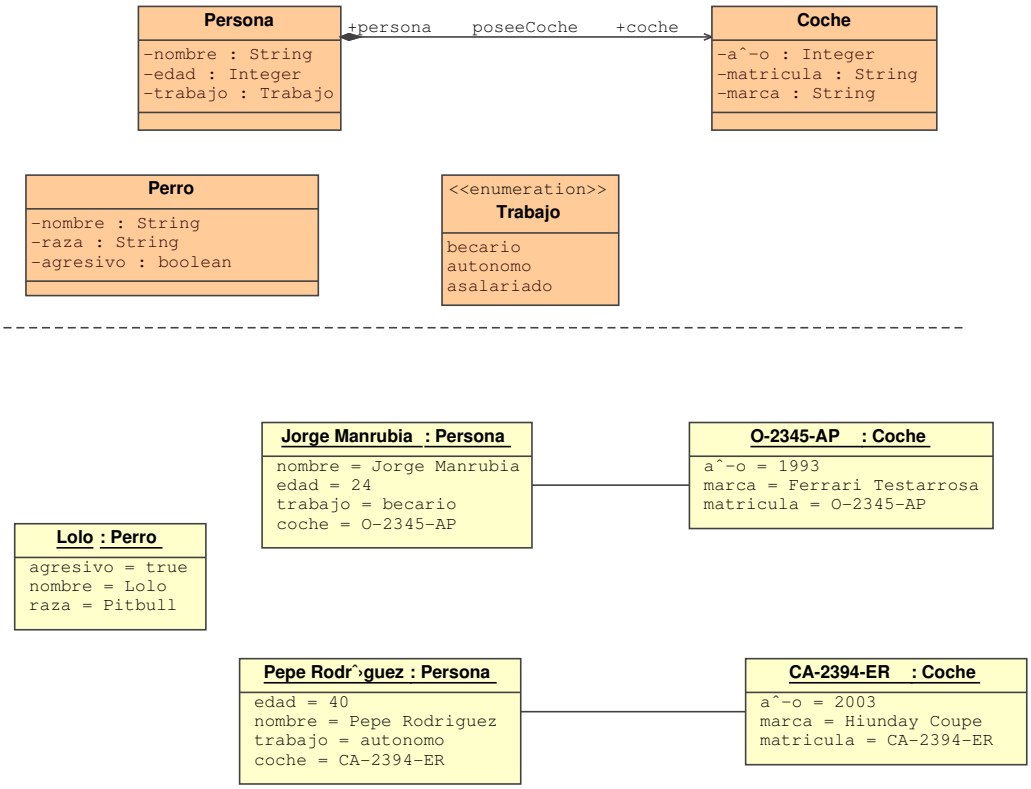

Figura 3.37: Modelo MOF del metamodelo del lenguaje representado mediante HUTN

gación se representa directamente como un atributo, donde el valor es la definición del propio objeto agregado, en este caso un coche.

Listado 3.10: Ejemplo de representación HUTN de un modelo MOF

```
becario Persona "Jorge␣Manrubia"{
  edad: 24;

  poseeCoche: "O-2345-AP"{
    marca: "Ferrari␣Testarrosa"
    año: 1993
  }
}

autónomo Persona "Pepe␣Rodríguez"{
  edad: 40;

  poseeCoche: "CA-2394-ER"{
    marca:"Hiunday␣Coupe"
    año: 2003;
  }
}
```

```
agresivo Perro "Lolo"{
  raza: "Pitbull"
}
```

El cómo generar el lenguaje HUTN concreto, dado un metamodelo MOF, se especifica mediante un lenguaje definido en base a una sintaxis abstracta cuyo metamodelo se proporciona en [OMG, 2004c]. Este metamodelo expresa las opciones de configuración relativas a convenios y atajos que especializan diferentes usos de HUTN, si bien todos estos lenguajes responden a un sintaxis común . Las abstracciones de dicho metamodelo se recogen en la Figura 3.38. Por ejemplo, la metaclase `ClassConfig` identifica una metaclase concreto del modelo MOF para la cual se va a configurar la representación textual HUTN. La metaclase `IdentifierConfig` , que especializa la anterior, identifica a un atributo concreto de una clase como único en su ámbito, y permite utilizarlo para identificar unívocamente una instancia de la misma. Otro ejemplo es la metaclase `EnumAdjectiveConfig`, que identifica un atributo de la clase cuyo valor puede aparecer como un adjetivo para las instancias de la clase.

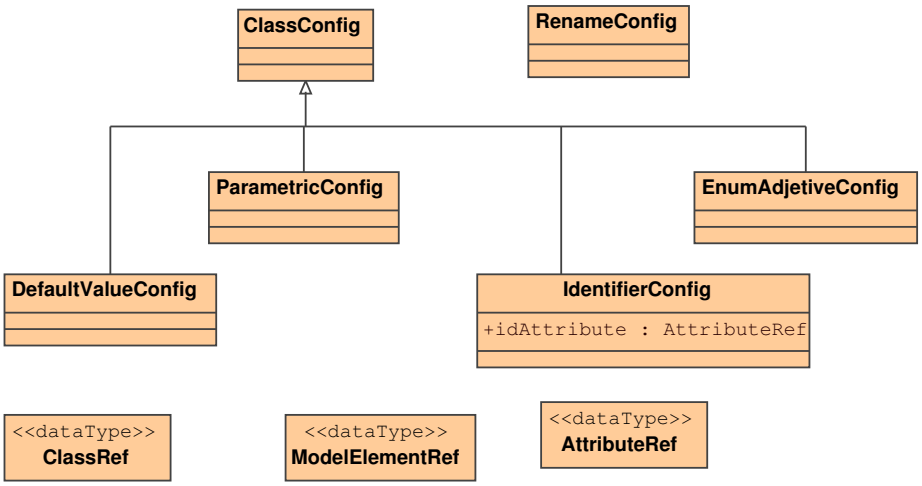

Figura 3.38: Metamodelo de configuraciones HUTN

La sintaxis HUTN viene especificada mediante una gramática expresada mediante reglas EBNF [Cueva Lovelle, 1998]. La gramática viene intercalada con explicaciones textuales que especifican mediante lenguaje natural cómo debe interpretarse. En el Listado 3.11 se recoge que especifica la sintaxis a utilizar para expresar instancias de clases. Los números entre corchetes se utilizan únicamente para identificar las reglas donde se referencian y donde son explotadas.

Listado 3.11: Especificación de la sintaxis HUTN mediante EBNF

```
[6] ClassInstance := 7:ClassHeader 10:ParametricAttrs
                     '{' 11:ClassContents '}' (';')?
```

```
[7] ClassHeader := 8:ClassAdjectives <ClassName>
                   (9:ClassIdentifier)?
```

Cuando estas reglas de producción se aplican a un metamodelo MOF, teniendo en cuenta las opciones de configuración especificadas, se obtiene su representación HUTN. Aunque la especificación actual de HUTN no indica que el metamodelo de lenguaje de configuración esté basado en MOF, sí especifica una representación HUTN de dicho metamodelo [OMG, 2004c]. Con este mecanismo se permite expresar las opciones de configuración HUTN utilizando una sintaxis HUTN.

3.3.4.3. JMI

Java Metadata Interface (JMI) es un estándar que especifica como trasladar modelos MOF para Java. Este estándar fue desarrollado en *Java Community Process* (JCP) en la JSR-40 [Sun, 2002]. En dicha especificación se define JMI como un servicio extensible de metadatos para la plataforma Java que proporciona un modelo de programación Java común para el acceso a metadatos.

JMI define como generar APIs que permitan manipular metamodelos MOF. Para cualquier modelo MOF, JMI define las plantillas necesarias para generar las APIs Java que permiten acceder y actualizar los metadatos instanciados de dicho modelo. Además, define un conjunto de interfaces reflectivos que pueden utilizarse para descubrir y manipular metamodelos MOF sin necesidad de tener un conocimiento previo del metamodelo.

Las reglas de traslación de MOF a Java definen cuatro tipos de metaobjetos de nivel M1 (§ 3.3.2):

Objetos *proxy* de clases. El interfaz de un objeto proxy de clases proporciona un conjunto de operaciones para acceder y actualizar el estado (atributos) del clasificador al que referencian. También proporcionan operaciones *factory* para permitir a sus usuarios crear instancias de objetos[33].

Objetos instancia. Un objeto instancia mantiene el estado correspondiente a los atributos de la instancia. Los objetos instancia están siempre ligados a un objeto proxy de clase, el cual proporciona operaciones para crear objetos instancia.

Objetos asociación. Son objetos que contienen una colección de enlaces correspondientes a una asociación definida en el metamodelo. Proporciona así mismo interfaces para navegar por los enlaces y modificarlos.

Objetos paquete. Representan un directorio de operaciones que permiten acceder a las colecciones de objetos definidas en un metamodelo.

A continuación se mostrará un ejemplo de los interfaces JMI que representan un modelo MOF muy sencillo. Dicho modelo, constará únicamente de una clase

[33] *Proxy* y *factory method* son patrones de diseño GoF [Gamma et al., 1995].

`Persona` (Figura 3.36). Para dicha clase, aplicando las reglas de traslación definidas en [Sun, 2002], se definirían dos interfaces[34]. A partir de dicho metamodelo se generarían un interfaz que define el proxy a la clase `Persona` (Listado 3.12) y otro interfaz que define los objetos instancia de `Persona` (Listado 3.13).

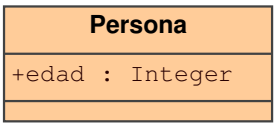

Figura 3.39: Metamodelo MOF compuesto de una única metaclase

Listado 3.12: Interfaz JMI correspondiente a los objetos proxy de la clase Persona

```
/**
 * Persona class proxy interface.
 */
public interface PersonaClass extends javax.jmi.reflect.
    RefClass {
    /**
     * The default factory operation used to create an
        instance object.
     * @return The created instance object.
     */
    public Persona createPersona();

    /**
     * Creates an instance object having attributes
        initialized by the passed
     * values.
     * @param edad
     * @return The created instance object.
     */
    public Persona createPersona(int edad);
}
```

Listado 3.13: Interfaz JMI correspondiente a los objetos instancia de Persona

```
/**
 * Persona object instance interface.
 */
public interface Persona extends javax.jmi.reflect.RefObject
    {
    /**
     * Returns the value of attribute edad.
     * @return Value of attribute edad.
     */
```

[34]El código mostrado corresponde al código generado por MDR, implementación de MOF que permite acceder a los metadatos mediante interfaces JMI § 3.3.5.1.

```
    public int getEdad();

    /**
     * Sets the value of edad attribute.
     * @param newValue New value to be set.
     */
    public void setEdad(int newValue);
}
```

La implementación de dichos interfaces corresponderá a la herramienta que exponga metamodelos MOF a través de ellos. JMI está pensado por lo tanto para estandarizar la manipulación de modelos MOF programáticamente utilizando Java.

3.3.5. Implementaciones de MOF

A continuación se presentan las dos implementaciones de MOF libres más conocidas. Debe destacarse que detrás de cada una de ellas existe una gran empresa del sector: Sun en un caso e IBM en el otro. Ambas implementaciones forman parte de la arquitectura de sendos entornos de desarrollo.

3.3.5.1. MDR

Metadata Repository (MDR) es una implementación del estándar MOF [NetBeans, 2006a] desarrollada dentro de la plataforma de herramientas de desarrollo NetBeans [NetBeans, 2006b]. En su versión actual, implementa la versión 1.4 de MOF y permite importar y exportar modelos utilizando XMI versión 1.1 y 1.2. Además, puede accederse a los metadatos del repositorio programáticamente utilizando la API JMI (§ 3.3.4.3), tanto en su versión específica al metamodelo como en su versión reflectiva genérica.

Existe un catálogo de implementaciones de metamodelos basados en MOF para utilizar con MDR[35] La versión del metamodelo de UML más alta soportada en la actualidad es la 1.5. Dichos metamodelos son cargados por MDR en su forma XMI complementada con etiquetas específicas JMI para guiar el proceso de generación de la implementación de las metaclases.

3.3.5.2. EMF

Eclipse Modeling Framework (EMF) [Eclipse, 2006e] es un framework de modelado y una utilidad de generación de código desarrollada como parte del proyecto Eclipse [Eclipse, 2006c]. El *framework* se edifica en torno la implementación de un metamodelo[36] denominado *ECore*. Dicho metamodelo es similar a MOF aunque no son compatibles [Budinsky et al., 2003]. Ofrece servicios para importar y exportar modelos utilizando XMI. Además, soporta importar modelos utilizando interfaces

[35] Disponible en `http://mdr.netbeans.org/metamodels.html`.

[36] Realmente puede ser considerado, como MOF, un meta-metamodelo. Permite definir construcciones para especificar metamodelos.

Java anotados. Ofrece mecanismos que permiten generar el código Java que implementa los modelos especificados. Si bien este código está generado en términos de interfaces e implementaciones, no es compatible con JMI.

Sobre dicho metamodelo se ha construido una implementación del metamodelo de UML 2.0 [Eclipse, 2006f].

3.3.6. Aportaciones y Carencias para la Investigación

MOF busca dar respuesta a una de las principales críticas realizadas a UML al intentar ser un lenguaje de modelado para definir todos los aspectos de un sistema. Como se ha visto, para que la promesa de MDA pueda convertirse en realidad es necesario poder definir lenguajes especializados que permitan especificar diferentes aspectos de los sistemas y hacerlo además para diferentes niveles de abstracción. MOF intenta cubrir esta necesidad.

Una arquitectura basada en un mecanismo de especificación de metamodelos común tiene varias ventajas importante, como se ha visto. Una consecuencia fundamental es la posibilidad de reutilizar diferentes servicios diseñados para trabajar con abstracciones MOF. Como MOF permite definir diferentes lenguajes, dichos servicios pueden reutilizarse con diferentes lenguajes sin necesidad de modificación.

No obstante hay que entender en toda su dimensión hasta qué punto permite MOF definir nuevos lenguajes. MOF permite describir únicamente los aspectos estructurales de dichos lenguajes. Define los lenguajes en base a las construcciones que permiten utilizar, esto es, su sintaxis abstracta. Esto permite la generación automática de servicios realmente útiles, como por ejemplo la serialización con XML (§ 3.3.4.1) o generación automática de los interfaces necesarios en un lenguaje de programación para manipular los metadatos (§ 3.3.4.3), pero en ningún caso especifica cómo se obtienen los metadatos a partir de la sintaxis concreta del lenguaje utilizado (si se utiliza alguna) o cómo deben ejecutarse los lenguajes definidos vía MOF. Esto significa que la definición de un nuevo lenguaje para MDA requiere un trabajo considerable por parte de su creador. Puede establecerse una similitud con el uso de XML. XML define una sintaxis estándar, al igual que MOF. Esto facilita la creación de analizadores para nuevos lenguajes basados en XML puesto que los analizadores léxico y sintáctico no varían y pueden reutilizarse. Sin embargo, el análisis semántico tiene que seguir realizándose como en cualquier otro procesador, y éste puede ser extremadamente complejo [Ortín Soler et al., 2004].

La representación de MOF como XML definida por XMI es muy interesante de cara a una herramienta MDA. XMI es, después de UML, el estándar relacionado con MDA con mayor soporte en la actualidad. La mayor parte de las herramientas UML actuales soportan la importación y exportación de modelos UML mediante XMI. Esto hace de XMI el candidato mejor posicionado para expresar los modelos que sirven de entrada a una herramienta de generación de código [Frankel, 2003]. Sin embargo, existe un problema relacionado con XMI relacionado con la sucesión de diferentes versiones de los estándares XMI, MOF y UML [Cook, 2004]. Las diferentes versiones de la arquitectura MOF y UML han introducido modificaciones

sustanciales en la estructura interna del lenguaje, por lo que las diferentes versiones de XMI no son compatibles entre sí. Además, se tratan de estándares realmente amplios y difíciles de implementar completamente. Por ello, existen diferentes herramientas que implementan diferentes versiones de MOF, y por ello, diferentes versiones de UML y XMI. Como a la hora de intercambiar modelos entre herramientas todas las combinaciones de versiones son posibles, la interoperabilidad que aspira conseguir XMI para intercambiar modelos se ve seriamente restringida[37].

[37]Por ejemplo, AndroMDA [AndroMDA, 2006] no puede trabajar con modelos UML 2.0 porque utiliza MDR (§ 3.3.5.1), el cual implementa la versión 1.4 de MOF y, sobre ella, la 1.5 de UML. Esto le impide importar modelos de las últimas versiones de la herramienta UML MagicDraw [Magic, 2006], que ya implementan UML 2.0.

Capítulo 4

TRANSFORMACIÓN DE MODELOS

Existen varias aproximaciones al problema de la transformación de modelos. Se comenzará analizando el estándar *Query/View/Transformation* (QVT) propuesto por el OMG y algunas de sus implementaciones. A continuación se analizarán otros estándares y soluciones específicas para este área del desarrollo dirigido por modelos. Finalmente se analizarán la problemática de la transformación de modelos en texto y de la generación de código.

4.1. Introducción

La transformación de modelos juega un papel clave en el desarrollo dirigido por modelos, puesto que serán un conjunto de transformaciones las que, partiendo de un conjunto de modelos que especifican un sistema, permitan conseguir el software ejecutable sobre una plataforma concreta. En este contexto, transformación de modelos hace referencia al proceso de convertir un modelo en otro modelo del mismo sistema [Hebach, 2005].

Utilizando la terminología de la iniciativa MDA, un proceso de transformación recibe como entrada un conjunto de modelos independientes de la plataforma (PIM) y un conjunto de reglas de transformación. Como producto de un proceso de transformación se obtiene un modelo específico a la plataforma (PSM). Esta estructura se denomina Patrón MDA y se recoge en la Figura 4.1 [Miller and Mukerji, 2003].

