# Desarrollo de software dirigido por modelos

Francisco Durán Muñoz
Javier Troya Castilla
Antonio Vallecillo Moreno

PID_00184466

UOC
Universitat Oberta de Catalunya

# Índice

# Introducción

El desarrollo de software dirigido por modelos (denominado MDD por su acrónimo en inglés, *model-driven development*) es una propuesta para el desarrollo de software en la que se atribuye a los modelos el papel principal, frente a las propuestas tradicionales basadas en lenguajes de programación y plataformas de objetos y componentes software.

El propósito de MDD es tratar de reducir los costes y tiempos de desarrollo de las aplicaciones software y mejorar la calidad de los sistemas que se construyen, con independencia de la plataforma en la que el software será ejecutado y garantizando las inversiones empresariales frente a la rápida evolución de la tecnología.

Los pilares básicos sobre los que se apoya MDD son los modelos, las transformaciones entre modelos y los lenguajes específicos de dominio. Estos son precisamente los temas que se cubren en este módulo. También discutiremos las ventajas que ofrece MDD, así como los principales retos y riesgos que implica su adopción en la actualidad.

# Objetivos

Este módulo introduce los principales conceptos, mecanismos y procesos relacionados con el desarrollo de software dirigido por modelos, centrándose sobre todo en una propuesta concreta como es *model-driven architecture* (MDA®) de la OMG™ (Object Management Group).

Más concretamente, los objetivos que persigue este módulo son los siguientes:

**1.** Dar a conocer el concepto de desarrollo de software dirigido por modelos, junto con sus principales características y mecanismos. También se presentan las ventajas e inconvenientes que su uso plantea para el diseño y desarrollo de sistemas software.

**2.** Introducir el metamodelado como una técnica esencial dentro de MDA para definir lenguajes de modelado específicos de dominio.

**3.** Presentar los lenguajes que ofrece la OMG para el desarrollo de lenguajes específicos de dominio, y en particular OCL y las extensiones de UML mediante perfiles UML.

**4.** Introducir los conceptos y mecanismos relacionados con las transformaciones de modelos, y en particular los que ofrece el lenguaje de transformación ATL.

**5.** Presentar los principales estándares relacionados con el desarrollo de software dirigido por modelos.

# 1. Desarrollo dirigido por modelos (MDA y MDE)

## 1.1. Introducción

El desarrollo de software dirigido por modelos surge como respuesta a los principales problemas que actualmente tienen las compañías de desarrollo de software: por un lado, para gestionar la creciente complejidad de los sistemas que construyen y mantienen, y por otro para adaptarse a la rápida evolución de las tecnologías software.

En términos generales, son varios los factores que hacen de MDD, y en particular de MDA (*model-driven architecture*), la propuesta adecuada para abordar estos problemas por parte de cualquier compañía que desee ser competitiva en el sector del desarrollo de software.

En primer lugar, se usan **modelos** para representar tanto los sistemas como los propios artefactos software. Cada modelo trata un aspecto del sistema, que puede ser especificado a un nivel más elevado de abstracción y de forma independiente de la tecnología utilizada. Este hecho permite que la evolución de las plataformas tecnológicas sea independiente del propio sistema, disminuyendo así sus dependencias.

En segundo lugar, se consigue también proteger gran parte de la inversión que se realiza en la informatización de un sistema, pues los modelos son los verdaderos artífices de su funcionamiento final y, por tanto, no será necesario empezar desde cero cada vez que se plantee un nuevo proyecto o se desee realizar algún tipo de mantenimiento sobre el producto.

En tercer lugar, en MDA los modelos constituyen la propia documentación del sistema, disminuyendo considerablemente el coste asociado y aumentando su mantenibilidad, puesto que la implementación se realiza de forma automatizada a partir de la propia documentación.

Este apartado presenta los principales conceptos que intervienen en el desarrollo de software dirigido por modelos, las notaciones utilizadas, y las técnicas y herramientas más comunes que se utilizan en este ámbito.

## 1.2. Conceptos básicos

### 1.2.1. Abstracción

A la hora de diseñar y desarrollar cualquier aplicación de software es preciso contar con lenguajes que permitan representar tanto la estructura como el comportamiento de los sistemas de forma precisa, concreta, correcta y adecuada. Para ello es fundamental introducir tres conceptos clave: **abstracción**, **expresividad** y **complejidad**.