4.2. QVT

QVT es la propuesta del OMG para resolver el problema de la transformación de modelos. Se trata de un estándar para la definición de transformaciones sobre modelos MOF. El proceso de definición del estándar se inició con un RFP en 2002

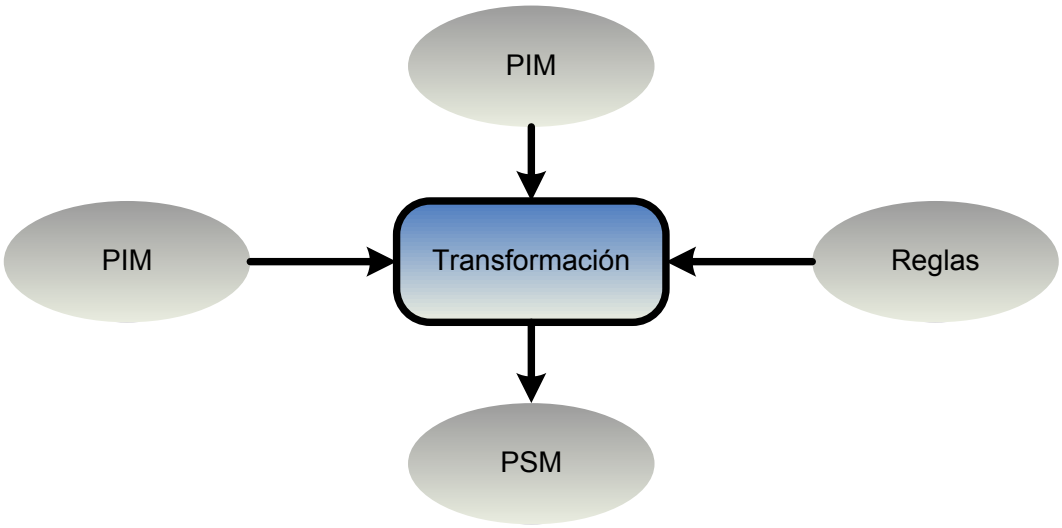

Figura 4.1: Patrón MDA

[OMG, 2002b] y culminó a finales de 2005 con la primera versión final de la especificación [OMG, 2005a], en la que convergieron las 8 propuestas inicialmente presentadas.

Cuando surgió la iniciativa MDA, los estándares UML, XMI y MOF ya gozaban de un cierto grado de madurez y fueron soportados, en mayor o menor medida, por parte de los fabricantes de herramientas. Sin embargo, las reglas de transformación que implementan dichas herramientas eran definidas en torno a un conjunto de tecnologías propietarias no estándares, tales como el uso de plantillas, la generación de metaclases específicas y el uso de otros sistemas propietarios. La consecuencia es que un elemento tan importante de MDA como es la realización de los sistemas depende totalmente de la herramienta utilizada. Este es el problema que QVT pretende solucionar.

El estándar QVT define tres abstracciones fundamentales, que se corresponden con sus siglas [Hebach, 2005]:

Consultas (*Queries*). Una consulta es una expresión que se evalúa sobre un modelo. Los resultados de una consulta son una o varias instancias de los tipos definidos en el modelo transformado, o en el propio lenguaje de consulta. Para la realización de consultas se utilizará un lenguaje de consultas.

Vistas (*Views*). Una vista es un modelo obtenido en su totalidad a partir de otro modelo base. Las consultas son un tipo restringido de vistas.

Transformaciones (*Transformations*). Una transformación genera un modelo a partir de otro modelo de origen. Ambos modelos podrán ser dependientes o independientes, según exista o no una relación que mantenga ambos sincronizados una vez se produzca la transformación. Las vistas son un tipo específico

de transformación. Si se modifica una vista, la correspondiente transformación debe ser bidireccional para reflejar los cambios en en el modelo fuente.

Las transformaciones se definirán utilizando un lenguaje de transformación. El lenguaje de transformación debe servir para generar vistas de un metamodelo, debe ser capaz de expresar toda la información necesaria para generar automáticamente la transformación de un modelo origen a uno destino, y debe además soportar cambios incrementales en un modelo origen que se ven reflejados en el modelo destino [OMG, 2002b].

De estas abstracciones se desprenden por lo tanto dos lenguajes, de consulta y de transformación, a los que se impuso como requisito que estuvieran definidos como metamodelos MOF 2.0 [OMG, 2002b]. Un último requisito fundamental de la propuesta QVT es relativo a los modelos manipulados: todos los modelos manipulados por los mecanismos de transformación serán además instancias de metamodelos MOF 2.0 (§ 3.3) [OMG, 2002b].

Una vez descrito el planteamiento conceptual de QVT conviene describir la solución propuesta por la especificación finalmente adoptada [OMG, 2005a]. Como lenguaje de consulta se definen extensiones al estándar OCL 2 (§ 3.1.5.1). Para la especificación de transformaciones se propone una solución de naturaleza híbrida declarativa e imperativa. La parte declarativa se divide en una arquitectura de 2 capas, que sirve como marco para la semántica de ejecución de la parte imperativa.

Las dos capas de la parte declarativa son:

***Relations*.** Define un lenguaje declarativa para expresar relaciones entre modelos MOF. Este lenguaje soporta reconocimiento de patrones , la creación de plantillas de objetos y la creación implícita de las trazas necesarias para registrar los cambios que se producen cuando se transforman modelos.

***Core*.** Define un lenguaje declarativo de menor nivel de abstracción que *Relations* pero con la misma potencia. La especificación de QVT define las reglas que permiten mapear la semántica de *Relations* a la de *Core*, y dichas reglas las define en base a transformaciones descritas utilizando a su vez *Core*. En este lenguaje, los objetos de traza deben definirse explícitamente y sólo soporta reconocimiento de patrones sobre un conjunto plano de variables, no sobre objetos complejos como en *Relations*.

En cuanto a la parte imperativa de QVT, se definen dos mecanismos para invocar implementaciones imperativas de transformaciones desde los lenguajes *Relations* o *Core*:

***Operational Mapping*.** Se trata de un lenguaje estándar que permite definir procedimientos imperativos de transformación. Dichos procedimientos deben emitir los mismos modelos de traza que el lenguaje *Relations*. Se trata de un lenguaje procedural, definido en torno a un conjunto de extensiones del lenguaje OCL que permiten efectos laterales, e incluye construcciones imperativas tales como bucles y condiciones.

Black-box MOF Operation. No se trata de un lenguaje, sino de un mecanismo que permite enlazar código escrito en cualquier lenguaje. Para ello, se utiliza el lenguaje *Relations* para derivar una operación MOF de una transformación. Esta operación será el punto de enlace con el lenguaje deseado, que deberá definir la operación a ejecutar con la misma signatura, y soportar un enlace con la implementación de MOF que se esté utilizando.

QVT define por lo tanto tres DSL: *Relations*, *Core* y *Operational Mappings*. Para todos ellos define una sintaxis abstracta en base a sus metamodelos MOF 2. La semántica de su ejecución se define mediante descripciones en texto plano. Para cada uno de ellos se proporciona una sintaxis textual concreta definida mediante gramáticas EBNF. Para el lenguaje *Relations* se define además una notación gráfica específica. Finalmente, para los lenguajes *Relations* y *Operational Mappings* se definen diversas extensiones de OCL 2. En la Figura 4.2 se puede observar las relaciones entre las principales abstracciones definidas en QVT.

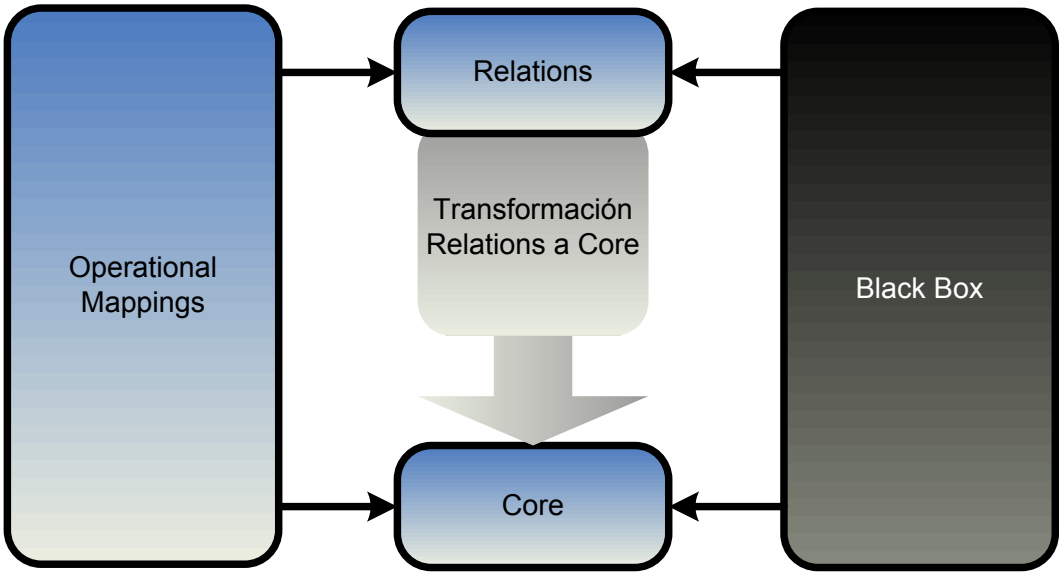

Figura 4.2: Relaciones entre los metamodelos de QVT

4.2.1. El Lenguaje Relations

El lenguaje *Relations* ofrece una aproximación declarativa para la especificación de transformaciones. Dado un par de modelos candidatos para la transformación, que deberán ser instancias de metamodelos MOF 2.0, ésta quedará definida como un conjunto de restricciones que los elementos de los modelos deben satisfacer.

Estas restricciones constituyen el concepto de *relación* del lenguaje *Relations*: se definen mediante dos o más dominios y mediante una pareja de predicados *when* (cuándo) y *where* (cómo). Los dominios especifican, mediante un patrón, qué elementos de los modelos candidatos participarán en la transformación. Por su parte, la

cláusula *when* especifica la relaciones que determinan cuándo la relación indicada debe sostenerse; y la cláusula *where* especifica la condición que deben satisfacer todos los elementos de modelos que participan en la relación [OMG, 2005a].

En el Listado 4.1 puede observarse un ejemplo de relación especificada mediante el lenguaje *Relations*. se define una relación `ClassToTable` que especifica las condiciones a satisfacer para transformar clases en tablas. Los modelos candidatos serán instancias de metamodelos MOF: el de UML (`uml`) y un metamodelo de un lenguaje de especificación de tablas relacionales (`rdbms`). Se definen dos dominios: el dominio `c` que especifica la clase de origen y el dominio `t` que especifica la clase de destino.

En el dominio `c` se especifica, mediante una plantilla, qué clases del modelo origen se utilizan en la transformación: serán todas aquellas contenidas en un paquete y cuyo nombre sea `cn`. Al no haber sido utilizada la variable `cn` previamente, servirá para crear una ligadura posterior. En el caso del dominio `t`, se define mediante un patrón como serán las tablas resultantes de la transformación. Estas tablas tendrán cuatro atributos: `schema`, `name`, `column` y `primaryKey`, cuyo contenido también es definido mediante los correspondientes patrones: el nombre de la tabla se corresponderá con el nombre de la clase y se creará una columna numérica que contendrá la clave primaria de la tabla. Por su parte, `PackageToSchema` y `AttributeToSchema` son a su vez relaciones que especifican, la concordancia de nombres en paquetes y esquemas de bases de datos, el primero, y la concordancia de atributos con columnas, el segundo.

Listado 4.1: Ejemplo de relación expresada mediante el lenguaje Relations

```
relation ClassToTable /* mapea cada clase a una tabla */{
  domain uml c:Class {
    namespace = p:Package {},
    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);
  }
}
```

4.2.2. El Lenguaje Operational Mapping

El lenguaje *Operational Mapping* permite definir transformaciones utilizando bien una aproximación imperativa o bien una aproximación híbrida, complementando las transformaciones relacionales declarativas con operaciones imperativas que implementen las relaciones.

Una transformación operacional define una transformación unidireccional expresada de forma imperativa. Se compone de una signatura que indica los modelos involucrados en la transformación y de una definición de la ejecución de la transformación. En la práctica, una transformación operacional se trata de una entidad instanciable con propiedades y operaciones.

En el Listado 4.2 puede observarse un ejemplo de transformación definida mediante el lenguaje *Operational Mapping*. Se define la signatura de la transformación, indicando que recibe como entrada una instancia de `UML` y de salida una instancia del metamodelo `RDBMS`. Ambos son metamodelos definidos mediante MOF 2. En punto de entrada de la transformación se define en la función `main()`. En el ejemplo, la transformación se define invocando la operación de mapeo `packageToSchema()` sobre todos los elementos del tipo `Package` del modelo UML.

A continuación se define la operación `packageToSchema` de forma imperativa. Dicha operación recibe como entrada un paquete UML (`Package`) y devuelve como salida un esquema relacional (`Scheme`). Especifica que en el objeto esquema de salida tendrá el mismo nombre que el paquete, y que las tablas del esquema se corresponderán con el resultado de invocar la operación de mapeo `class2table` sobre las clases del paquete.