**Abstraer** significa destacar una serie de características esenciales de un sistema u objeto, desde un determinado punto de vista, ignorando aquellas otras características que no son relevantes desde esa perspectiva.

**Consulta recomendada**

**E. W. Dijkstra** (1972). "The Humble Programmer". *ACM Turing Lecture*. http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html

En palabras de Edsger W. Dijkstra:

> "ser abstracto no significa en absoluto ser impreciso [...] El principal propósito de la abstracción es definir un nivel semántico en el que poder ser totalmente preciso".

Es por ello por lo que es vital determinar el punto de vista desde donde plantear el problema, que identifica de forma precisa tanto los conceptos que se deben tener en cuenta a la hora de plantear el problema y su solución, como aquellos aspectos que podemos ignorar por no ser relevantes desde ese punto de vista.

**Lectura recomendada**

Es interesante el análisis que se hace de este tema en el artículo de Jean Bezivin (2005) "The Unification Power of Models". *SoSym* (vol. 2, núm. 4, págs. 171-188).

Pensemos por ejemplo en los distintos lenguajes y notaciones que se utilizan a la hora de describir una ciudad. Así, tenemos el plano del metro, que muestra las paradas y las conexiones entre ellas, pero ignora las distancias entre las estaciones por ser irrelevantes en ese nivel de detalle. Por otro lado está el callejero, que muestra las calles y las distancias relativas entre ellos, pero ignora otros datos, como por ejemplo la orografía de la ciudad, es decir, las inclinaciones de las calles y sus desniveles (algo importante si estamos interesados en planificar una carrera urbana, por ejemplo). Para ello está el plano topográfico, que muestra los niveles de altitud sin fijarse en las calles concretas o en los edificios. Es posible también disponer hoy en día, por ejemplo, de mapas de ruido de una ciudad, con el grado de contaminación acústica a distintas horas del día y de la noche. Cada uno de estos planos usa una notación diferente, la más adecuada para describir los conceptos que se pretenden expresar, y cada uno de ellos se centra en unos aspectos esenciales, ignorando otros. Por supuesto, sería posible tener un plano con absolutamente toda la información, pero, al igual que la realidad, sería demasiado complejo para entenderlo, manejarlo y razonar sobre él.

### 1.2.2. Expresividad

Otro de los factores importantes a considerar es el de la expresividad del lenguaje que usemos (sea de programación, de modelado, etc.) para diseñar el sistema, describir los requisitos, implementar la solución, etc.

La **expresividad** es la capacidad de un lenguaje de poder describir determinados elementos y las características de un sistema de forma adecuada, completa, concisa y sin ambigüedad.

Siempre es muy importante contar con el mejor lenguaje para describir lo que queremos expresar. Es por ello por lo que existen en la actualidad numerosos lenguajes de programación, cada uno de ellos indicado para expresar determinados aspectos:

- declarativos y funcionales para especificar los tipos de datos y sus algoritmos,
- orientados a objetos para describir comportamiento de sistemas software reactivos,
- lenguajes de consulta para acceder a bases de datos,
- de programación web para desarrollar aplicaciones sobre Internet, etc.

Esto es similar a lo que ocurre en otras disciplinas ingenieriles, en donde se cuenta con lenguajes visuales para diseñar los planos de un edificio, fórmulas matemáticas para calcular las cargas de las estructuras, textos para describir los requisitos de seguridad, etc.

### 1.2.3. Complejidad

La complejidad es otro de los conceptos clave a tener en cuenta a la hora de diseñar y desarrollar cualquier sistema software. Una de las definiciones de complejidad más simples y fáciles de entender dice:

Un sistema es **complejo** cuando no es posible que una sola persona sea capaz de comprenderlo, manejarlo y razonar sobre él.

Fred Brooks, en su famoso artículo "No Silver Bullet", distingue entre dos tipos de complejidad: esencial y accidental.

La **complejidad esencial** de un problema o de un sistema es la que se deriva de su propia naturaleza, la que impide la existencia de soluciones "simples".

Las aplicaciones para gestionar un aeropuerto o una central nuclear son intrínsecamente complejas por serlo los sistemas a gestionar.

**Consulta recomendada**

**Fred Brooks** (1986). "No Silver Bullet: Essence and Accidents in Software Engineering". *Proceedings of the IFIP 10th World Computing Conference* 1069-1076.