Listado 4.2: Ejemplo de transformación expresada mediante el lenguaje Operational Mapping

```
transformation Uml2Rdbms(in uml:UML,out rdbms:RDBMS) {
  main() {
    uml.objectsOfType(Package)->map packageToSchema();
  }

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

4.2.3. Implementaciones de QVT

Debido al poco tiempo que ha transcurrido desde que el OMG liberase la primera especificación final de QVT [OMG, 2005d] no existe disponible ninguna implementación completa del estándar.

4.2.3.1. Borland Together Architect 2006

Dentro de las herramientas comerciales que soportan parcialmente QVT se encuentra Borland Together Archiect2006 [Borland, 2006]. La herramienta implementa un motor QVT basado en la propuesta de especificación QVT de marzo de 2005 [OMG, 2005f]. No soporta el lenguaje declarativo de *Relations*.

Sólo soporta las transformaciones operacionales (imperativas). Soporta además OCL en su versión 2, y lo extiende siguiendo las directrices de QVT. En cuanto a las operaciones de caja negra (*Black-box MOF Operation*), la herramienta permite importar e invocar sentencias Java desde las transformaciones QVT.

Para las transformaciones de modelos a texto (*text-to-model*) la herramienta utiliza el motor de plantillas *Java Emitter Templates* (JET) [Popma, 2003], del proyecto Eclipse [Eclipse, 2006c], el cuál se basa en EMF § 3.3.5.2.

4.2.3.2. SmartQVT

SmartQVT [Belaunde and Dupe, 2006] es una implementación de código abierto de la especificación QVT realizada por France Telecom R&D. En el momento de escribir estas líneas sus autores están trabajando para implementar la versión final de la especificación [OMG, 2005d]. Al igual que Together Architect, se basa en metamodelos EMF para los modelos que intervienen en las transformaciones y únicamente considera el lenguaje *Operational Mapping*.

También utiliza EMF para especificar el metamodelo de QVT. Su funcionamiento se basa en una transformación en dos fases:

- En una fase inicial se traduce la sintaxis textual concreta del lenguaje operacional de QVT a instancias del metamodelo de QVT.
- En una segunda fase, se transforma el modelo de QVT en un programa Java basado en EMF que ejecuta la transformación. Para serializar el modelo QVT de entrada se utiliza XMI 2.0 § 3.3.4.1.

4.2.3.3. MOMENT

MOdel manageMENT (MOMENT) es un prototipo de motor de transformación que implementa el lenguaje *Relations* de la especificación QVT (§ 4.2.1). En su implementación utiliza EMF para la especificación de metamodelos. A partir del código del lenguaje *Relations*, instancia el metamodelo QVT que implementa la herramienta. Como paso previo a obtener el modelo transformado en términos de EMF, se genera una especificación intermedia en el lenguaje Maude [Clavel et al., 2003],

aprovechándose de sus propiedades intrínsecas, como la concordancia de patrones, la parametrización y la reflexión.

4.2.4. QVT y Otras Aproximaciones para la Transformación de Modelos

Paralelamente al desarrollo de QVT han aparecido otras aproximaciones para la transformación de modelos. Entre éstas destacan las propuestas por VIATRA § 4.3 y por ATL § 4.4, que actualmente presentan diferentes niveles de conformidad con QVT.

En este sentido, gracias a la separación que se realiza en la especificación de QVT de sintaxis abstracta, definida formalmente en base a un metamodelo, y concreta, es posible hacer evolucionar las sintaxis concretas de otros lenguajes de transformación para que se basen en la sintaxis abstracta del metamodelo de QVT.

4.3. VIATRA

VIsual Automated model TRAnsformations (VIATRA) 2 es un framework de propósito general para dar soporte a todo el ciclo de vida de los procesos de transformación de modelos. Desarrollado desde 1998 en la Universidad de Tecnología y Económicas de Budapest, ha sido utilizado en diversos proyectos de ámbito europeo en el ámbito de los sistemas empotrados [VIATRA, 2006].

VIATRA se define en torno a un conjunto de especificaciones propias, al margen de las planteadas por el OMG. Define un lenguaje de metamodelado propio denominado *Visual and Precise Metamodeling* (VPM) [Varro and Pataricza, 2003]. El metamodelo definido es mucho más sencillo que MOF, definiendo dos entidades básicas: `Entity` y `Relation`. Las relaciones pueden ser agregaciones y pueden tener multiplicidades. Además se definen dos relaciones especiales, la generalización y la relación "ser instancia de". La compatibilidad con otros lenguajes de metamodelado, como MOF § 3.3, se plantea utilizando *plugins* de conversión.

Al igual que MOF, VPM se define en base a un metamodelo formal que constituye su sintaxis abstracta. Se definen dos sintaxis concretas sobre ella, una gráfica similar a UML y una textual denominada *Viatra Textual (Meta)Modeling Language* (VTML).

En cuanto a las transformaciones, además de las transformaciones modelo a modelo, VIATRA considera las transformaciones modelo a texto. Para las transformaciones entre modelos se utilizan patrones de grafos para definir restricciones y condiciones en los modelos. Para especificar las transformaciones se utilizan dos tipos de reglas:

Reglas de transformación de grafos: para definir manipulaciones elementales de modelos utilizando la técnica de transformación de grafos [Ehrig et al., 1999].

Máquinas abstractas de estados: Para la descripción de estructuras de control .

Para la especificación de ambos tipos de reglas se ha creado un lenguaje denominado *Viatra Textual Command Language* (VTCL) que es textual y tiene como objeto servir de base para la construcción de editores gráficos que se edifiquen sobre él. VTCL incluye algunas construcciones básicas del lenguaje *Abstract State Machine* (ASM) [Börger and Stärk, 2003] para las construcciones típicas de un lenguaje imperativo.

En cuanto a las transformaciones de modelos a texto se utiliza un lenguaje de plantillas propio denominado *Viatra Textual Template Language* (VTTL), similar al motor Velocity del proyecto Apache Jakarta [Apache, 2006b] pero sustituyendo Java por el lenguaje ASM.

4.4. ATL

ATLAS Transformation Language (ATL) [INRIA Nantes, 2006] es la respuesta dada por el instituto de investigación *Institut national de recherche en informatique et en automatique* (INRIA) a la propuesta de *Request for Proposal* (RFP) para QVT lanzada por el OMG en 2002 [OMG, 2002b]. Actualmente se encuadra dentro del proyecto *Generative Modeling Technologies* (GMT) de la Fundación Eclipse [Eclipse, 2006b], que es la rama de proyectos de investigación abierta en el seno del proyecto *Eclipse Modeling Project* [Eclipse, 2006d].

ATL define un lenguaje de transformación de modelos especificado tanto por su metamodelo como por una sintaxis concreta textual. Al igual que QVT tiene una naturaleza híbrida entre declarativa e imperativa. Para la manipulación de modelos se basa en MOF (§ 3.3).

Como parte del proyecto se ha definido una máquina virtual orientada a la transformación de modelos, con el fin de aumentar la flexibilidad de la arquitectura [INRIA Nantes, 2005]. Las transformaciones especificadas en ATL son transformadas a un conjunto de primitivas básicas de transformación de la máquina virtual. De este modo, las transformaciones ATL son ejecutables porque existe una transformación específica del metamodelo ATL al bytecode de esta máquina virtual. Además, puede extenderse el lenguaje traduciendo las nuevas construcciones al bytecode de la máquina virtual.

La sintaxis concreta de ATL es similar a la definida por QVT, aunque ambos lenguajes no son interoperables por las divergencias en sus metamodelos. En [Jouault and Kurtev, 2006] se encuentra un análisis de la alineación arquitectónica de ambos lenguajes.

4.5. Motor de Transformaciones de BOA 2

El framework BOA 2 es la evolución del framework BOA [Padrón Lorenzo, 2004], una propuesta de framework MDA desarrollado por la empresa Open Canarias. Mientras que en la versión 1 del framework se realizaba una única operación de transformación basada en la traducción de modelos UML serializados mediante

XMI utilizando plantillas *Extensible Stylesheet Language Transformations* (XSLT), en la versión 2 se ha planteado reescribir el motor de transformación implementando la especificación QVT [Padrón Lorenzo et al., 2005].

El motor de transformación de BOA 2 se centra en la transformación modelo a modelo manipulando modelos EMF § 3.3.5.2. En la presentación del framework [Padrón Lorenzo et al., 2005] se plantea una implementación de un motor QVT con soporte para características avanzadas como trazabilidad, compilación incremental, transaccionalidad de las transformaciones y bidireccionalidad. Para implementar la bidireccionalidad, propone utilizar observadores [Gamma et al., 1995] EMF para registrar la información de cambio de los modelos, puesto que existen transformaciones inherentemente unidireccionales en las que se pierde información tras su ejecución.

Con el fin de independizar el motor de transformación de los lenguajes de transformación que soporte el framework BOA 2 propone un lenguaje de transformaciones de bajo nivel denominado *Atomic Transformation Code* (ATC). Se trata de un lenguaje imperativo neutro de bajo nivel firmemente alineado con EMF. Se prevé una arquitectura basada en máquina virtual que, tras la estabilización del lenguaje, evolucione a una solución donde se transforme el código ATC en código Java por razones de eficiencia. Sobre ATC, se prevé dar soporte a los lenguajes *Relational* y *Operational Mapping* de QVT.

Para la generación final de código de las aplicaciones a partir de los modelos transformados el framework BOA 2 utiliza el motor de plantillas JET [Popma, 2003].

4.6. M2M

Model-to-Model Transformation (M2M) es un proyecto Open Source encuadrado dentro del grupo de proyectos de la iniciativa *Eclipse Modeling Project* de la fundación Eclipse [Eclipse, 2006d]. En su estado actual, es una propuesta de proyecto aceptada en espera de las contribuciones de las compañías Borland, Compuware e INRIA para comenzar su desarrollo.

El planteamiento del proyecto es desarrollar un *framework* de lenguajes de transformación de modelo a modelo. Busca ofrecer una arquitectura modular donde puedan configurarse diferentes motores de transformación. Está previsto implementar 3 motores de transformación: para ATL (§ 4.4), para el lenguaje *Operational Mapping* de QVT § 4.2.2 y para los lenguajes *Core* y *Relational* de QVT § 4.2.1.

El proyecto busca por lo tanto salvar las diferencias existentes entre QVT y ATL a la hora de ejecutar transformaciones elevando el nivel de abstracción. Desde el punto de vista de esta investigación, una aportación clave de este proyecto será la de proporcionar una implementación open source del estándar QVT. Esta apuesta responde al propósito perseguido por el *Eclipse Modeling Project* de apostar fuertemente por los estándares existentes, si bien el único motor de transformación implementado hasta la fecha se basa en ATL.

4.7. Sistemas de Generación de Código

En los apartados anteriores se han estudiado diferentes propuestas y tecnologías de transformación de modelos. Todos ellos se centran en la transformación que el patrón MDA denomina de PIM a PSM (§ 4.1), es decir, transformaciones de modelo a modelo. Para completar el ciclo de vida de un proceso de transformación que obtenga como producto una aplicación ejecutable es necesario ejecutar una transformación de modelo a texto. Es decir, a partir de los modelos transformados es necesario obtener el código fuente final que se ejecutará sobre la plataforma de destino.

La forma más conocida de generador de código son los compiladores tradicionales [Cueva Lovelle, 1998] que reciben como entrada el código fuente creado por el desarrollador. Tal y como se vio en el Capítulo 2, lo que se plantea en el desarrollo dirigido por modelos es elevar el nivel de abstracción respecto al modelo de desarrollo tradicional, siendo los artefactos de primer nivel que manejarán los desarrolladores modelos. El código fuente, junto con todos los descriptores y ficheros de configuración necesarios para desplegar la aplicación sobre la plataforma de destino, será ahora el producto obtenido a partir de los modelos manejados por los desarrolladores. Se eleva el nivel de abstracción del código generado puesto que éste se edificará sobre la plataforma tecnológica utilizada, cuya existencia responde a su vez a una necesidad de acercar los artefactos manejados al desarrollador humano.

Para enumerar los tipos de generadores de código existentes se utilizará una variante de la clasificación que realiza Jack Herrington en [Herrington, 2003], prescindiendo de la generación de clases parciales y de la generación de capas en una arquitectura de n-capas, que se considerarán un caso particular de los anteriores:

Code munging[1]**.** Este tipo de generadores de código toma determinados aspectos del código de entrada y genera uno o más ficheros de salida. Un ejemplo típico son los generadores de documentación a partir de los comentarios del código fuente de un programa.

Expansor de código en línea (*inline-code expander*). . Este tipo de generadores toma como entrada código fuente que contienen algún tipo especial de marcado que el expansor reemplaza con código de producción cuando crea los ficheros de salida. Un ejemplo son los lenguajes incrustados dentro de un lenguaje anfitrión. Dentro de esta ámbito se encuentran los sistemas que permiten incrustar SQL en un lenguaje de propósito general, como SQLj [Clossman et al., 1998].

Un tipo particular de generador de código dentro de esta categoría son los motores de plantillas (*template engine*). En este caso, los ficheros de entrada del generador son plantillas que combinan contenido estático que es enviado directamente al fichero de salida con contenido dinámico que es interpretado en un proceso específico. El resultado de esta interpretación puede tener un propósito general, aunque suele orientarse a la generación dinámica de código en función de los metadatos que guían el proceso de generación. Por su importancia, se analizarán en § 4.7.1.

Generación de código mezclado (*mixed-code generation*) Se trata de una variante del anterior modelo en el cual el fichero de código de salida sobreescribe al fichero de origen. En [Herrington, 2003] se incluye un ejemplo de generador de este tipo que permite generar el código de tests unitarios en programas escritos en C.

Tal y como se ha analizado en los apartados previos, los procesos de transformación son los suficientemente complejos como para desaconsejar su acometida en una única etapa. Es decir, el planteamiento correcto no es tomar los modelos que sirven de entrada para la aplicación y generar todo el código necesario a partir de ellos. Es necesario una etapa media de transformación modelo a modelo. Esta etapa se justifica por la necesidad de realizar transformaciones entre diferentes niveles de abstracción (por ejemplo, entre el ámbito de la programación orientada a objetos y el ámbito de una plataforma tecnológica), así como entre diferentes dominios (por ejemplo, entre el mundo de los objetos y el mundo relacional).

Así pues, la entrada de la herramienta generadora de código no serán los modelos de análisis inicialmente elaborados. Estos deberán ser sometidos a sucesivas transformaciones que desemboquen en modelos detallados para cada dominio y entorno tecnológico asociado en la aplicación final. Así pues, los modelos que finalmente recibirán los generadores de código tendrán el suficiente nivel de detalle como para no exigir una lógica de generación compleja. Esta es la razón por la que el estándar QVT no contempla la traducción modelo a texto (§ 4.2). Las implementaciones analizadas de QVT (§ 4.2.3), VIATRA (§ 4.3) y ATL (§ 4.4) utilizan diferentes sistemas de generación de código, todos ellos basados en motores de plantillas. La idoneidad de esta tecnología de generación se justifica en su capacidad para definir esqueletos de código parametrizables, sin limitar la lógica de aplicación que puede ser necesaria para decisiones complejas durante la generación.

4.7.1. Motores de plantillas

Un motor o procesador de plantillas es un programa que permite combinar una o varias plantillas con un modelo de datos para obtener uno o más documentos de salida [Wikipedia, 2006].

Un motor de plantillas recibe como entradas:

- Un modelo de datos, que constituirá el contexto de información accesible desde la plantilla.

- Las plantillas, que constituirán la fuente de la traducción. Harán uso de la información contenido modelo de datos para su posterior procesamiento por el motor. Este uso puede consistir simplemente en referenciar parámetros del modelo de datos que se sustituirán por su valor cuando la plantilla se procese; o puede ser un uso más complejo, incluyendo el control del flujo de generación basado en la información del modelo.

Los motores de plantillas han sido usados profusamente para la generación dinámica de código en el ámbito de las aplicaciones Web. Se han desarrollado diversos motores de scripting que se ejecutan en el lado del servidor y que permiten generar dinámicamente el código que se sirve al cliente que accede con un navegador Web. En este área se encuentran *Java Server Pages* (JSP) para Java EE [Bergsten, 2003]; *Active Server Pages* (ASP) dentro de ASP.NET [Thai and Lam, 2002] o el motor de PHP [Tatroe et al., 2006]. El fundamento de todas estas tecnologías se basa en poder introducir código escrito en los lenguajes soportados por la plataforma correspondiente mediante secuencias de escape dentro de las plantillas que contienen el código estático, típicamente HTML.

Dentro de los motores de plantillas de propósito general destacan Velocity dentro del proyecto Apache Jakarta [Apache, 2006b] y JET de la Fundación Eclipse [Popma, 2003]. Éste último ha sido utilizado para completar el ciclo de vida de transformación de algunas herramientas analizadas en los apartados anteriores (ATL en § 4.4, QVT en § 4.2.3). Otras herramientas optan por desarrollar un motor de plantillas propio específico para sus necesidades. Este es el caso de VTTL en VIATRA (§ 4.3) o XPand en la herramienta *openArchitectureWare* (§ 5.2).

El OMG publicó en 2004 una RFP para aceptar proposiciones con el objeto de estandarizar un sistema de traducción de modelos a texto bajo la denominación de *MOF Model to Text Transformation Language* [OMG, 2004]. En la actualidad se está trabajando en la versión final de la especificación a adoptar [OMG, 2006a]. La propuesta se basa en una aproximación basada en plantillas. Las plantillas permiten secuencias de escape que permiten extraer datos de los modelos. Estas secuencias consisten en expresiones especificadas sobre los elementos de metamodelado. Utiliza consultas OCL como el mecanismo principal para seleccionar y extraer información de los modelos.

Dentro de la iniciativa *Eclipse Modeling Project* [Eclipse, 2006d] existe una propuesta de proyecto open source denominada *Model-to-Text Transformation* (M2T). Mientras que el proyecto M2M (§ 4.6) se centra en las transformaciones entre modelos, M2T se centra en tecnologías para transformar modelos en texto. Del forma similar a M2M, M2T busca situarse por encima de diversos motores de plantillas existentes, permitiendo configurar diferentes aproximaciones a través de un mismo framework. Los motores de plantillas que está previsto integrar en el framework son JET [Popma, 2003], XPand (§ 5.2) y OMG MOF Model to Text [OMG, 2006a].

4.8. Aportaciones y Carencias para la Investigación

En este capítulo se han analizado diversas aproximaciones y estándares para la consecución del ciclo de vida completo de un proceso de transformación. La transformación de modelos se muestra como un componente clave de una herramienta MDD. En el Capítulo 3 se analizó el problema de cómo especificar modelos formalmente y permitir su manipulación. El cómo someter dichos modelos a sucesivas transformaciones automáticas para obtener la aplicación final ejecutable aparece co-

mo un segundo elemento clave de análisis del paradigma de desarrollo dirigido por modelos.

En primer lugar cabe resaltar la importancia y la necesidad de estandarizar los componentes de transformación de las herramientas MDD. El utilizar estándares para la especificación de los modelos es clave para que el desarrollador no quede ligado a una herramienta de modelado concreta. Pero de cara a una herramienta MDD, los modelos carecen de valor sin la información detallada acerca de cómo transformarlos. Por lo tanto, si los mecanismos de especificación de transformaciones dependen de cada herramienta, sigue sin conseguirse la independencia de la herramienta utilizada.

El estándar que se erige como la referencia en el momento actual es QVT, que tiene detrás al OMG, aunque aún es pronto para determinar si este estándar va a ser finalmente adoptado por los desarrolladores de herramientas. En primer lugar presenta ciertos problemas comunes a todos los estándares definidos por un comité para ámbitos de la industria dónde no había experiencia previa. Se trata de un estándar amplio y complejo que a día de hoy no tiene una implementación funcional completa. Esto hace imposible predecir cual será su impacto real en la industria, aunque cómo se ha visto existen o se prevén desarrollar implementaciones para todos los lenguajes que define: *Operational Mapping* (§ 4.2.3.1, § 4.6, § 4.2.3.2) y *Relations* (§ 4.6, § 4.2.3.3). Por otra parte, los lenguajes de transformación definidos en QVT hacen uso intensivo de OCL, lenguaje que no está ampliamente extendido en la comunidad de desarrolladoras y que tiene un pobre soporte por parte de las herramientas.

El lenguaje QVT *Relations* se trata de un lenguaje declarativo, basado en patrones y que permite la transformación bidireccional de modelos, pues únicamente especifica las reglas que los modelos transformados deben cumplir. Este lenguaje se muestra apropiado para ámbitos dónde los modelos transformados sean semánticamente equivalentes, como por ejemplo, un modelo de objetos y un modelo relacional asociado.

Sin embargo, en un proceso de transformación real serán habituales las transformaciones que exijan sucesivos refinamientos y saltos de abstracción en los que necesariamente se perderá información. Éstas transformaciones serán unidireccionales por su naturaleza y exigirán un lenguaje de transformación imperativo que QVT resuelve con la propuesta de *Operational Mapping* (§ 4.2.2). En este último caso una dificultad asociada es que se define un nuevo lenguaje imperativo que, siendo similar a los lenguajes de objetos más utilizados en la actualidad, exigirá a los desarrolladores su aprendizaje y a los desarrolladores de herramientas su soporte. Ambos factores deben tenerse en cuenta a la hora de predecir posibles problemas en su adopción por la comunidad.

VIATRA (§ 4.3) y ATL (§ 4.4) presentan la ventaja de ofrecer implementaciones funcionales de sus especificaciones. VIATRA presenta un metamodelo propio y un conjunto de lenguajes específicos para la manipulación de las transformaciones. ATL se basa en metamodelos MOF, aunque también presenta un lenguaje de transformación imperativo propio basado en una sintaxis abstracta propia de ATL. Esto

hace que ambos lenguajes sean incompatibles con QVT.

Una solución pragmática al problema de la incompatibilidad de los sistemas de transformación de modelos viene dada por M2M (§ 4.6). Aunque es todavía un proyecto propuesto que debe desarrollarse plantea un framework en el que pueden configurarse diferentes estrategias de transformación ofreciendo al desarrollador una vista unificada.

En cuanto a las implementaciones de los motores de transformación, destaca el enfoque utilizado por ATL (§ 4.4) y por el motor del framework BOA 2 (§ 4.5) basado en máquinas virtuales y lenguajes intermedios. Esta aproximación se fundamenta en crear un lenguaje de transformación de bajo nivel. Los lenguajes concretos de transformación utilizados se traducirán a este lenguaje intermedio, que será posteriormente ejecutado sobre una máquina virtual. Se trata de otra solución al problema de la multiplicidad de lenguajes de transformación distinta a la utilizada por M2M (§ 4.6).

Sin duda se trata de una propuesta mucho más compleja de implementar, en cuanto a que introducir soporte para nuevos lenguajes implica construir nuevos compiladores al código intermedio, pero también más flexible. Un problema que puede plantearse es la adecuación del código intermedio a cualquier lenguaje de transformación. En la propuesta de BOA 2 se plantea dar soporte a las variantes imperativa y declarativa de QVT, sin embargo se señala la dificultad de construir un compilador para el lenguaje declarativo [Padrón Lorenzo et al., 2005]. En este sentido, la implementación del lenguaje QVT *Relations* realizada por MOMENT (§ 4.2.3.3), se basa en utilizar el lenguaje Maude [Clavel et al., 2003] por sus características intrínsecas como la la concordancia de patrones, la parametrización y la reflexión.

QVT, VIATRA y ATL definen un metamodelo específico para sus lenguajes de transformación. Esto permite definir un número arbitrario de sintaxis concretas sobre dichos lenguajes. Esta característica puede ser aprovechada para construir implementaciones de los mismos utilizando lenguajes ampliamente extendidos en la comunidad de desarrollo. Ninguno de los sistemas analizados utiliza este enfoque.

Todos los sistemas analizados presentan un enfoque de transformación modelo a modelo. La complejidad de la transformación se absorbe realizando sucesivas transformaciones entre modelos, a diferente nivel de abstracción y pertenecientes a diferentes dominios. Ningún sistema plantea la transformación de modelos en una única etapa. La transformación de modelos en texto se muestra como otro elemento de análisis que se corresponde con la última etapa de un proceso de transformación. Como se ha visto en § 4.7, existen diferentes aproximaciones para la generación de código.

El enfoque más utilizado para el desarrollo dirigido por modelos es el basado en motores de plantillas. QVT no aborda este área, para la cual el OMG está desarrollando un estándar específico: MOF Model to Text Transformation Language § 4.7.1. De nuevo es difícil predecir cuál será el impacto de esta propuesta en la industria. Aunque se trata de un área donde sí existe experiencia previa, existen motores de plantillas con una amplia aceptación en la comunidad, como Velocity

[Apache, 2006b] y JET [Popma, 2003]. Las implementaciones de QVT (§ 4.2, § 4.5) (§ 4.2.3) y ATL (§ 4.4) se basan en éstas implementaciones de motores de plantillas. VIATRA por su parte opta por desarrollar un lenguaje de plantillas específico: VTCL (§ 4.3).

El proyecto M2T, de forma similar a M2M, plantea resolver el problema de la multiplicidad de motores de plantillas ofreciendo un framework que permita configurar diferentes motores ofreciendo una única vista al desarrollador.

Los paradigmas de la especificación de modelos y de la transformación de modelos conforman los cimientos de una estrategia MDD. Las especificaciones y estándares de transformación sirven como base para la reutilización de software. Un ejemplo son los repositorios de transformaciones, que permiten al desarrollador acceder a definiciones de transformaciones previamente desarrolladas. Por ejemplo, ATL ofrece una lista de definiciones de transformación que en el momento de escribir estas líneas cuenta con 73 definiciones [Eclipse, 2006a].

Capítulo 5

ANÁLISIS DE SISTEMAS DE DESARROLLO DIRIGIDO POR MODELOS

5.1. AndroMDA

AndroMDA [AndroMDA, 2006] es un framework MDA de generación de código. El líder del proyecto es Matthias Bohlen siendo 15 los componentes del núcleo de desarrollo del proyecto en la actualidad. El proyecto se deriva de la herramienta UML2EJB creada por Matthias Bohlen en 2002 [Bohlen, 2002].

En la actualidad el la arquitectura de la herramienta se encuentra sometida a una profunda revisión que verá la luz en su versión 4. En esta versión se plantean cambios que faciliten la evolución futura de la herramienta, debido a ciertas limitaciones que presenta la herramienta en su versión 3[1]. Dado que la versión actual de la herramienta goza de una gran implantación tanto en la comunidad como en la industria de desarrollo de software, se comenzará analizando sus fundamentos. A continuación se describirán cuáles son los cambios arquitectónicos previstos para la versión 4.

5.1.1. AndroMDA versión 3

AndroMDA en su versión 3 es una herrramienta generadora de código a partir de modelos UML[2]. La herramienta es capaz de generar código para las plataformas

[1] En el momento de escribir estas líneas la última versión estable de la herramienta es la 3.2

[2] Realmente AndroMDA permite aceptar modelos correspondientes con metamodelos distintos de UML pero todos los plugins desarrollados hasta la fecha usan modelos UML [Architecture, 2006].

Java EE y .NET. Dentro de cada una de ellas, permite generar código para múltiples frameworks y tecnologías ampliamente utilizadas en la industria.

Para la manipulación de los modelos introducidos AndroMDA utilizaba la implementación de MOF MDR (§ 3.3.5.1) leyendo los modelos de entrada en formato XMI (§ 3.3.4.1). Esto restringía el abanico de herramientas UML a utilizar pues MDR implementa la versión 1.4 de MOF y las herramientas de modelado actuales soportan MOF 2.0. Esta situación se ha solucionado en las últimas versiones introduciendo una fachada (*facade*) ([Gamma et al., 1995]) que sirve como punto de acceso a diferentes repositorios de metamodelado. De este modo, se da soporte también a metamodelos EMF (§ 3.3.5.2) desde la versión 3.2.

Para la tarea de generar código AndroMDA presenta una arquitectura modular basada en el concepto de cartucho (*cartridge*) [AndroMDA, 2006]. Los cartuchos son un tipo especial de plugin que puede configurarse en la herramienta y en el que se delega la tarea de generar código a partir de los elementos los modelos de entrada. Los cartuchos especifican qué elementos del modelo serán procesados y los asocian a determinadas plantillas que determinan qué código será generado a partir de ellos. Para seleccionar dichos elementos se utilizan estereotipos UML [**?**] así como condiciones que pueden inferirse del modelo en función de las propiedades de sus elementos.

Para el acceso a los elementos del metamodelo AndroMDA propone utilizar el patrón Fachada [Gamma et al., 1995]. A las fachadas de los elementos del metamodelo los denomina *metafacades* [AndroMDA, 2006]. Estas fachadas encapsulan los detalles de la implementación del metamodelo utilizado y ofrecen una API orientada a objetos que permiten acceder a sus elementos desde las plantillas. Ésto permite centralizar la inteligencia y la responsabilidad de la generación de código en las fachadas, no el lenguaje del motor de plantillas.

En cuanto a los motores de plantillas utilizados AndroMDA Velocity [Apache, 2006b], si bien soporta un número arbitrario de motores de plantillas permitiendo configurar cualquiera que implemente un determinado interfaz. En la versión 3.2 se ha añadido soporte para el motor Freemarker [FreeMarker, 2006].

Un problema que surge con la generación de código en AndroMDA es la necesidad de modificar el código una vez este ha sido generado. Los desarrolladores de cartuchos han adoptado diferentes aproximaciones para resolver este problema. En algunos casos se generan automáticamente clases hijas de las generadas en las el desarrollador puede definir métodos para extender su funcionalidad. Otra alternativa es permitir al desarrollador reproducir manualmente la estructura de archivos generados en un directorio específico donde puede modificarlos a su gusto. Los archivos modificados sobreescriben a los generados a la hora de generar la aplicación. Ambas aproximaciones presentan serios inconvenientes. La primera, porque la redefinición de métodos limita mucho las posibilidades de extender la funcionalidad de la aplicación. La segunda, porque recae en el desarrollador la responsabilidad de mantener sincronizados los modelos de entrada y los artefactos de código generados.

Una de las principales virtudes de AndroMDA ha sido el pragmatismo que ha

guiado el desarrollo del proyecto. Se han desarrollado cartuchos para generar código basado en los frameworks y tecnologías más extendidos y aceptados en la comunidad. De este modo, se han creado cartuchos para Spring [Johnson et al., 2005], Hibernate [Bauer and King, 2004], Struts [Apache, 2006a] y *Java Server Faces* (JSF) [Mann, 2005].

Otra característica destacada de AndroMDA es que, en la generación de código, los cartuchos hacen uso de las mejores prácticas y patrones de diseño. Es decir, aunque el código se genera automáticamente, se busca que sea un código de calidad, que conforme una aplicación estructurada fácilmente comprensible por un desarrollador humano. Como ejemplo, pueden señalarse el uso de los patrones *Data Access Object* (DAO) y *Value Object* [Alur et al., 2003] en el código generado por los cartuchos de Spring e Hibernate.

Desde el punto de vista de una herramienta de desarrollo dirigido por modelos AndroMDA presenta, en su versión 3, ciertas carencias. Por un lado, se trata de una herramienta que contempla exclusivamente transformaciones de modelos a texto. Como se vio en el Capítulo 4, este enfoque dificulta la acometida de transformaciones complejas, para lo que se requieren múltiples transformaciones modelo a modelo.

Por otro lado, todos los cartuchos desarrollados hasta el momento se basan exclusivamente en modelos UML. En las áreas donde UML permite crear modelos con toda la información necesaria para su transformación a otros dominios es donde AndroMDA se muestra más útil. Un ejemplo son los modelos estáticos de clases (§ 3.1.4.1.1) que son transformados a tablas en el modelo relacional. A partir de los diagramas estáticos de clases, AndroMDA permite generar todo el código necesario para gestionar la persistencia de las entidades: los ficheros con la información de mapeo objeto-relacional de Hibernate; las clases Java que se corresponden con las entidades persistentes, que serán objetos POJO [Fowler, 2000] con métodos de acceso a las propiedades de la clase y la implementación de los métodos `equals()` y `hashCode()`; así como las clases DAO que implementan operaciones *Create, read, update and delete* (CRUD) para los objetos persistentes asociados a cada clase.

Sin embargo, existen muchas áreas donde los modelos UML no permiten especificar la información de entrada. Un ejemplo es la especificación de interfaces de usuario. En el cartucho de Struts [AndroMDA, 2006] de AndroMDA, a partir de diagramas de actividad UML (§ 3.1.4.2.2) se puede especificar el comportamiento de la capa de presentación de una aplicación Web. Los diagramas de actividad se muestran adecuados para modelar el flujo de información entre pantallas, así como para indicar qué controladores serán los responsables de gestionar las peticiones realizadas por el usuario. A partir de los diagramas, AndroMDA permite generar toda la información de configuración necesaria para Struts, las vistas JSP y los controladores Java asociados. Se generan además determinadas clases donde el desarrollador puede introducir código manualmente. Sin embargo, si se desea personalizar el aspecto de la aplicación, la única alternativa disponible es copiar el código generado un directorio y modificarlo. De este modo, se está rompiendo el ciclo de vida de la transformación automática, puesto que el código del interfaz deja

de ser generado si el desarrollador lo modifica. Además, como esta modificación es manual, es fácil llegar a una situación donde los modelos y los artefactos de código generados dejen de estar sincronizados.

5.1.2. AndroMDA versión 4

La arquitectura de AndroMDA se encuentra en la actualidad en un proceso de profunda revisión. El objetivo es acometer una serie de cambios que verán la luz con la versión 4 de la herramienta [Bohlen, 2006]. Los principales objetivos que motivan esta revisión son:

- Soportar un número arbitrario de metamodelos, para que los lenguajes utilizados para construir los modelos de entrada no estén obligados a basarse en el metamodelo de UML.
- Soporte para procesos de transformación modelo a modelo. Como se vio en § 5.1.1 actualmente la herramienta sólo soporta un enfoque modelo a texto. AndroMDA hará uso de ATL (§ 4.4) para implementar el motor de transformación, aunque permitirá configurar un número arbitrario de motores de transformación.
- Permitir utilizar cartuchos en cadena, sirviendo las salidas de un cartucho como entradas del cartucho siguiente. Se trata de orquestar el funcionamiento de los cartuchos utilizando el patrón Cadena de Responsabilidad [Gamma et al., 1995].

En la Figura 5.1 puede observarse la arquitectura propuesta para la versión 4 de AndroMDA. Los componentes de la herramienta se organizan según el patrón arquitectónico que propone dividir la arquitectura en capas [Buschmann et al., 1996]. Va a utilizarse esta estructura para analizar los principales componentes de la herramienta que son novedad en la versión 4 y que presentan interés desde el punto de vista de esta investigación.

En la capa de servicios técnicos se introduce un motor de workflow. Su función es orquestar las etapas de transformación. Su justificación radica en la complejidad de los flujos de tareas que aparece al someter múltiples modelos de entradas a sucesivas acciones de transformación, recibiendo unas etapas la salida que producen otras. Para la definición de procesos el motor de workflow utiliza un lenguaje XML específico de AndroMDA.

Dentro de la capa de gestión de metadatos se incluyen interfaces para abstracciones básicas comunes para importar y manejar metadatos. Se definen tres abstracciones básicas: un repositorio de metadatos (`IMetadataRepository`) del que se pueden obtener modelos (`IModel`) los cuales están compuestos de elementos de modelado (`IModelElement`). Para utilizar sistemas concretos de gestión de metamodelos, como EMF (§ 3.3.5.2) o MDR (§ 3.3.5.1), deben crearse adaptadores [Gamma et al., 1995] para los interfaces (`IMetadataRepository`) e (`IModel`). Los metamodelos indican la sintaxis abstracta de los modelos, que deberá ser única. Para permitir procesar las diferentes sintaxis concretas definidas sobre ellos, se deben