(1987). *Computer* (vol. 20, núm. 4, abril, págs. 10-19).

(1995). "Mythical Man-Month, Silver Bullet Refired" (cap. 17). Addison-Wesley.

La **complejidad accidental**, sin embargo, deriva de la adecuación de los lenguajes, algoritmos y herramientas que se utilizan para plantear la solución a un problema.

**Ved también**

Véase lo estudiado en la asignatura *Diseño de estructuras de datos* del grado de Ingeniería Informática.

En general, cada tipo de lenguaje de programación ofrece una serie de mecanismos que lo hacen más (o menos) apropiado al tipo de problema que tratamos de resolver y al tipo de sistema que pretendemos desarrollar.

Apretar un tornillo se convierte en una tarea compleja si solo contamos con un martillo. De igual forma, diseñar aplicaciones web se convierte en una tarea muy difícil si tratamos de realizarlas usando lenguajes como ensamblador o COBOL.

Es importante señalar que la expresividad de un lenguaje influye directamente en la complejidad accidental de la solución, aunque no es el único factor a tener en cuenta.

Si lo que queremos es ordenar un vector de números, podemos utilizar diferentes algoritmos, cada uno más apropiado dependiendo de la naturaleza y previo estado de los datos (aunque todos estén expresados con el mismo lenguaje de programación). Así, el método de ordenación por inserción es el más adecuado si los datos están ya casi ordenados previamente, y Quicksort es el más eficiente si están muy desordenados.

### 1.2.4. Reutilización

Además de la abstracción como mecanismo para abordar la complejidad, otro de los objetivos de cualquier disciplina ingenieril es el de la reutilización. En vez de repetir la especificación de un código, una estructura o un conjunto de funciones, la reutilización permite definirlos una sola vez y utilizarlos varias, aunque, como muy acertadamente dice Clemens Szyperski:

**Ved también**

Estudiaremos con detalle la reutilización en el módulo "Desarrollo de software basado en reutilización".

"Reutilizar no significa solamente usar más de una vez, sino poder usar en diferentes contextos".

**Consulta recomendada**

**C. Szyperski**.(2002). *Component Software – Beyond Object-Oriented Programming*. (2.ª ed.). Addison-Wesley.

#### Mecanismos de reutilización

La búsqueda de mejores mecanismos de reutilización ha sido otra de las constantes en la evolución de la programación. Así, las macros y funciones de los lenguajes de programación permitieron reutilizar grupos de líneas de código, y los tipos abstractos fueron definidos para agrupar varias funciones en torno a ciertos tipos de datos. Los objetos permiten encapsular el estado y comportamiento de los elementos de una misma clase en unidades básicas. Sin embargo, pronto se vio que la granularidad de los objetos era insuficiente para abordar grandes sistemas distribuidos y aparecieron los componentes, que agrupaban un número de objetos relacionados entre sí para ofrecer una serie de servicios a través de interfaces bien definidas, con la propiedad de gestionar de forma conjunta su ciclo de vida y su despliegue (*deployment*). Basados en componentes, los marcos de trabajo (*frameworks*) supusieron un avance importante porque permitían la reutilización no solo de una serie de componentes, sino también de su arquitectura software. Finalmente, los servicios web han permitido cambiar el modelo de "instala-y-usa" de los componentes software por el de "invoca-y-usa", resolviendo el problema de la actualización de versiones locales, la particularización de componentes a plataformas hardware o software concretas, etc.

**Cita**

Un software *framework* es una aplicación parametrizable que proporciona un conjunto de librerías y servicios a través de un conjunto de interfaces. Más información en: **M. Fayad; D. C. Schmidt**. (1997). "Object-Oriented Application Frameworks". *Comms. of the ACM* (vol. 40, núm. 10).

Aunque los continuos avances en los lenguajes y mecanismos de programación son realmente significativos, vemos que aún no se han resuelto todos los problemas, y que los sistemas de información que se construyen siguen sin ofrecer soluciones de reutilización a gran escala. De hecho, una crítica muy

común entre los máximos responsables de las grandes compañías es su dependencia absoluta de determinadas plataformas hardware y software, de lenguajes de programación, de tecnologías concretas de bases de datos, etc. Ellos perciben una enorme volatilidad en las tecnologías hardware y software, que evolucionan a gran velocidad y dejan obsoletos sus sistemas de información en muy pocos años –aun cuando ni siquiera se ha amortizado la inversión realizada en ellos. Y la informática lo único que les ofrece ahora mismo es que actualicen todos sus sistemas y los sustituyan por otros nuevos, para lo cual hay que empezar casi desde cero a diseñarlos y después migrar todos sus datos y servicios a estas nuevas plataformas.