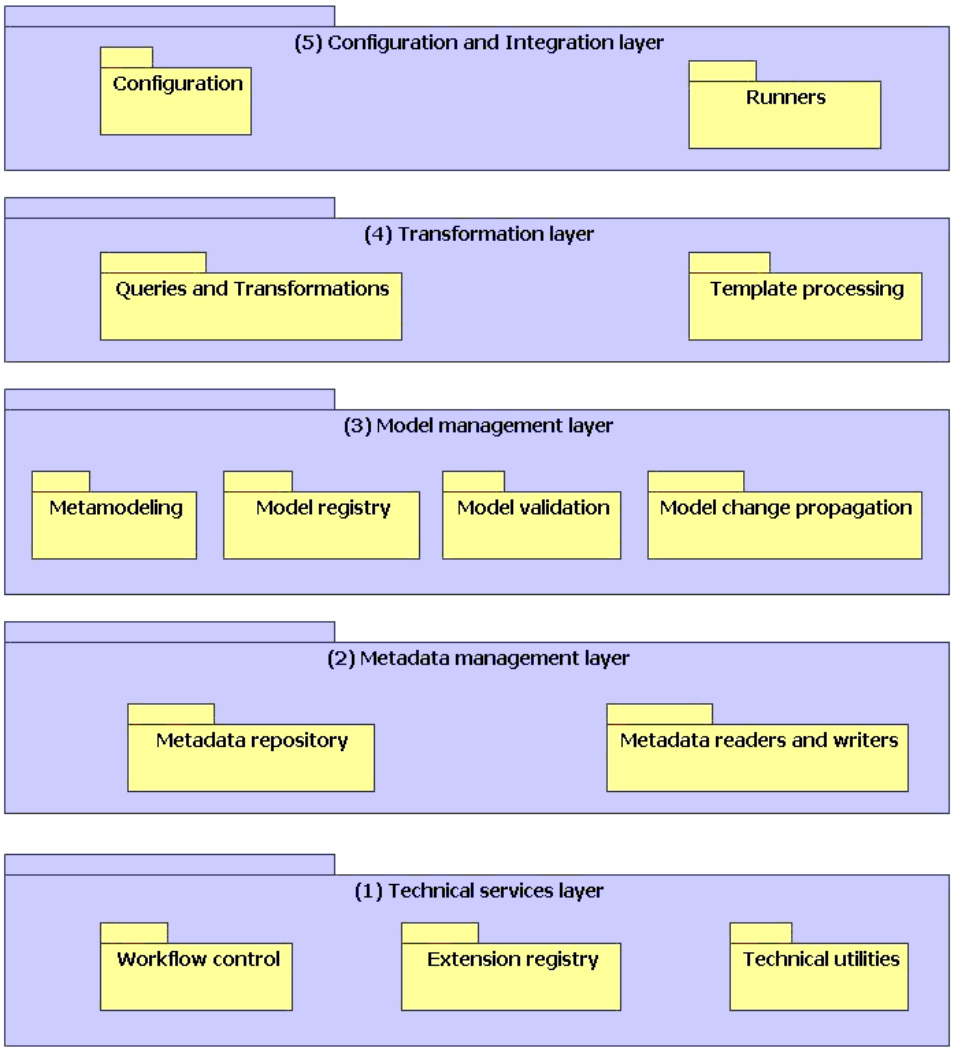

Figura 5.1: Arquitectura de AndroMDA versión 4

crear objetos que sean capaces de leer dicha sintaxis y transformarla a la sintaxis abstracta correspondiente.

En la capa de gestión de modelos se introduce sistema que permite validar modelos durante las transformaciones verificando las restricciones OCL que se definan sobre ellos. También se introduce un sistema encargado de propagar los cambios en los modelos al código. Este sistema tiene como fin atenuar el problema de la modificación manual del código generado que fue descrita en § 5.1.1. Se propone una etapa posterior a la transformación de modelos en las que se detecte los cambios realizados y se generen comandos de *refactoring* [Fowler et al., 1999]. Estos comandos serán posteriormente por un motor que podrá configurarse con alguno de los motores de *refactoring* existentes. De esta manera los cambios introducidos en el modelo pueden ser propagados al código introducido manualmente.

Dentro de la capa de transformación se encuentran los componentes que implementan las transformaciones modelo a modelo y modelo a texto. Para las transformaciones modelo a modelo se usará ATL (§ 4.4) aunque se permite configurar otros motores de transformación que deberán implementar un interfaz `IRunnable` que define tareas ejecutables que el motor de workflow puede orquestar. Este será también el interfaz que deberán implementar los motores de plantillas que se usarán para las transformaciones modelo a texto. Al igual que en la versión 3 se ofrecerán implementaciones para FreeMarker [FreeMarker, 2006] y Velocity [Apache, 2006b].

5.2. OpenArchitectureWare

openArchitectureWare (oAW) [openArchitectureWare, 2006] es un framework generador de aplicaciones modular que sigue una filosofía MDD. Soporta un número arbitrario de modelos de entrada y ofrece una familia de lenguajes que permiten validar y transformar modelos, así como generar código a partir de ellos. Como parte del proyecto se ofrece un importante soporte para la plataforma Eclipse [Eclipse, 2006c] que incluye editores para todos los lenguajes que componen el framework. En la actualidad el framework se encuadra dentro del proyecto GMT de Eclipse [Eclipse, 2006b].

Un aspecto novedoso de cara a esta investigación radica en la propuesta del framework para la gestión de modelos y metamodelos, que difiere al planteamiento estudiado en otros sistemas. Para la especificación de metamodelos oAW implementa un sistema de tipos muy sencillo. Sobre dicho sistema de tipos, se define un lenguaje, muy similar a Java, de que permite definir y ejecutar expresiones. Este lenguaje sirve a su vez como base para el resto de lenguajes del framework. Una primera consecuencia de este diseño es que se presenta una sintaxis unificada para los diferentes lenguajes propuestos.

El sistema de tipos [Efftinge, 2006b] define un conjunto de tipos primitivos. Se define además las entidades básicas del sistema de especificación de metamodelos, que es extremadamente simple: en el sistema de tipos se define la metaclase Tipo (`Type`), que puede tener Propiedades (`Property`) y Operaciones (`Operation`). Además un tipo puede heredar de uno o varios tipos (se permite herencia múltiple).

La principal novedad de oAW radica en que se permite configurar el sistema de tipos con un número arbitrario de implementaciones de metamodelos. De este modo, cuando, desde alguno de los lenguajes definidos por el framework, se haga referencia a un tipo, esta petición será traducida a la implementación concreta utilizada (por ejemplo ECore § 3.3). Los tipos del metamodelo definidos en oAW sirven como enlace con los equivalentes de la implementación utilizado. Por ejemplo, una `Operation` de oAW se corresponde con una `MOperation` de MOF. Las implementaciones de metamodelos que pueden integrarse en el sistema de tipos por defecto son: EMF (§ 3.3.5.2), la implementación de UML sobre EMF realizada en el proyecto Eclipse [**?**], y una implementación basada en Java que utiliza el sistema de introspección de Java [Sun, 1997] para implementar el acceso a los metatipos y a

sus elementos.

Sobre el sistema de tipos descrito se construye un lenguaje de expresiones denominado *Expressions* ([Efftinge, 2006b]). El lenguaje es una mezcla de Java y OCL (§ **??**). Por ejemplo, introduce una sentencia `select` para filtrar los elementos de una colección según un criterio dado en una consulta. Este lenguaje sirve como base para definir el resto de lenguajes del framework.

El framework incluye un lenguaje para definir restricciones sobre modelos denominado *Check* [Efftinge, 2006a]. Se trata de un lenguaje declarativo, similar a OCL (§ **??**), que permite asociar condiciones a los modelos que deben verificarlas.

Para el problema de la transformación de modelos el framework define un lenguaje específico denominado *Extend* [Efftinge, 2006c]. Se trata de un lenguaje de corte funcional [Hudak, 1989]. El lenguaje fue en principio diseñado para permitir extender de los modelos de una forma no intrusiva. Para realizar esta extensión, permite definir librerías de operaciones basadas en Java o en el lenguaje *Extensions*, que son añadidas a los modelos.

A partir de la versión 4.1 de la herramienta, se incluyó en el lenguaje la capacidad de definir transformaciones de modelos. Se introdujo una construcción especial denominada *create extension*. Se trata de operaciones que reciben como parámetro el modelo a transformar y devuelven como resultado el objeto transformado. El cuerpo de estas extensiones puede especificarse, como cualquier otra extensión, utilizando el lenguaje Java o utilizando el lenguaje *Expressions*.

El lenguaje *Extend* permite realizar transformaciones modelo a modelo. Par las transformaciones modelo a texto se define un lenguaje basado en plantillas (§ 4.7.1) denominado *Xpand2* [Efftinge, 2006d]. El lenguaje soporta características avanzadas como el polimorfismo paramétrico. Si hay dos plantillas con el mismo nombre definidas para dos metaclases que heredan de la misma superclase, *Xpand* usa la plantilla de la subclase correspondiente en el caso que ésta sea referenciada como la superclase. Además, permite extender las plantillas de forma no intrusiva utilizando programación orientada a aspectos [Kiczales et al., 1997]. Incluye construcciones para introducir código externo en determinados puntos de una plantilla. El motor se encarga de tejer la plantilla final.

Para orquestar todo el proceso de transformación, la herramienta incluye un motor de workflow [Efftinge and Voelter, 2006a]. Para la especificación de los procesos se utiliza un lenguaje XML específico de oAW. El motor se basa en el patrón de inyección de dependencias [Fowler, 2004] para configurar los componentes que participan en el flujo de procesos. El motor contiene toda la información necesaria para guiar el proceso de generación. En su entrada se indican cuáles son los modelos y metamodelos de entrada, los ficheros de restricciones sobre dichos modelos, así como las transformaciones a las que éstos son sometidos.

Utilizando el framework oAW se ha construido otro framework, denominado *Xtext*, para desarrollar lenguajes DSL textuales [Efftinge and Voelter, 2006b]. Este framework permite definir un DSL a partir de su gramática EBNF [Cueva Lovelle, 1998]. A partir de esta entrada, *Xtext* genera automáticamente

- El meta modelo que representa la sintaxis abstracta del lenguaje definido.

- Un analizador sintáctico (*parser*) que permite procesar la sintaxis concreta de un modelo especificado mediante el DSL e instanciar el *Abstract Syntax Tree* (AST) correspondiente. Para la generación del parser se utiliza Antlr [Antlr, 2006].

- Un editor para Eclipse específico para el DSL creado.

- Los ficheros de restricciones en lenguaje *Check* que permiten definir restricciones semánticas al lenguaje.

5.3. Software Factories

Software factories[3] representa la apuesta de Microsoft para el desarrollo dirigido por modelos. El origen de esta iniciativa se sitúa en una reacción a UML como lenguaje para modelar todos los componentes de un sistema. En su lugar, se apuesta por utilizar intensivamente lenguajes específicos de dominio (DSL) para especificar los diferentes componentes comprendidos en una aplicación.

El término *factory* es una referencia al objetivo que, en última instancia, persigue esta propuesta: industrializar el desarrollo de software [Greenfield et al., 2004]. Esto significa conseguir un proceso de desarrollo que presente características ya presentes en otras industrias con muchos más años de madurez que el software. Entre estas características se encuentran la capacidad de personalizar y ensamblar componentes estándares para producir variantes de un mismo producto; estandarizar, integrar y automatizar los procesos de producción; desarrollar herramientas extensibles y configurables que permitan automatizar las tareas repetitivas. Lo que se busca es conseguir establecer líneas de producción que automaticen la creación de productos.

Al igual que otras iniciativas en el campo del desarrollo dirigido por modelos, *software factories* utiliza modelos formales para la definición de los programas. Los modelos podrán especificarse con un número arbitrario de sintaxis concretas que se edificarán sobre una sintaxis abstracta. La principal diferencia radica en que, en primer lugar, se renuncia a disponer de un lenguaje de metamodelado común para la especificación de los metamodelos de los lenguajes utilizados.

En [Greenfield et al., 2004] se discuten las características de las gramáticas libres de contexto [Cueva Lovelle, 2001] y metamodelos para especificar la sintaxis abstracta de los lenguajes. Se concluye que ambas aproximaciones deben ser combinadas. Por otra parte, se hace énfasis en la necesidad de utilizar lenguajes específicos de dominio (DSL), gráficos o textuales, desarrollados expresamente para la parte del sistema que se desea modelar.

Aunque en los documentos de presentación de la iniciativa se critica continuamente el intento de utilizar UML para modelar todos los aspectos de un sistema [Cook, 2004, Greenfield et al., 2004], la necesidad de crear lenguajes específicos

[3] El término *software factory*, fábrica o factoría de software, se usará sin traducir en esta memoria.

de modelado ya había sido resaltada en la iniciativa MDA [Frankel, 2003]. Ésta fue la principal razón del desarrollo de MOF (§ 3.3), que es el estándar alrededor del cual giran el resto de especificación del OMG que comprenden la iniciativa MDA. Sin embargo, la propuesta de Microsoft no incluye un lenguaje de especificación de metamodelos único, aunque reconoce la importancia de los metamodelos.

Alrededor de los lenguajes específicos de dominio, la propuesta de *software factories* comprende un conjunto de conceptos que conviene analizar. Para dicho análisis se partirá de la definición de *software factory* dada en [Greenfield et al., 2004] por los creadores de la propuesta:

> *A software factory is a software **product line** that configures extensible tools, processes, and content using a software factory **template** based on a software factory **schema** to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring **framework-based components**.*

En la definición se han resaltado los principales componentes de la iniciativa. Se identifican los frameworks como la construcción que liga los modelos al código. Los frameworks comprenden el código que implementa los aspectos que son comunes a un dominio, y exponen los elementos del dominio que varían como puntos de extensión del framework [Cook, 2004]. En *software factories*, el papel de los modelos es definir cómo se extienden dichos puntos.

Otro componente destacado en la iniciativa son los patrones. Los patrones se entiende que son modelos parametrizables que comprenden un conjunto de reglas acerca de cómo pueden combinarse, total o parcialmente, esos con otros [Cook, 2004]. Con este planteamiento, lo que se desea desarrollar es una tecnología que permita ensamblar modelos específicos, frameworks y patrones para construir aplicaciones. De este enfoque surge el concepto de línea de productos.

Al desarrollar una línea de productos se busca capturar sistemáticamente el conocimiento acerca de cómo producir los componentes de la aplicación, exponiéndolo a través de activos de producción. Los activos de producción comprenden activos de implementación, tales como lenguajes, patrones y frameworks; y también comprenden activos de proceso, que serán microprocesos que indican cómo utilizar diversos activos específicos de implementación y herramientas para automatizar partes del proceso [Greenfield et al., 2004].

El concepto de *factory schema* es de vital importancia en la iniciativa. Una *factory schema* se define como un grafo dirigido. Sus nodos representan perspectivas o puntos de vista del sistema desde los cuales se desea desarrollar algún aspecto del sistema [Greenfield et al., 2004]. Por ejemplo, un punto de vista de un sistema puede ser su arquitectura de requisitos. Una vez identificados los puntos de vista del sistema, se utiliza para identificar y agrupar los artefactos que deben desarrollarse para crear el producto software. Las conexiones del grafo representan relaciones entre los diferentes puntos de vista y se denominan *mappings*.

Las relaciones entre los puntos de vista realmente definen las relaciones entre los artefactos en ellos agrupados. Un *mapping* encapsula el conocimiento acerca

de cómo implementar los artefactos descritos por un punto de vista en términos de los artefactos descrito por otro. Los *mappings* pueden ser directos o inversos, según cuál sea la dirección de la derivación. La derivación puede además ser parcial o completa.

Desde el punto de vista de esta investigación, los *mappings* que resultan interesantes son los computables. Con su ejecución se pueden generar, total o parcialmente, los artefactos descritos en el punto de vista de destino a partir de los artefactos contenidos en el punto de vista de origen. Para definir un *mapping* computable, deben utilizarse definiciones formales de los puntos de vista de origen y de destino, o lo que es lo mismo, deben utilizarse lenguajes formales para expresarlos y permitir su computación. El concepto de transformación (Capítulo 4) en *software factories* surge precisamente de la computación de los *mappings*.

Del concepto de *factory schema* se deriva el de *factory template*. Una *factory template* es una instancia de una *factory schema*. Comprende la implementación de los DSL, los patrones, los frameworks y las herramientas.

En [Greenfield et al., 2004] se discuten en profundidad las etapas que comprende un proceso de creación de software factories. El proceso se descompone en dos macroetapas. En la primera se construye el *factory schema*, lo que implica realizar un análisis y un diseño de la línea de productos. En la segunda etapa se construye la *factory template* que instancia la línea de productos software.

Microsoft ha publicado recientemente la versión 1 de DSL Tools [Microsoft, 2006a], que se incluye con la versión 3 del SDK para el entorno de desarrollo Microsoft Visual Studio 2005. DSL Tools Permite la creación de lenguajes específicos de dominio gráficos. Para ello ofrece un diseñador gráfico con el que especificar los símbolos, relaciones y propiedades utilizados en la notación. A partir de la definición del DSL, la herramienta puede generar automáticamente un conjunto de artefactos para su gestión: un editor gráfico específico, APIs para su manipulación, serializadores XML para los modelos especificados con el DSL y un conjunto de plantillas para generar código a partir de ellos. En octubre de 2006 ha publicado una versión preliminar de *The Enterprise Framework Factory (the EFx Factory)*. Se trata de la implementación de una *software factory* para la plataforma .NET [Microsoft, 2006b], que se integra con Microsoft Visual Studio 2005.

La propuesta de software factories ha sido objeto de confrontación entre miembros del OMG y miembros de Microsoft. En *The MDA Journal* se produjo un caluroso debate entre Steve Cook, antiguo miembro del OMG ahora en Microsoft, y Michael Guttman. Steve Cook, además de presentar la propuesta de *software factories*, criticaba al OMG por su propuesta de utilizar UML para modelar todos los aspectos del software. Por su parte, Michael Guttman acusaba a Microsoft de tener como única motivación en su iniciativa el utilizar formatos propietarios, ignorando los estándares existentes del OMG. Un debate similar se produjo entre Grady Booch [Booch, 2004] y Jack Greenfield [Greenfield, 2004, Greenfield, 2005].

5.4. Borland Together Architect Edition

Borland Together Architect Edition [Borland, 2006] es un entorno de desarrollo que se integra con los entornos Eclipse y Visual Studio .NET. El entorno cuenta, desde su versión 2006, con características de MDA.

Desde el punto de vista de esta investigación, este entorno resulta interesante porque introduce, en un producto comercial listo para producción, un conjunto de herramientas que comprenden una solución MDA integral. Comprende un diseñador de diagramas UML 2 para construir los modelos, un motor de transformación modelo a modelo basado en la especificación QVT en su versión previa al estándar finalmente adoptado (§ 4.2)[4] e integra el motor de plantillas JET [Popma, 2003] para la generación de código (transformación modelo a texto).

En su implementación, la herramienta realiza una implementación propia de MOF (§ 3.3) que, en su versión para Eclipse, integra con EMF (§ 3.3.5.2). En cuanto a la implementación de QVT, aunque actualmente es cerrada, se prevé que sea liberada como consecuencia de la colaboración de Borland en el proyecto M2M (§ 4.6) de Eclipse.

5.5. Aportaciones y Carencias para la Investigación

En este capítulo se han analizado diversas propuestas que abordan por completo una solución para el desarrollo dirigido por modelos.

Una de las principales virtudes que ha hecho de AndroMDA (§ 5.1) un herramienta ampliamente utilizada ha sido su enfoque pragmático. Este enfoque se refleja tanto en la implementación de la herramienta como en el código que genera la misma.

Para la implementación de la herramienta, ésta se ha apoyado en todo momento en implementaciones externas de los estándares disponibles. En la versión 3 (§ 5.1.1), se utilizó primero MDR (§ 3.3.5.1) como implementación de MOF. A continuación se hizo evolucionar la arquitectura para soportar EMF (§ 3.3.5.2). Para la generación de código se utilizó Velocity como motor de plantillas [Apache, 2006b], permitiendo en su versión actual aceptar otros motores.

Además, como se indicó en § 5.1, la herramienta hace énfasis en la calidad del código generado automáticamente. Éste hace uso de patrones y buenas prácticas, además de configurarse sobre los frameworks tecnológicos más aceptados por la comunidad Java EE.

Pero la versión actual de AndroMDA también presenta ciertas carencias desde el punto de vista de esta investigación. No sigue un enfoque MDA estricto, donde los modelos se someten a refinamientos y saltos de abstracción mediante transformaciones. Se trata más bien de una herramienta generadora de código. Uno de los principales problemas señalados en § 5.1 es la necesidad de modificar el código una

[4] Únicamente implementa el lenguaje QVT *Operational Mapping* (ver § 4.2.3.1).

vez generado, que contrasta con la desincronización de artefactos que se produce cuándo se modifica el código manualmente.

En § 5.1.2 se analizó la nueva propuesta arquitectónica de AndroMDA que va a resolver estos problemas. En ella sí se apuesta de forma decidida por las transformaciones modelo a modelo. Aunque permite configurar un número arbitrario de motores de transformación, destaca el hecho de utilizar como implementación de referencia ATL (§ 4.4). La razón es que, aunque ATL no es compatible con el estándar QVT (§ 4.2), su implementación open-source goza de mayor madurez y estabilidad que ninguna de las existentes para QVT en el momento actual (ver Capítulo 4). Otro aspecto resaltable de la nueva arquitectura es su intento de mitigar las modificaciones manuales del código generado, analizando los cambios y emitiendo comandos de *refactorings* [Fowler et al., 1999] para que se ejecuten sobre el código modificado.

OpenArchitectureWare § 5.2 también se trata de una ambiciosa propuesta open source, aunque en la actualidad, goza de menos popularidad que AndroMDA. En este caso, se trata de una propuesta que no apuesta por el uso de los estándares del OMG (salvo MOF, que es soportado como una implementación de metamodelos a través de EMF § 3.3.5.2). En su lugar, lo que se propone es un conjunto de lenguajes específicos (DSL) construidos en torno a un sistema de tipos común que permite configurar un número arbitrario de metamodelos. Una consecuencia destacable de este diseño es la implementación de un adaptador que permite utilizar Java como lenguaje de especificación de metamodelos. Además, los lenguajes propuestos utilizan una sintaxis muy similar a la de Java y permiten invocar sentencias Java directamente. Estas características, pueden sin duda facilitar la adopción de los lenguajes propuestos por la comunidad.

Como aspecto negativo de la propuesta debe señalarse que todos los DSL específicos se centran en áreas dónde ya existían estándares o sistemas definidos. Así, para la transformación de modelos (lenguaje *Extend*), se ignoran QVT (§ 4.2) del OMG así como los sistemas como ATL (§ 4.4) o VIATRA (§ 4.3). Del mismo modo, para el lenguaje de restricciones (*Check*), existe OCL (§ 3.1.5) el cuál comienza a gozar de soporte por parte de las herramientas (§ 5.4, § 5.1.2). Esta situación es consecuencia de una decisión de diseño deliberada: ofrecer una sintaxis única para todos los DSL del framework de transformación. Pero también implica que el desarrollador quedará ligado a la herramienta puesto que sus especificaciones de modelos (en el momento en el que no estén basadas en MOF-EMF) y, sobre todo, de transformaciones, no serán portables.

La propuesta de *software factories* de Microsoft § **??** aporta cuestiones interesantes al problema del desarrollo dirigido por modelos, aunque también presenta severas carencias. Presenta un objetivo ambicioso, industrializar el desarrollo de software. Para cumplir dicho objetivo, identifica una serie de artefactos clave, además de los modelos formales. Entre estos artefactos pueden señalarse los patrones, las buenas prácticas y los frameworks. Nótese que, aunque AndroMDA (§ 5.1) tenga un planteamiento totalmente distinto, otorga importancia a los mismos elementos durante la generación de código. El planteamiento de Microsoft difiere de la pro-

puesta de MDA del OMG en cuanto a que no busca la independencia de la plataforma. La iniciativa *software factories* se estructura en torno a una plataforma (.NET) y a un entorno de desarrollo (Microsoft Visual Studio).

Otra aportación importante de las *software factories* de Microsoft es la importancia que otorga a los lenguajes específicos de dominio (DSL). Los modelos se especificarán formalmente con lenguajes diseñados específicamente para el dominio de éstos. En la actualidad, el grupo de estándares de MDA del OMG gira en torno a la especificación MOF, que define un lenguaje para especificar formalmente metamodelos. Es decir, se asume que será necesario crear DSL en forma de sintaxis concretas sobre un metamodelo definido formalmente. Y este metamodelo estará definido con MOF. En la iniciativa de *software factories*, sin embargo, no se utiliza un mecanismo común para especificar metamodelos. Esta aproximación presenta el inconveniente de que no se puede crear un lenguaje de transformación genérico basado en los metamodelos de los modelos de entrada y salida del proceso de transformación.

Es precisamente la discusión "UML frente a DSL" la que ha motivado fuertes discusiones públicas entre el OMG y Microsoft. Debe entenderse que, en 2004, cuando Microsoft se publicó el artículo de Steve Cook en el que se explicaban los fundamentos de la iniciativa de Microsoft [Cook, 2004], la especificación de MOF 2 aún no existía, y la especificación de QVT estaba en medio de un largo proceso de elaboración que terminó a finales del año siguiente [OMG, 2005a]. Baste poner como ejemplo el artículo de Grady Booch de finales de 2004 [Booch, 2004] donde indicaba que casi siempre podía desarrollarse un perfil UML [**?**] para especificar los modelos, y en los casos en los que no, siempre podría hacerse un DSL específico sobre la semántica de UML.

Por otra parte, el concepto de *factory schema* (§ 5.3), cuando todos los *mappings* son computables, lo que se obtiene es un motor que orquesta transformaciones. Se trata de una estructura equivalente a los motores de workflow que ofrece oAW (§ 5.2) o que propone AndroMDA en su versión 4 (§ 5.1.2). El problema es que las transformaciones en *software factories*, al igual que la especificación de modelos, se resuelven con tecnologías y herramientas propietarias de Microsoft.

Capítulo 6

CONCLUSIONES

6.1. Conclusiones

A continuación se recogen las principales conclusiones extraídas de la investigación.

6.1.1. Especificación de Modelos en MDD

Como se concluyó en el Capítulo 3, el mejor lenguaje para especificar modelos en MDD simplemente no existe. La aproximación más inteligente es utilizar lenguajes específicos de dominio (DSL) específicos para cada aspecto del sistema a resolver.

En este contexto, debe señalarse que UML está destinado a jugar un papel importante en MDD. Es el estándar más conocido de los que componen el *framework* MDA y también el que cuenta con un mayor soporte por parte de las herramientas de modelado. Por otra parte, las últimas versiones del lenguaje y en especial la última (UML 2) han estado orientadas a mejorar el soporte del lenguaje a MDA.

Sin embargo, se han encontrado diversas debilidades en UML. Por un lado, se trata de un lenguaje extremadamente amplio. Como se ha visto a lo largo de esta Memoria, este hecho convierte a UML en un lenguaje difícil de utilizar de una manera formal, que es la manera requerida por MDA. Por otro lado, si bien se ha visto que las construcciones estructurales de UML, en especial los modelos de clases y derivados, resultan muy adecuadas para modelar los aspectos estructurales de los sistemas (incluso en dominios especializados), se han detectado puntos débiles en sus artefactos para describir comportamiento de una manera formal. Se han analizado tres aproximaciones para modelar comportamiento formalmente:

- UML (§ 3.1).

- UML complementado con OCL (§ 3.1.5).
- UML Ejecutable (§ 3.2).

De las tres, la única que permite especificar construir modelos computacionalmente completos es la tercera, aunque también ésta presenta sus inconvenientes derivados de la ausencia de una sintaxis concreta estándar y, sobre todo, del bajo nivel de abstracción en el que plantea modelar comportamiento. Otro aspecto a favor de esta aproximación son las profundas modificaciones que ha sufrido el metamodelo de UML 2 para mejorar la integración de *Action Semantics*. Esto habilita a las herramientas UML de propósito general para comenzar a soportar UML Ejecutable en el futuro. Además, indica que el OMG ha entendido como necesario el disponer de mecanismos de especificación de comportamiento imperativos en los modelos PIM, y no sólo declarativos.

La especificación de UML viene dada por una sintaxis abstracta definida por su metamodelo. Dicho metamodelo es un mecanismo de especificación especialmente adecuado para los desarrolladores de herramientas. Describe las construcciones manejadas en el lenguaje en términos de conceptos básicos de orientación a objetos. Cualquier herramienta que manipule modelos UML deberá utilizar una implementación de dicho metamodelo, para poder manipular los conceptos del lenguaje programáticamente. La separación la sintaxis abstracta (metamodelo) de posibles representaciones concretas permite crear una arquitectura basada en un repositorio único de elementos de modelado. Para alimentar dicho repositorio el usuario puede utilizar diferentes representaciones concretas o "vistas" de los elementos del repositorio, siendo una de ellas la notación gráfica UML. Dicho repositorio podría jugar en una herramienta MDA un papel similar a una tabla de símbolos en un compilador tradicional.

6.1.2. El Papel de MOF en MDD

MOF es sin duda el estándar más importante de cuántos define la iniciativa MDA. Se trata del mecanismo propuesto para especificar lenguajes DSL específicos a los aspectos y dominios del problema a resolver (§ 3.3). En la comunidad existe un consenso acerca de la necesidad de utilizar DSL en cualquier aproximación al software dirigido por modelos. Donde no existe acuerdo es acerca de si MOF es la tecnología con la que deben especificarse todos los DSL creado.

El hecho de que los metamodelos de los lenguajes creados compartan el mismos meta-metamodelos tiene importantes ventajas. La principal es que pueden generarse servicios diseñados para trabajar con instancias de MOF, y reutilizar dichos servicios con diferentes lenguajes con diferentes metamodelos MOF. El mecanismo más conocido y utilizado es XMI, que permite serializar en XML cualquier modelo creado con un lenguaje edificado sobre MOF. Sin embargo, también debe de tenerse en cuenta que MOF únicamente especifica la sintaxis de los metamodelos creados, esto es, sus aspectos estructurales. Por ello, aunque suela ser definido como el estándar que permite especificar nuevos lenguajes para MDA, debe entenderse que

únicamente especifica su estructura, no su dinámica de ejecución o cómo deben interpretarse dichos lenguajes.

XMI es quizás el estándar del OMG que mejor demuestra las ventajas derivadas de utilizar arquitecturas basadas en estándares (§ 3.3.4.1). Ha sido un estándar abrazado por las herramientas comerciales, en especial para representar modelos UML, y esto ha permitido que la idea de intercambiar modelos se haya convertido en una realidad. No obstante también se ha visto un problema con el desarrollo de las especificaciones por parte del OMG. La arquitectura de MOF y de UML se ha visto drásticamente modificada en los últimos años, y con cada modificación se ha definido un estándar XMI diferente. Como se ha visto, esto ha repercutido negativamente en la compatibilidad de herramientas, máxime cuando se tratan de estándares muy complejos y costosos de implementar.

Un área que demuestra la utilidad de tener metamodelos comunes a los modelos manejados es la transformación de modelos. El disponer de un lenguaje de metamodelado común, permite especificar procesos genéricos de transformación que manipulan los elementos de los modelos de origen y de destino de la transformación a través de sus metatipos. En los capítulos 4 y 5 se han analizado diversos sistemas que, si bien no se trata de aproximaciones MDA puras, se basan en implementaciones de MOF para la definición de sus sistemas de transformación.

Por otra parte, una de las mayores modificaciones en la arquitectura de MOF que se ha realizado con UML 2.0 ha sido crear una biblioteca con un núcleo reutilizable de conceptos para definir tanto MOF como UML (biblioteca *Infrastructure*, § 3.3.3). Este enfoque permite a un desarrollador de herramientas desarrollar el núcleo de dicha infraestructura y, sobre él, generar automáticamente el resto de especificaciones basadas en ella mediante *bootstrapping*, leyendo su especificación en un formato manipulable, por ejemplo en XMI.

6.1.3. Problema de Soporte Comercial

El principal problema de los estándares sobre los que se edifica MDA es su escaso soporte comercial. En el mundo del software, el riesgo de que los estándares nacidos y diseñados en un comité no lleguen a tener éxito comercial es alto[1]. Además, en el caso de MDA, este problema se ve agudizado porque los estándares producidos por el OMG, dado lo ambicioso de sus objetivos, resultan extremadamente extensos. Como ejemplo pueden citarse las casi mil páginas que comprenden los dos volúmenes que especifican el lenguaje UML. Este problema se ve agravado por el hecho de que las especificaciones no están ligadas a plataformas tecnológicas concretas. Por otra parte, las especificaciones del OMG viene especificadas de una manera informal utilizando como herramienta básica el lenguaje natural. Se encuentra formalidad en los metamodelos de los lenguajes definidos, cuya estructura

[1] Como prueba puede señalarse el fracaso de la plataforma de EJBs en versiones J2EE anteriores a la 1.5. En la versión 1.5, se ha planteado una arquitectura totalmente distinta basada en implementaciones comerciales exitosas [Sun, 2005]. El ejemplo más claro es el sistema de persistencia de EJBs con un enfoque idéntico al de Hibernate [Bauer and King, 2004].

se formaliza definiendo su sintaxis abstracta, pero la semántica de los mismos viene definida de una manera informal. Además, no existen implementaciones de referencia que los fabricantes puedan utilizar para evaluar la completud de sus desarrollos.

Los anteriores problemas, junto con las profundas modificaciones que han sufrido los estándares MDA en los últimos años, hacen que su implementación por parte de las herramientas comerciales sea muy lenta. Este problema no debe subestimarse. Quizás sea el mayor problema que existe con la iniciativa MDA. Las implementaciones comerciales son un requisito clave para que la promesa se haga realidad.

Un caso particular del problema señalado es el escaso soporte comercial existente a UML Ejecutable (§ 3.2). Aunque *Action Semantics* fue estandarizado en 2001 [OMG, 2001] e incluido como parte de UML en 2003 [OMG, 2003], no es implementado ni siquiera por las herramientas UML comerciales actuales más completas. Como razones que se han señalado pueden destacarse la inexistencia de una sintaxis concreta estándar para un lenguaje de acción, así como la dificultad de que tanto los usuarios como los desarrolladores de herramientas adopten e implementen nuevos lenguajes de programación.

6.1.4. Escepticismo Respecto a la Propuesta MDA

Al analizar las diferentes posiciones adoptadas en la comunidad de desarrollo dirigido por modelos respecto a la propuesta MDA del OMG se encuentra una crítica recurrente: se pone en duda la posibilidad real de que dicha propuesta llegue a materializarse, y en especial, que dicha materialización llegue a realizarse a través de los estándares que componen el *framework* MDA.

Para empezar, se ha visto como en el corazón de la propuesta se plantean preguntas que no tienen una única respuesta, como por ejemplo, ¿qué se entiende por independencia de la plataforma? o ¿qué se considera un nivel elevado de abstracción?. Por otra parte, existen problemas de índole técnico directamente relacionados con sus estándares. Se plantea especificar los diferentes componentes de los sistemas mediante modelos formales independientes de la plataforma. Sin embargo, el lenguaje de modelado estándar UML ha mostrado importantes carencias a la hora de dar respuesta a esta necesidad. En primer lugar, existen aspectos relativos a dominios específicos de los sistemas que no pueden ser modelados con UML de una manera estándar, como por ejemplo los interfaces de usuario. Además, se han detectado serios problemas en UML en su propósito de ser un lenguaje de modelado de propósito general: aunque ofrece diversos mecanismos para modelar el comportamiento de los sistemas, no existe un ningún enfoque que solucione completamente este problema. De los enfoques estudiados en los Capítulos 3 y § 3.2, la mejor aproximación analizada ha sido la propuesta por UML Ejecutable y los Lenguajes de Acción, aunque como se ha señalado, esta apuesta no exenta de inconvenientes.

Con respecto a la definición de lenguajes específicos de dominio utilizando MOF, resulta difícil predecir su futura materialización real en formas de lenguajes concretos. Es cierto que MOF permite definir lenguajes cuyos metamodelos com-

parten una estructura común y se han señalado diversas ventajas de este enfoque (§ 3.3). Sin embargo cabe hacerse la pregunta acerca de sí esta sintaxis común facilita notablemente la implementación de nuevos lenguajes que puedan ser utilizados por los desarrolladores. Por ejemplo, si hasta ahora no se ha definido una manera estándar de modelar interfaces de usuario para aplicaciones de escritorio, ¿el disponer de una mecanismo estándar para especificar sintaxis abstractas de lenguajes hará que la situación cambie?.

Quizás la razón de este escepticismo y de las fuertes y contrarias reacciones que produce la propuesta MDA se encuentre en los mensajes de "*marketing*" que han aparecido alrededor de la propuesta del OMG. Estos aspectos hacen referencia cuestiones tales como la posibilidad de utilizar UML para especificar todos los aspectos de un sistema, o la posibilidad de lograr una independencia absoluta de la plataforma de ejecución a la hora de especificar los sistemas. Como señala David Frankel en [Frankel, 2004b], la propuesta de MDA no puede ser entendida hoy más que como una propuesta a largo plazo que ya está ofreciendo resultados parciales a corto plazo.

6.1.5. Transformación de Modelos

La transformación de modelos aparece, junto con el problema de la especificación de modelos, como uno de los grandes problemas a resolver en una herramienta MDD. En el Capítulo 4 diferentes propuestas para la transformación de modelos.