Uno de los principales problemas con estas migraciones, aparte del coste y tiempo que representan, es que vuelven a dejarles con un conjunto de programas y datos dependientes de una plataforma concreta, y a un nivel de abstracción que no permite su reutilización cuando dentro de unos años sea preciso actualizar de nuevo sus aplicaciones (y haya que comenzar los diseños casi desde el principio otra vez).

Otro problema importante que tienen estas empresas es el de la interoperabilidad con los sistemas de información de sus clientes y proveedores. La heterogeneidad actual representa un impedimento importante para el intercambio de datos, funciones y servicios entre aplicaciones. Una de las principales razones para estos problemas de interoperabilidad es el bajo nivel de detalle en el que se describen los sistemas, los servicios y sus interfaces. No es suficiente con describir las operaciones proporcionadas o el orden en el que hay que invocarlas, sino también es necesario tener en cuenta el modelo de negocio de ambas empresas, sus contextos legales y fiscales, los contratos de calidad de servicio, las responsabilidades por fallos en el servicio, etc.

Por eso sería conveniente describir los sistemas a un nivel de abstracción que permitiera la reutilización de sus datos y procesos de negocio, de forma independiente de los lenguajes de programación, la tecnología subyacente y las plataformas en donde se han de ejecutar los programas, y que además facilitase la interoperabilidad con otros sistemas externos.

### 1.2.5. Modelos

Uno de los términos clave en la filosofía del MDD es el concepto de **modelo**.

> De forma sencilla podríamos definir un modelo como una abstracción simplificada de un sistema o concepto del mundo real.

Sin embargo, esta no es la única definición que encontraremos en la literatura sobre el término "modelo". A modo de ejemplo, las siguientes citas muestran algunas de las acepciones más comunes de este concepto en nuestro contexto:

- Un modelo es una **descripción** de un sistema, o parte de este, escrito en un lenguaje bien definido (Warmer y Kleppe, 2003).

- Un modelo es una **representación** de una parte de la funcionalidad, estructura y/o comportamiento de un sistema (OMG, *Model driven architecture – A technical perspective*, 2001).

- Un modelo es una **descripción** o **especificación** de un sistema y su entorno definida para cierto propósito (OMG, MDA *Guide*, 2003).

- Un modelo **captura una vista** de un sistema físico, con un cierto propósito. El propósito determina lo que debe ser incluido en el modelo y lo que es irrelevante. Por tanto, el modelo describe aquellos aspectos del sistema físico que son relevantes al propósito del modelo, y al nivel de abstracción adecuado (OMG, UML *Superstructure*, 2010).

- Un modelo es un **conjunto de sentencias** sobre un sistema (Seidewitz, "What models mean", IEEE *Computer*, 2003). En esta referencia, Seidewitz entiende por sentencia una expresión booleana sobre el sistema, que puede estar representada tanto gráfica como textualmente.

- M es un modelo de S si M puede ser utilizado para **responder preguntas** acerca de S (D. T. Ross y M. Minsky, 1960).

**Ved también**

Las referencias bibliográficas mostradas aquí las podéis encontrar en el apartado de "Bibliografía".

**Web recomendada**

Es interesante también consultar lo que define la RAE como modelo: http://buscon.rae.es/draeI/SrvltConsulta?LEMA=modelo.

Basándonos en ellas, podemos concluir en la siguiente definición:

> Un **modelo** de un cierto <X> es una **especificación** o **descripción** de ese <X> desde un determinado punto de vista, expresado en un lenguaje bien definido y con un propósito determinado.

En esta definición, <X> representa el objeto o sistema que queremos modelar, y que puede ser tanto algo concreto como algo abstracto. De hecho, el modelo proporciona una "especificación" de <X> cuando se da la circunstancia de que <X> no existe todavía (por ejemplo, es el sistema a construir), mientras que proporciona una "descripción" cuando representa un sistema que ya existe en la realidad y que queremos modelar con algún propósito determinado[1].