Si en una herramienta MDD los modelos son artefactos de primer nivel, es necesario un proceso que permita transformar dichos modelos en una aplicación ejecutable. Para resolver este problema el OMG ha propuesto QVT (§ 4.2), al que se suman otras alternativas como ATL (§ 4.4) o VIATRA (§ 4.3). Todas estas alternativas se basan en el mismo enfoque. Ofrecen lenguajes para procesar transformaciones entre elementos de modelado que tienen un metamodelo común. Es decir, las definiciones de transformación se definen sobre los metamodelos origen y destino de la transformación. En cuanto a los lenguajes, aunque QVT ofrece tanto una alternativa imperativa como una declarativa, las implementaciones demuestran una preferencia por la versión imperativa. Del mismo modo, ATL, que goza de cierta implantación en la industria, ofrece un lenguaje de transformación imperativo.

El ciclo de vida de un proceso de transformación comprende dos etapas. En la primera, se realizan transformaciones modelo a modelo, que permiten trasladar a los modelos de dominio y someterlos a saltos de abstracción, especializándolos para la plataforma tecnológica de destino. En la segunda etapa se recogen los modelos especializados y se realiza una traducción de modelo a texto, generando el código fuente y los artefactos de configuración necesarios. Este es el enfoque propuesto por el patrón MDA (§ 4.1) y es el enfoque seguido por las herramientas MDD estudiadas más avanzadas. La razón de no traducir directamente los modelos de entrada a código fuente responde a la necesidad de un proceso que permita absorber la complejidad del ciclo de transformación.

6.2. Trabajo Futuro

En la primera parte de la investigación se han analizado los principales estándares de la propuesta MDA, estudiando los puntos fuertes y débiles que plantean sus especificaciones, así como el soporte dado en implementaciones reales. Se han detectado diversos problemas y necesidades relativos a dichos estándares y se han analizado diferentes posturas existentes en la comunidad respecto de su uso.

La segunda fase de la investigación se ha centrado en estudiar el problema de la transformación de modelos y diversas herramientas MDD existentes. Como consecuencia se ha ampliado el objetivo del estudio englobando la iniciativa MDA dentro de un paradigma mayor, el del desarrollo dirigido por modelos.

A continuación se plantean futuras líneas de investigación extraídas del trabajo realizado.

Especificación de Modelos mediante DSL. Una vez abandonada la idea de utilizar UML para modelar todos los aspectos de un sistema, se sugiere la posibilidad de construir lenguajes específicos al domino, gráficos o textuales, del problema que simplifique la labor del desarrollador.

Desde el punto de vista de una herramienta de MDD, los DSL deben representar una sintaxis concreta de las múltiples que pueden establecerse sobre una única sintaxis abstracta. Esta sintaxis abstracta vendrá dada por un lenguaje de especificación de metamodelos.

Con este planteamiento, puede investigarse la posibilidad de crear un sistema de especificación de DSL basado en MOF. El sistema debe ofrecer algún mecanismo que permita especificar la sintaxis del lenguaje. Además debe ofrecer la posibilidad de introducir restricciones semánticas acerca de los elementos del mismo. Con estas entradas, el sistema debe ser capaz de generar un analizador que instancie los elementos del metamodelo del lenguaje a partir de una entrada válida en el lenguaje.

Especificación de Comportamiento. Se ha analizado cómo aún no existe un mecanismo claro acerca de cómo especificar el comportamiento de un sistema.

Las transformaciones entre modelos resultan especialmente eficientes cuando los modelos origen y destino de la transformación se corresponden con una vista estática del dominio al que pertenecen. Los frameworks y tecnologías subyacentes permiten absorber parte de la complejidad de la dinámica del sistema, pero siempre es necesario especificar el comportamiento de los objetos manualmente.

Para esta especificación, puede crearse un DSL que instancie la sintaxis abstracta de *Action Semantics* de UML 2. La sintaxis concreta podría corresponderse con un lenguaje imperativo ampliamente extendido, como Java, para facilitar su aprendizaje.

Gestión de los Flujos de Transformación. Las transformaciones representan procesos complejos que deben orquestarse. Se han estudiado diversas propuestas que proponen usar un motor de workflow para coordinar las etapas de transformación y generación de código. Sin embargo, estas herramientas se basan en formatos de especificación de procesos propietarios, específicos de cada una de ellas. Esto impide, por ejemplo, utilizar herramientas existentes que soporten estándares de especificación de procesos, como *XML Process Definition Language* (XPDL) de la *Workflow Management Coalition* (WfMC) ([**?**]) o *Business Process Execution Language* (BPEL) de OASIS [OASIS, 2006].

Por lo tanto se propone estudiar las posibilidades de extensión y especialización de los estándares existentes para permitir especificar con ellos flujos de transformación.

LISTA DE ACRÓNIMOS

ASL (*Action Specification Language*)

ASM (*Abstract State Machine*)

ASP (*Active Server Pages*)

AST (*Abstract Syntax Tree*)

ATC (*Atomic Transformation Code*)

ATL (*ATLAS Transformation Language*)

BPEL (*Business Process Execution Language*)

CASE (*Computer-Aided Software Engineering*)

CMOF (*Complete MOF*)

CORBA (*Common Object Request Broker Architecture*)

CRUD (*Create, read, update and delete*)

CWM (*Common Warehouse Metamodel*)

DAO (*Data Access Object*)

DDL (*Data Definition Language*)

DSL (*Domain Specific Language*)

DTD (*Document Type Definition*)

EBNF (*Extended Backus Naur Form*)

EMF (*Eclipse Modeling Framework*)

EMOF (*Essential MOF*)

GMT (*Generative Modeling Technologies*)

HUTN (*Human-Usable Textual Notation*)

INRIA (*Institut national de recherche en informatique et en automatique*)

JCP (*Java Community Process*)

Java EE (*Java Platform, Enterprise Edition*)

JET (*Java Emitter Templates*)

JMI (*Java Metadata Interface*)

JSF (*Java Server Faces*)

JSP (*Java Server Pages*)

MDA (*Model Driven Architecture*)

MDD (*Model Driven Development*)

MDR (*Metadata Repository*)

MOF (*Meta Object Facility*)

MOMENT (*MOdel manageMENT*)

M2M (*Model-to-Model Transformation*)

M2T (*Model-to-Text Transformation*)

OCL (*Object Constraint Language*)

OMG (*Object Management Group*)

OMT (*Object Modeling Technique*)

OOPSLA (*Object-Oriented Programming, Systems, Languages & Applications*)

OOSE (*Object-oriented Software Engineering*)

PIM (*Platform Independent Model*)

PSI (*Platform Specific Implementation*)

PSL (*Process Specification Language*)

PSM (*Platform Specific Model*)

QVT (*Query/View/Transformation*)

RFP (*Request for Proposal*)

RUP (*Rational Unified Proccess*)

SDL (*Specification and Design Language*)

SQL (*Structured Query Language*)

TDD (*Test-Driven Development*)

UML (*Unified Modeling Language*)

UP (*Unified Proccess*)

VIATRA (*VIsual Automated model TRAnsformations*)

VPM (*Visual and Precise Metamodeling*)

VTCL (*Viatra Textual Command Language*)

VTML (*Viatra Textual (Meta)Modeling Language*)

VTTL (*Viatra Textual Template Language*)

WfMC (*Workflow Management Coalition*)

XMI (*XML Metadata Interchange*)

XML (*Extensible Markup Language*)

XP (*Extreme Progamming*)

XPDL (*XML Process Definition Language*)

XSD (*XML Schema Definition*)

XSLT (*Extensible Stylesheet Language Transformations*)

BIBLIOGRAFÍA

[Abdurazik and Offutt, 2000] Abdurazik, A. and Offutt, J. (2000). Using UML Collaboration Diagrams for Static Checking and Test Generation. *Proceeding of the UML 2000 - The Unified Modeling Language. Advancing the Standard: Third International Conference*.

[Agile Alliance, 2006] Agile Alliance (2006). Agile Alliance Home Site. `http://www.agilealliance.org`.

[Akehurst and Bordbar, 2001] Akehurst, D. H. and Bordbar, B. (2001). On Querying UML Data Models with OCL. *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts and Tools, 4th International Conference, Toronto, Canada*.

[Alur et al., 2003] Alur, D., Malks, D., and Crupi, J. (2003). *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall.

[Ambler, 2004] Ambler, S. W. (2004). *Agile Model Driven Development with UML 2*. Cambridge University Press, 3rd edition.

[Ambler and Jeffries, 2002] Ambler, S. W. and Jeffries, R. (2002). *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. Wiley, 1st edition.

[AndroMDA, 2006] AndroMDA (2006). Andromda bpmfors-truts cartridge. `http://galaxy.andromda.org/docs/andromda-bpm4struts-cartridge/index.html`.

[AndroMDA, 2006] AndroMDA (2006). Andromda cartridges documentation. `http://galaxy.andromda.org/docs/andromda-cartridges/index.html`.

[AndroMDA, 2006] AndroMDA (2006). Andromda home site. `http://www.andromda.org/`.

[AndroMDA, 2006] AndroMDA (2006). Andromda metafacades. `http://galaxy.andromda.org/docs/andromda-metafacades/index.html`.

[Antlr, 2006] Antlr (2006). Antlr Parser Generator Home. AntlrParserGeneratorHome.

[Apache, 2006a] Apache (2006a). Apache Struts. http://struts.apache.org/.

[Apache, 2006b] Apache (2006b). Apache Velocity Home Page. http://jakarta.apache.org/velocity/.

[Architecture, 2006] Architecture, A. (2006). Matthias bohlen. http://galaxy.andromda.org/index.php?option=com_content&task=blogsection&id=10&Itemid=78.

[Bauer, 1999] Bauer, B. (1999). Extending UML for the Specification of Interaction Protocols. *Submission for the 6th Call for Proposal of FIPA and revised version part of FIPA 99*.

[Bauer and King, 2004] Bauer, C. and King, G. (2004). *Hibernate in Action*. Action. Manning Publications, 1st edition.

[Beck, 1999] Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1st edition.

[Beck, 2002] Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley Professional, 1st edition.

[Beck et al., 2001] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Agile Manifesto. http://agilemanifesto.org/.

[Belaunde and Dupe, 2006] Belaunde, M. and Dupe, G. (2006). SmartQVT Home Page. http://smartqvt.elibel.tm.fr/index.html.

[Bergsten, 2003] Bergsten, H. (2003). *JavaServer Pages, Third Edition*. O'Reilly & Associates, third edition.

[Beust, 2006] Beust, C. (2006). Agile people still don't get it. *Otaku, Cedric's Weblog*. Disponible en http://beust.com/weblog/archives/000392.html.

[Blankenhorn and Jeckle, 2004] Blankenhorn, K. and Jeckle, M. (2004). A UML Profile for GUI Layout. *Proceedings of Object-Oriented and Internet-Based Technologies: 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*, pages 27–30.

[Bock, 1999a] Bock, C. (1999a). Unified Behavior Models. *Journal Of Object-Oriented Programming*, 12(5).

[Bock, 1999b] Bock, C. (1999b). Unified Behavior Models. *Journal Of Object-Oriented Programming*, 12(5).

[Bock, 2003a] Bock, C. (2003a). UML 2 Activity and Action Models. *Journal Of Object Technology*, 2(4):43–53.

[Bock, 2003b] Bock, C. (2003b). UML 2 Activity and Action Models. Part 2: Actions. *Journal Of Object Technology*, 2(5):42–56.

[Bock, 2003c] Bock, C. (2003c). UML without Pictures. *IEEE Software*, pages 33–35.

[Bock, 2004] Bock, C. (2004). UML 2 Composition Model. *Journal of Object Technology*, 3(10):47–73.

[Bock, 2005] Bock, C. (2005). UML 2 Activity and Action Models. Part 6: Structured Activities. *Journal Of Object Technology*, 4(4):44–66.

[Boehm, 1988] Boehm, B. (1988). A spiral model of software development and enhancement. *Computer*, 21(5):61–72.

[Bohlen, 2002] Bohlen, M. (2002). *Enterprise Java Beans Ge-Packt.* Mitp-Verlag.

[Bohlen, 2006] Bohlen, M. (2006). The AndroMDA Architecture. `http://galaxy.andromda.org/index.php?option=com_content&task=blogsection&id=10&Itemid=78`.

[Booch, 1993] Booch, G. (1993). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional, 2nd edition.

[Booch, 2004] Booch, G. (2004). Microsoft and Domain Specific Languages. `http://www-03.ibm.com/developerworks/blogs/page/gradybooch?entry=microsoft_and_domain_specific_languages`.

[Booch et al., 2004] Booch, G., Brown, A., Iyengar, S., Rumbaugh, J., and Selic, B. (2004). An MDA Manifesto. *The MDA Journal*, pages 133–144.

[Booch et al., 1996] Booch, G., Jacobson, I., and Rumbaugh, J. (1996). The Unified Modeling Language for Object-Oriented Development. *Rational Software Corporation. Documentation Set. Version 0.91. Addendum. UML Update*. Disponible en `http://www.microgold.com/Stage/uml91.pdf`.

[Booch et al., 1999] Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley Professional, 1st edition.

[Borland, 2006] Borland (2006). Borland Together Architect 2006. `http://www.borland.com/`.

[Börger et al., 2000] Börger, E., Cavarra, A., and Riccobene, E. (2000). An ASM Semantics for UML Activity Diagrams. *Proceedings of Algebraic Methodology and Software Technology: 8th International Conference, AMAST*.

[Börger and Stärk, 2003] Börger, E. and Stärk, R. (2003). Abstract state machines. a method for high-level system design and analysis. *Springer-Verlag*.

[Budinsky et al., 2003] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., and Grose, T. J. (2003). *Eclipse Modeling Framework: A Developer's Guide*. Addison Wesley.

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. 1st edition.

[Chamberlin and Boyce, 1974] Chamberlin, D. D. and Boyce, R. F. (1974). SEQUEL: A structured English query language. *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264.

[Clavel et al., 2003] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2003). The maude 2.0 system. In Nieuwenhuis, R., editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag.

[Clossman et al., 1998] Clossman, G., Shaw, P., Hapner, M., J. Klein, R. P., and Becker, B. (1998). *Java and Relational Databases: SQLJ*. ACM SIGMOD Record.

[Cockburn, 2000] Cockburn, A. (2000). *Writing Effective Use Cases*. Addison-Wesley Professional, 1st edition.

[Cockburn, 2001] Cockburn, A. (2001). *Agile Software Development*. Addison-Wesley Professional, 1st edition.

[Cockburn, 2004] Cockburn, A. (2004). *Crystal Clear : A Human-Powered Methodology for Small Teams*. Addison-Wesley Professional, 1st edition.

[Contract4J, 2006] Contract4J (2006). Contract4j. design by contract for java. `http://www.contract4j.org/contract4j`.

[Cook, 2004] Cook, S. (2004). Domain-Specific Modeling and Model Driven Architecture. *The MDA Journal*, pages 80–94.

[Cook and Daniels, 1994] Cook, S. and Daniels, J. (1994). *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice Hall, 1st edition.

[Cueva Lovelle, 1998] Cueva Lovelle, J. M. (1998). *Conceptos Básicos de Procesadores de Lenguaje*. Cuaderno Didáctico número 10. Editorial Servitec.

[Cueva Lovelle, 2001] Cueva Lovelle, J. M. (2001). *Lenguajes, gramáticas y autómatas*. Editorial Servitec.

[Dijkstra, 1968] Dijkstra, E. W. (1968). Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11:147–148.

[Douglass, 1999] Douglass, B. P. (1999). *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Pearson Education, 2nd edition.

[Dumas and ter Hofstede, 2001] Dumas, M. and ter Hofstede, A. H. (2001). UML Activity Diagrams as a Workflow Specification Language. *Proceedings of Modeling Languages, Concepts, and Tools: 4th International Conference, Toronto, Canada*.

[Eclipse, 2006a] Eclipse (2006a). *ATL Transformations List*. Eclipse Foundation.

[Eclipse, 2006b] Eclipse (2006b). Eclipse gmt home page. `http://www.eclipse.org/gmt/`.

[Eclipse, 2006c] Eclipse (2006c). Eclipse Home. `http://www.eclipse.org/`.

[Eclipse, 2006d] Eclipse (2006d). Eclipse Modeling Project Home Page. `http://www.eclipse.org/modeling/`.

[Eclipse, 2006e] Eclipse (2006e). Eclipse Modelling Framework (EMF). `http://www.eclipse.org/emf/`.

[Eclipse, 2006f] Eclipse (2006f). UML 2. `http://www.eclipse.org/uml2/`.

[Efftinge, 2006a] Efftinge, S. (2006a). *OpenArchitectureWare 4.1 Check - Validation Language*. openArchitectureWare.

[Efftinge, 2006b] Efftinge, S. (2006b). *OpenArchitectureWare 4.1 Expressions Framework Reference*. openArchitectureWare.

[Efftinge, 2006c] Efftinge, S. (2006c). *OpenArchitectureWare 4.1 Extend Language Reference*. openArchitectureWare.

[Efftinge, 2006d] Efftinge, S. (2006d). *OpenArchitectureWare 4.1 Xpand2 Language Reference*. openArchitectureWare.

[Efftinge and Voelter, 2006a] Efftinge, S. and Voelter, M. (2006a). *OpenArchitectureWare 4.1 Workflow Engine Reference*. openArchitectureWare.

[Efftinge and Voelter, 2006b] Efftinge, S. and Voelter, M. (2006b). *OpenArchitectureWare 4.1 Xtext Reference Documentation*. openArchitectureWare.

[Ehrig et al., 1999] Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G. (1999). Handbook on graph grammars and computing by graph transformation. *Applications, Languages and Tools. World Scienti?c*, 2.

[Engels et al., 1999] Engels, G., Sauer, S., and Wagner, A. (1999). UML Collaboration Diagrams and Their Transformation to Java. *Proceedings of The Unified Modeling Language: Beyond the Standard, Second International Conference, Fort Collins, CO, USA*.

[Eshuis and Wieringa, 2001] Eshuis, R. and Wieringa, R. (2001). A Real-Time Execution Semantics for UML Activity Diagrams. *Proceedings of Fundamental Approaches to Software Engineering : 4th International Conference, FASE 2001*.

[Fowler, 1996] Fowler, M. (1996). *Analysis Patterns : Reusable Object Models*. Addison-Wesley Professional, 1st edition.

[Fowler, 2000] Fowler, M. (2000). Acronym pojo. `http://www.martinfowler.com/bliki/POJO.html`.

[Fowler, 2003a] Fowler, M. (2003a). UML As Blueprint. Disponible en `http://www.martinfowler.com/bliki/UmlAsBlueprint.html`.

[Fowler, 2003b] Fowler, M. (2003b). UML As Programming Language. *Martin Fowler's Bliki*. Disponible en `http://www.martinfowler.com/bliki/UmlAsProgrammingLanguage.html`.

[Fowler, 2003c] Fowler, M. (2003c). Uml As Sketch. Disponible en `http://www.martinfowler.com/bliki/UmlAsSketch.html`.

[Fowler, 2003d] Fowler, M. (2003d). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison Wesley, third edition.

[Fowler, 2003e] Fowler, M. (2003e). Uml Mode. Disponible en `http://www.martinfowler.com/bliki/UmlMode.html`.

[Fowler, 2003f] Fowler, M. (2003f). Unwanted Modeling Language. *Martin Fowler's Bliki*. Disponible en `http://martinfowler.com/bliki/UnwantedModelingLanguage.html`.

[Fowler, 2004] Fowler, M. (2004). Inversion of control containers and the dependency injection pattern. *Martin Fowler's Bliki*. Disponible en `http://martinfowler.com/articles/injection.html`.

[Fowler, 2005a] Fowler, M. (2005a). Language Workbenches and Model Driven Architecture. *Martin Fowler's Bliki*. Disponible en `http://martinfowler.com/articles/mdaLanguageWorkbench.html`.

[Fowler, 2005b] Fowler, M. (2005b). Language Workbenches: The Killer-App for Domain Specific Languages? Disponible en `http://www.martinfowler.com/articles/languageWorkbench.html#ASimpleExampleOfLanguageOrientedProgramming`.

[Fowler, 2005c] Fowler, M. (2005c). The New Methodology. Disponible en `http://www.martinfowler.com/articles/newMethodology.html`.

[Fowler et al., 1999] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

[Frankel, 2004a] Frankel, D. (2004a). MDA and the Object Technology Barrier. *The MDA Journal*, pages 29–46.

[Frankel, 2004b] Frankel, D. (2004b). The MDA Marketing Message and the MDA Reality. *The MDA Journal*, pages 107–112.

[Frankel, 2003] Frankel, D. S. (2003). *Model Driven Architecture. Applyng MDA to Enterprise Computing*. OMG Press.

[FreeMarker, 2006] FreeMarker (2006). Freemarker home page. `http://freemarker.sourceforge.net/`.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition.

[Gilb, 1988] Gilb, T. (1988). *Principles of Software Engineering Management*. Addison Wesley.

[Gorlen and Plexico, 1990] Gorlen, K. E. and Plexico, P. S. (1990). *Data Abstraction and Object-oriented Programming in C++*. John Wiley and Sons.

[Gosling et al., 1996] Gosling, J., Joy, B., and Seele, G. (1996). *The Java Language Specification*. Addison-Wesley.

[Gotel and Finkelstein, 1994] Gotel, O. and Finkelstein, A. (1994). An analysis of the requirements traceability problem. *Proceedings of the IEEE International Conference on Requirements Engineering*, pages 94–102.

[Greenfield, 2004] Greenfield, J. (2004). Microsoft and domain specific languages, reprise. `http://blogs.msdn.com/jackgr/archive/2004/12/20/327726.aspx`.

[Greenfield, 2005] Greenfield, J. (2005). Putting the cart where it belongs. `http://blogs.msdn.com/jackgr/archive/2005/01/03/346035.aspx`.

[Greenfield et al., 2004] Greenfield, J., Short, K., Cook, S., Kent, S., and Crupi, J. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 1st edition.

[Guttman, 2004] Guttman, M. (2004). Microsoft Should Not Compete With MDA. *The MDA Journal*, pages 95–103.

[Harel, 1987] Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274.

[Harel and Politi, 1998] Harel, D. and Politi, M. (1998). *Modeling Reactive Systems With Statecharts : The Statemate Approach*. McGraw-Hill Companies, 1st edition.

[Harold and Means, 2004] Harold, E. R. and Means, W. S. (2004). *XML in a Nutshell*. O'Reilly Media, 3rd edition.

[Hartmann et al., 2004] Hartmann, J., Vieira, M., Foster, H., and Ruder, A. (2004). UML-based Test Generation and Execution. *Siemens Corporate Research, Inc*.

[Hebach, 2005] Hebach, M. (2005). MDA with QVT. Borland Together 2006 Presentation.

[Heineman and Councill, 2001] Heineman, G. T. and Councill, W. T. (2001). *Component Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Professional, 1st edition.

[Hennicker and Koch, 2001] Hennicker, R. and Koch, N. (2001). Modeling the User Interface of Web Applications with UML. *Proceedings of Practical UML-Based Rigorous Development Methods, Workshop of the pUML-Group at the UML01*.

[Herrington, 2003] Herrington, J. (2003). *Code Generation in Action*. Manning Publications.

[Horrocks, 1999] Horrocks, I. (1999). *Constructing the User Interface with Statecharts*. Addison-Wesley Professional.

[Howe, 2002] Howe, D. (2002). The Free Online Dictionary. "Virtual Machine". `www.foldoc.org`.

[Hudak, 1989] Hudak, P. (1989). *Conception, evolution, and application of functional programming languages*. ACM Computing Surveys.

[IBM, 1970] IBM (1970). *Data Processing Techniques*. IBM. Disponible en `http://www.fh-jena.de/~kleine/history/software/IBM-FlowchartingTechniques-GC20-8152-1.pdf`.

[IBM, 2006] IBM (2006). Rational software architect. `http://www-306.ibm.com/software/rational/`.

[INRIA Nantes, 2005] INRIA Nantes, A. . (2005). *ATL: Atlas Transformation Language. Specification of the ATL Virtual Machine v 0.1*.

[INRIA Nantes, 2006] INRIA Nantes, A. . (2006). *ATL: Atlas Transformation Language. ATL User Manual. Version 0.7*.

[ISO, 2000] ISO (2000). *ISO 9000:2000, Quality management systems - Fundamentals and vocabulary.* International Organization for Standardization.

[ITU, 1992] ITU (1992). *ITU-T. Specification and Description Language (SDL). ITU-T Recommendation Z.100*. International Telecommunications Union.

[ITU, 2000] ITU (2000). *ITU-T. SDL combined with UML. Z.109*. International Telecommunications Union.

[Izquierdo Castanedo, 2002] Izquierdo Castanedo, R. (2002). RDM: Arquitectura Software para el Modelado de Dominios en Sistemas Informáticos. Master's thesis, Universidad de Oviedo.

[Jacobson et al., 1999] Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley.

[Jacobson et al., 1994] Jacobson, I., Christerson, M., and Constantine, L. L. (1994). The OOSE method: a use-case-driven approach. *Sigs Advances In Object Technology Series. SIGS Publications, Inc*, pages 247–270.

[JBoss, 2006] JBoss (2006). Jboss jbpm. `http://www.jboss.org/products/jbpm`.

[Johnson et al., 2005] Johnson, R., Hoeller, J., Arendsen, A., Risberg, T., and Sampaleanu, C. (2005). *Professional Java Development with the Spring Framework*. Wrox.

[Jouault and Kurtev, 2006] Jouault, F. and Kurtev, I. (2006). On the architecturalalignment of atl and qvt. *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 06). ACM Press, Dijon, France, chapter Model transformation*.

[Juan Fuente et al., 2006] Juan Fuente, A. A., Cueva Lovelle, J. M., Ortín Soler, F., Izquierdo Castanedo, R., Luengo Díez, M. C., and Labra Gayo, J. E. (2006). *Tablas de Símbolos*. Universidad de Oviedo. Departamento de Informática.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997.). Aspect oriented programming. *Proceedings of ECOOP'97 Conference. Finland.*

[Kleppe et al., 2003] Kleppe, A., Warmer, J., and Bast, W. (2003). *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley.

[Kruchten, 2000] Kruchten, P. (2000). *The Rational Unified Process. An Introduction*. Addison Wesley, 2nd edition.

[Larman, 2003] Larman, C. (2003). *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley Professional, 1st edition.

[Larman, 2004] Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Addison Wesley Professional, third edition.

[Lowy, 2003] Lowy, J. (2003). *Programming .NET Components*. O'Reilly, 1st edition.

[Magic, 2006] Magic, N. (2006). Magic Draw Home Site. `http://www.magicdraw.com/`.

[Mann, 2005] Mann, K. (2005). *JavaServer Faces in Action*. Manning Publications.

[May and Zimmer, 1996] May, E. and Zimmer, B. (1996). The evolutionary development model for software. *Hewlett-Packard Journal*, 47(4):39–45.

[McCall et al., 1977] McCall, J., Richards, P., and Walters, G. (1977). Factors in software quality. *Rome Air Development Center, RADC TR-77-369, 1977*.

[McIver and Conway, 1996] McIver, L. and Conway, D. (1996). Seven deadly sins of introductory programming language design. *In Proceedings, 1996 Conference on Software Engineering: Education and Practice. IEEE Computing Society Press, Los Alamitos, CA, USA*, pages 309–316.

[McNeile and Simons, 2004] McNeile, A. and Simons, N. (2004). Methods of Behavior Modelling: A Commentary on Behaviour Modelling Techniques for MDA. *Metamaxim*.

[Mellor et al., 2004] Mellor, B. S. J., Scott, K., Uhl, A., and Weise, D. (2004). *MDA Distilled: Principles of Model-Driven Architecture*. Addison Wesley.

[Mellor, 2004] Mellor, S. J. (2004). Agile MDA. *The MDA Journal*, pages 144–160.

[Mellor and Balcer, 2002] Mellor, S. J. and Balcer, M. J. (2002). *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley.

[Meyer, 1991] Meyer, B. (1991). *Eiffel : The Language*. Prentice Hall.

[Meyer, 2000] Meyer, B. (2000). *Object-Oriented Software Construction*. Computer Science. Prentice-Hall, 2nd edition.

[Microsoft, 2006a] Microsoft (2006a). Domain-specific language tools version 1. `http://msdn2.microsoft.com/en-us/teamsystem/bb217251.aspx`.

[Microsoft, 2006b] Microsoft (2006b). The EFx Architectural-Guidance Software Factory. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/efxagsftftr.asp`.

[Miller and Mukerji, 2003] Miller, J. and Mukerji, J. (2003). *MDA Guide Version 1.0.1*. Object Management Group.

[Mok and Paper, 2002] Mok, W. Y. and Paper, D. (2002). *Journal of Database Management*, 13(3):17–34.

[Moore, 1956] Moore, E. F. (1956). Gedanken-experiments on Sequential Machines. *Automata Studies, Annals of Mathematical Studies. Princeton University Press, Princeton, N. J*, (34):129–153.

[Myers et al., 2004] Myers, G. J., Sandler, C., Badgett, T., and Thomas, T. M. (2004). *The Art of Software Testing*. John Wiley & Sons, 2nd edition.

[NetBeans, 2006a] NetBeans (2006a). Metadata Repository (MDR). `http://mdr.netbeans.org/`.

[NetBeans, 2006b] NetBeans (2006b). NetBeans IDE. `http://www.netbeans.org/`.

[OASIS, 2006] OASIS (2006). Oasis web services business process execution language (wsbpel) tc. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`.

[Odell et al., 2000] Odell, J., Parunak, H. V. D., and Bauer, B. (2000). Extending UML for Agents. *Proceedings of the 2nd International Conference Workshop on Agent-Oriented Information Systems, AOIS'00*, pages 3–17.

[OMG, 1997] OMG (1997). *Meta Object Facility (MOF) Specification version 1.1. OMG Document ad/97-08-14*. Object Management Group.

[OMG, 2001] OMG (2001). *UML Action Semantics Revised Final Submission. OMG document ad/01-08-04*. Object Management Group.

[OMG, 2002a] OMG (2002a). *Meta Object Facility (MOF) Specification. Version 1.4*. Object Management Group.

[OMG, 2002b] OMG (2002b). *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*. Object Management Group.

[OMG, 2002c] OMG (2002c). *UML Profile for CORBA Specification Version 1.0*. Object Management Group.

[OMG, 2003] OMG (2003). *Unified Modeling Language Specification. Version 1.5*. Object Management Group.

[OMG, 2004a] OMG (2004a). *Common Object Request Broker Architecture (CORBA/IIOP). Version 3.0.3*. Object Management Group.

[OMG, 2004b] OMG (2004b). *Enterprise Collaboration Architecture (ECA) Specification. Version 1.0. formal/04-02-01*. Object Management Group.

[OMG, 2004c] OMG (2004c). *Human-Usable Textual Notation (HUTN) Specification version 1.0*. Object Management Group.

[OMG, 2004] OMG (2004). *MOF Model to Text Transformation Language . Request For Proposal. OMG Document: ad/2004-04-07*. Object Management Group, april edition.

[OMG, 2004a] OMG (2004a). *UML Profile and Interchange Models for Enterprise Application Integration (EAI) Specification. OMG Formal Specification. formal/04-03-26*. Object Management Group.

[OMG, 2004b] OMG (2004b). *Unified Modeling Language: Superstructure. Version 2.0*. Object Management Group.

[OMG, 2005a] OMG (2005a). *Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification. Final Adopted Specification*. Object Management Group.

[OMG, 2005b] OMG (2005b). *MOF 2.0/XMI Mapping Specification, v2.1*. Object Management Group.

[OMG, 2005c] OMG (2005c). *MOF 2.0/XMI Mapping Specification, v2.1. Formal/05-09-01*. Object Management Group.

[OMG, 2005d] OMG (2005d). *MOF QVT. Final Adopted Specification. ptc/05-11-01*. Object Mangement Group.

[OMG, 2005e] OMG (2005e). *OCL 2.0 Specification. Version 2.0*. Object Management Group.

[OMG, 2005f] OMG (2005f). *Revised submission for MOF 2.0 Query/View/Transformation RFP (ad/2002-04-10)*. Object Management Group.

[OMG, 2005g] OMG (2005g). *Semantics of a Foundational Subset for Executable UML Models. Request For Proposal. OMG Document: ad/2005-04-02*. Object Management Group.

[OMG, 2005h] OMG (2005h). *UML Profile for CORBA Components. OMG Available Specification. Version 1.0. formal/05-07-06*. Object Management Group.

[OMG, 2005i] OMG (2005i). *UML Testing Profile. Version 1.0. formal/05-07-07*. Object Management Group.

[OMG, 2005j] OMG (2005j). *Unified Modeling Language: Diagram Interchange*. Object Management Group.

[OMG, 2005k] OMG (2005k). *Unified Modeling Language: Infrastructure. Version 2.0*. Object Management Group.

[OMG, 2006a] OMG (2006a). *Business Process Modeling Notation Specification. Final Adopted Specification version 1.0. dtc/06-02-01*. Object Management Group.

[OMG, 2006b] OMG (2006b). *Meta Object Facility (MOF) Core Specification. Version 2.0. Formal/06-01-01*. Object Management Group.

[OMG, 2006a] OMG (2006a). *MOF Models to Text Transformation. Final adopted specification. ptc/06-11-01*. Object Management Group.

[OMG, 2006b] OMG (2006b). Object Management Group. UML Resource Page. `http://www.uml.org/`.

[OMG, 2006] OMG (2006). *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*. Object Management Group.

[openArchitectureWare, 2006] openArchitectureWare (2006). openArchitectureWare (oAW) Home Site. `http://www.openarchitectureware.org/`.

[Ortín Soler, 2001] Ortín Soler, F. (2001). Sistema Computacional de Programación Flexible Diseñado sobre una Máquina Abstracta Reflectiva no Restrictiva. Master's thesis, Universidad de Oviedo.

[Ortín Soler et al., 2004] Ortín Soler, F., Cueva Lovelle, J. M., Luengo Díez, M. C., Juan Fuente, A. A., Labra Gayo, J. E., and Izquierdo Castanedo, R. (2004). *Análisis Semántico en Procesadores de Lenguaje*. Cuaderno Didáctico Nº 38. Servitec.

[Padrón Lorenzo et al., 2005] Padrón Lorenzo, J., García Luna, J., Sánchez Rebull, E., and Estévez García, A. (2005). Implementación de un motor de transformaciones con soporte mof 2.0 qvt. *II Taller sobre Desarrollo Dirigido por Modelos. MDA y Aplicaciones. DSDM '05*.

[Padrón Lorenzo, 2004] Padrón Lorenzo, J. E. G. A. R. G. J. G. L. F. (2004). BOA, un framework MDA de alta productividad. *I Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones (DSDM'04)*.

[Peterson, 1981] Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. Prentice Hall.

[Popma, 2003] Popma, R. (2003). JET Tutorial. *Eclipse Corner*. `http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html`.

[Pressman, 1987] Pressman, R. S. (1987). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2nd edition.

[Projtech-Technology, 2006] Projtech-Technology (2006). Projtech-Technology. Executable UML. `http://www.projtech.com/pubs/xuml.html`.

[Raistrick et al., 2004] Raistrick, C., Francis, P., Wright, J., Carter, C., and Wilkie, I. (2004). *Model Driven Architecture with Executable UML*. Cambridge University Press.

[Reeves, 1992] Reeves, J. M. (1992). What is software design? *C++ Journal*. Disponible en `http://www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm`.

[Richard and Ledgard, 1977] Richard, F. and Ledgard, H. F. (1977). A reminder for language designers. *ACM SIGPLAN Notices, Vol. 12 No. 12*, pages 73–82.

[Royce, 1970] Royce, W. W. (1970). Managing the Development of Large Software Systems. *IEEE Wescon*, pages 1–9.

[Rumbaugh et al., 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen., W. (1991). *Object-Oriented Modeling and Design*. Prentice Hall.

[Rumbaugh et al., 1998] Rumbaugh, J., Jacobson, I., and Booch, G. (1998). *The Unified Modeling Language Reference Manual*. Addison Wesley.

[Samek, 2002] Samek, M. (2002). *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*. CMP Books.

[Schwaber and Beedle, 2001] Schwaber, K. and Beedle, M. (2001). *Agile Software Development with SCRUM*. Prentice Hall, 1st edition.

[Selic, 1998] Selic, B. (1998). Using UML for Modeling Complex Real-Time Systems. *Proceedings of Languages, Compilers, and Tools for Embedded Systems: ACM SIGPLAN Workshop LCTES'98*.

[Selic et al., 1994] Selic, B., Gullekson, G., and Ward, P. T. (1994). *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1st edition.

[Shlaer and Mellor, 1988] Shlaer, S. and Mellor, S. J. (1988). *Object-oriented Systems Analysis: Modeling the World in Data*. Prentice Hall.

[Shlaer and Mellor, 1991] Shlaer, S. and Mellor, S. J. (1991). *Object Lifecycles: Modeling the World in States*. Yourdon Press Computing Series, 1st edition.

[Soley and *OMG Staff Strategy Group*, 2000] Soley, R. and *OMG Staff Strategy Group* (2000). *Model Driven Architecture. White Paper. Draft 3.2*. Object Management Group.

[Sommerville, 2001] Sommerville, I. (2001). *Software Engineering*. Addison-Wesley, 6th edition.

[Stepanian, 2004] Stepanian, L. S. (2004). Solving the Requirements Traceability Problem: A Comparison of Two Pre-Requirements Specification Traceability Enablers. *Computer Systems Research Group. Department of Computer Science. University of Toronto*. Disponible en `http://www.cs.toronto.edu/~levon/projects/RTSurvey/RT.pdf`.

[Sun, 1997] Sun (1997). *Java Core Reflection. API and Specification*. Sun Microsystems.

[Sun, 2001] Sun (2001). *JSR 26: UML/EJB Mapping Specification*. Sun Microsystems.

[Sun, 2002] Sun (2002). *Java Metadata Interface(JMI) Specification. JSR 040. Version 1.0. Final Specification*. Sun Microsystems.

[Sun, 2003] Sun (2003). Programming With Assertions. `http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html`.

[Sun, 2004] Sun (2004). Java Doc 5.0 Tool. `http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/index.html`.

[Sun, 2005] Sun (2005). *JSR-000244 JavaTM Platform, Enterprise Edition 5 Specification*. Sun Microsystems.

[Sun, 2006] Sun (2006). *JSR 220: Enterprise JavaBeans,Version 3.0. EJB 3.0 Simplified API*. Disponible en `http://java.sun.com/products/ejb/docs.html`.

[Sunye et al., 2001] Sunye, G., Pennaneac'h, F., Ho, W.-M., Guennec, A. L., and Jézéquel, J.-M. (2001). Using UML Action Semantics for Executable Modeling and Beyond. *In Advanced Information Systems Engineering. 13th International Conference, CAiSE 2001*.

[Tatroe et al., 2006] Tatroe, K., Lerdorf, R., and MacIntyre, P. (2006). *Programming PHP*. O'Reilly Media, 2nd edition.

[Thai and Lam, 2002] Thai, T. L. and Lam, H. (2002). *.NET Framework Essentials*. O'Reilly, second edition.

[Thomas, 2003] Thomas, D. (2003). UML - Unified or Universal Modeling Language? *Journal of Object Technology*, 2(1):7–12.

[Turk et al., 2002] Turk, D., France., R., and Rumpe, B. (2002). Limitations of Agile Software Processes. *Proceedings of Third International Conference on eXtreme Programming and Agile Processes in Software Engineering*.

[Tyrrell, 2001] Tyrrell, S. (2001). The many dimensions of the software process. *ACM Crossroads*.

[Vanderburg, 2005] Vanderburg, G. (2005). A simple model of agile software processes – or – extreme programming annealed. *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 40(10):539–545.

[Varro and Pataricza, 2003] Varro, D. and Pataricza, A. (2003). Vpm: A visual, precise and multilevel metamodeling framework for describing mathematical domains and uml. *Journal of Software and Systems Modeling*, pages 187–210.

[VIATRA, 2006] VIATRA (2006). The VIATRA2 Model Transformation Framework. `http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html`.

[W3C, 2004] W3C (2004). *XML Schema Part 0: Primer Second Edition*. World Wide Web Consortium. Disponible en `http://www.w3.org/TR/xmlschema-0/`.

[Wand and Friedman, 1988] Wand, M. and Friedman, D. P. (1988). The mystery of the tower revealed: A nonreflective description of the reflective tower. *Higher-Order and Symbolic Computation. Springer Netherlands*, 1(1):11–38.

[Warmer and Kleppe, 2003] Warmer, J. and Kleppe, A. (2003). *Object Constraint Language, The: Getting Your Models Ready for MDA*. 2nd edition.

[Wikipedia, 2006] Wikipedia (2006). Template processor definition. `http://en.wikipedia.org/wiki/Template_processor`.

[Wilkie et al., 2002] Wilkie, I. T., King, A., Clarke, M., Weaver, C., Raistrick, C., and Francis, P. (2002). UML ASL Reference Guide, ASL Language Level 2.5 Revision D. *Kennedy Carter Ltd*.

[Wilkie and Mellor, 1999] Wilkie, I. T. and Mellor, S. J. (1999). A Mapping from Shlaer-Mellor to UML. *Kennedy Carter Ltd/Project Technology*.

ÍNDICE ALFABÉTICO