(1)Por ejemplo, analizar alguna de sus características o propiedades, entender su funcionamiento, abstraer alguno de sus aspectos, etc.

### 1.2.6. Uso y utilidad de los modelos

Aunque no hay consenso sobre cuáles son las características que deben tener los modelos para ser considerados útiles y efectivos, una de las descripciones más acertadas es la de Bran Selic, uno de los fundadores del MDD y pionero en el uso de sus técnicas. Según él, los modelos deberían ser:

**Consulta recomendada**

Véase por ejemplo, la presentación que Bran Selic hizo en el congreso MODPROD en Suecia en 2011: "Abstraction Patterns in Model-Based Engineering".

- **Adecuados**: Construidos con un propósito concreto, desde un punto de vista determinado y dirigidos a un conjunto de usuarios bien definido.
- **Abstractos**: Enfatizan los aspectos importantes para su propósito a la vez que ocultan los aspectos irrelevantes.
- **Comprensibles**: Expresados en un lenguaje fácilmente entendible para sus usuarios.
- **Precisos**: Representan fielmente al objeto o sistema modelado.
- **Predictivos**: Pueden ser usados para responder preguntas sobre el modelo e inferir conclusiones correctas.
- **Rentables**: Han de ser más fáciles y baratos de construir y estudiar que el propio sistema.

Bran Selic también señala las principales funciones que los modelos deberían tener en el ámbito de la ingeniería del software:

- **Comprender** el sistema
  - La estructura, el comportamiento y cualquier otra característica relevante de un sistema y su entorno desde un punto de vista dado.
  - Separar adecuadamente cada uno de los aspectos, describiéndolos al nivel conceptual adecuado.
- Servir de **mecanismo de comunicación**
  - Con los distintos tipos de *stakeholder* del sistema (desarrolladores, usuarios finales, personal de soporte y mantenimiento, etc.).
  - Con las otras organizaciones (proveedores y clientes que necesitan comprender el sistema a la hora de interoperar con él).
- **Validar** el sistema y su diseño
  - Detectar errores, omisiones y anomalías en el diseño tan pronto como sea posible (cuanto antes se detecten, menos cuesta corregirlos).

***Stakeholder***

Persona o entidad que está implicada en la adquisición, desarrollo, explotación o mantenimiento de un sistema software.

  - Razonar sobre el sistema, infiriendo propiedades sobre su comportamiento (en caso de modelos ejecutables que puedan servir como prototipos).

  - Poder realizar análisis formales sobre el sistema.

- **Guiar** la implementación
  - Servir como "planos" para construir el sistema y que permitan guiar su implementación de una forma precisa y sin ambigüedades.

  - Generar, de la forma más automática posible, tanto el código final como todos los artefactos necesarios para implementar, configurar y desplegar el sistema.

Algo que diferencia la ingeniería del software del resto de las ingenierías es que en estas últimas el medio en el que se construyen los planos y modelos es muy diferente del medio en el que finalmente se construyen los edificios, puentes o aviones. La ventaja de que en la ingeniería del software el medio sea el mismo (los ordenadores) va a permitir que puedan definirse transformaciones automáticas que sean capaces de generar la implementación a partir de los modelos de alto nivel, algo más costoso en otras disciplinas. Es por ello por lo que en el caso de MDD (y en particular en MDA) lo que se persigue es que la generación de las implementaciones de un sistema sean lo más automatizadas posibles, algo que es factible gracias al uso de las transformaciones de modelos.

**Ved también**

Las transformaciones de modelos se estudian con más detalle en el apartado 4.

### 1.2.7. Metamodelos

Es importante señalar el hecho de que el lenguaje que se usa para describir el modelo debe estar bien definido y ofrecer un nivel de abstracción adecuado para expresar el modelo y para razonar sobre él. En este sentido, la idea compartida por todos los paradigmas englobados dentro del MDD es la conveniencia de utilizar para el modelado lenguajes de mayor nivel de abstracción que los lenguajes de programación, esto es, lenguajes que manejen conceptos más cercanos al dominio del problema, denominados lenguajes específicos de dominio (o DSL, *domain-specific language*). Estos lenguajes requieren a su vez una descripción precisa; aunque esta puede darse de diferentes maneras, en el mundo de MDD lo normal y más apropiado es definirlos como modelos también. Como veremos con más detalle en el apartado 3, el metamodelado es una estrategia muy utilizada en la actualidad para la definición de lenguajes de modelado.

**Ved también**

Los lenguajes específicos de dominio se estudian más adelante en el apartado 3.

Un **metamodelo** es un modelo que especifica los conceptos de un lenguaje, las relaciones entre ellos y las reglas estructurales que restringen los posibles elementos de los modelos válidos, así como aquellas combinaciones entre elementos que respetan las reglas semánticas del dominio.

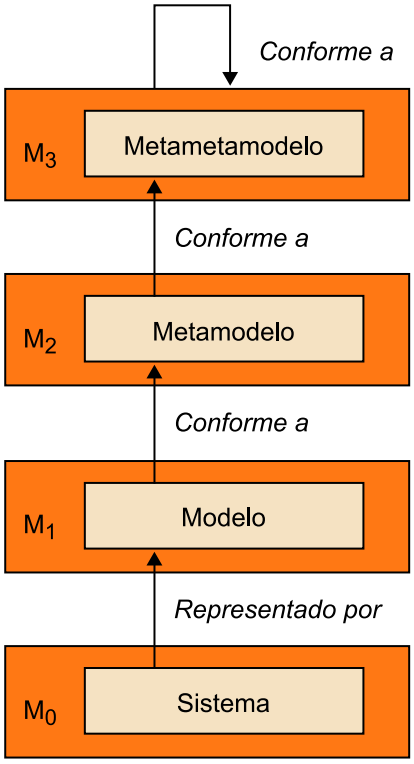
El metamodelo de UML es un modelo que contiene los elementos para describir modelos UML, como Package, Classifier, Class, Operation, Association, etc. El metamodelo de UML también define las relaciones entre estos conceptos, así como las restricciones de integridad de los modelos UML. Un ejemplo de estas restricciones son las que obligan a que las asociaciones solo puedan conectar *classifiers*, y no paquetes u operaciones.

De esta forma, cada modelo se escribe en el lenguaje que define su metamodelo (su lenguaje de modelado), quedando establecida la relación entre el modelo y su metamodelo por una relación de "conformidad" (y diremos que un modelo es "conforme a" un metamodelo).

**Nota**

Hay gente que suele decir que un modelo es una "instancia" de su metamodelo, pero eso es similar a decir que un programa es una instancia de su gramática, lo cual no es del todo correcto.

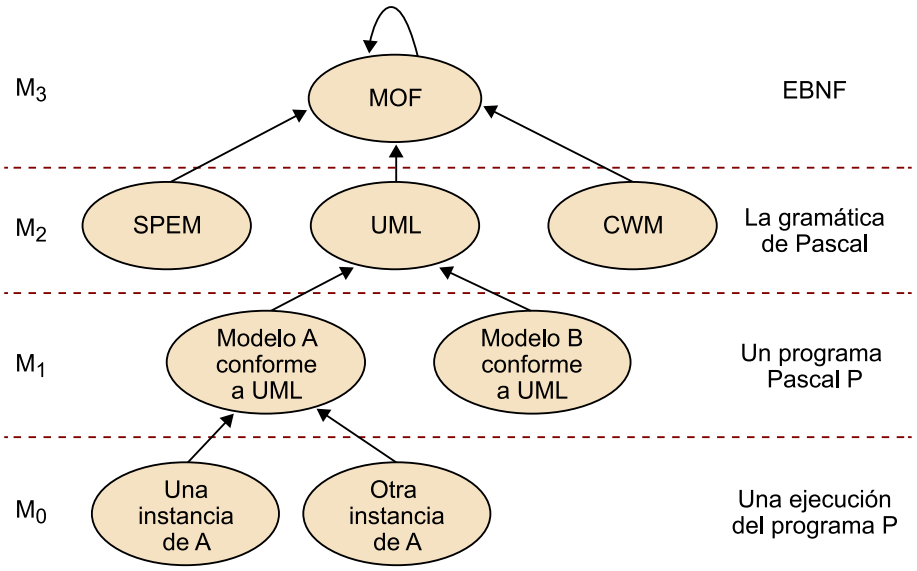
Figura 1. La organización en cuatro niveles de la OMG




Llegados a este punto, no olvidemos que los metamodelos son a su vez modelos y, por tanto, están escritos igualmente en el lenguaje definido por su metametamodelo. Según la propuesta de la OMG, el procedimiento recursivo de definir modelos conforme a otros modelos de mayor grado de abstracción acaba cuando se alcanza el nivel de metametamodelo, ya que los metametamodelos se dice que son conformes a ellos mismos, tal y como se ilustra en la figura 1.

La figura 2 representa una instanciación concreta de la arquitectura en cuatro niveles representada en la figura 1. En ella, MOF (*meta-object facility*) es el lenguaje de la OMG para describir metamodelos; y UML, SPEM y CWM (*common warehouse metamodel*) son lenguajes de modelado conformes a MOF (para describir sistemas, procesos de negocio y almacenes de datos, respectivamente). La figura 2 ilustra también la analogía existente entre esta organización en cuatro niveles y la jerarquía de niveles de los lenguajes tradicionales de programación (representada en el lateral derecho).

Figura 2. Ejemplo de la organización en cuatro niveles de la OMG




**Nota**

EBNF: Extended Backus-Naur Form, define una manera formal de describir la gramática de un lenguaje de programación.

### 1.2.8. Transformaciones de modelos en MDA

De particular importancia en MDA es la noción de transformación entre modelos.

Una **transformación de modelos** es el proceso de convertir un modelo de un sistema en otro modelo del mismo sistema. Asimismo, dícese de la especificación de dicho proceso.

En esencia, una transformación establece un **conjunto de reglas** que describen cómo un modelo expresado en un lenguaje origen puede ser transformado en un modelo en un lenguaje destino.

**Nota**

Como veremos en el apartado 4, las transformaciones de modelos son también modelos, lo cual permite realizar un tratamiento unificado de todos los artefactos que intervienen en cualquier proceso MDD.

Para definir una transformación entre modelos es necesario:

**1)** Seleccionar el tipo de transformación y el lenguaje de definición de transformaciones a utilizar y

**2)** seleccionar la herramienta que nos permita implementar los modelos y las transformaciones de forma automática.

## 1.3. Terminología

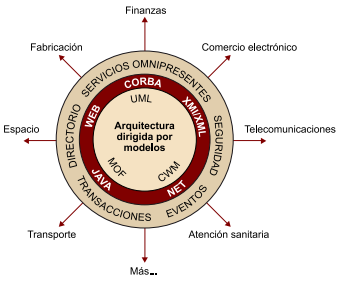
La comunidad utiliza en la actualidad un conjunto de acrónimos referidos a diferentes enfoques relacionados con la ingeniería del software que usa modelos como elementos clave de sus procesos: MDD, MBE, MDA, etc. Aunque algunos de ellos ya se han mencionado en secciones anteriores, en este apartado trataremos de aclarar estos conceptos y las diferencias entre ellos.

**MDD** (*model-driven development*) es un paradigma de desarrollo de software que utiliza modelos para diseñar los sistemas a distintos niveles de abstracción, y secuencias de transformaciones de modelos para generar unos modelos a partir de otros hasta generar el código final de las aplicaciones en las plataformas destino.

MDD tiene un enfoque claramente descendente, aumentando en cada fase el nivel de detalle y concreción de los modelos generados por las transformaciones.

En MDD el código se puede considerar como un modelo más, aunque algunos autores distinguen entre modelos y texto, y hablan no solo de transformaciones "modelo-a-modelo" sino también de transformaciones "modelo-a-texto". Para otros, la representación de un código fuente como un modelo o con líneas de código es una cuestión meramente sintáctica.

**MDA** (*model-driven architecture*) es la propuesta concreta de la OMG para implementar MDD, usando las notaciones, mecanismos y herramientas estándares definidos por esa organización.




MDA fue la primera de las propuestas de MDD, y la que consiguió hacer despegar el uso y adopción de los modelos como piezas clave del desarrollo de software. MDA, al igual que MDD, aboga por la separación de la especificación de la funcionalidad de un sistema independientemente de su implementación en cualquier plataforma tecnológica concreta, y en el uso de transformaciones de modelos para transformar unos modelos en otros hasta obtener las implementaciones finales.

Los estándares que la OMG ofrece para realizar MDA incluyen, entre otros:

- **UML** (*unified modeling language*) como lenguaje de modelado.
- **MOF** (*meta-object facility*) como lenguaje de metamodelado.

**Ved también**

UML se ha visto en la asignatura *Ingeniería del software* del grado de Ingeniería Informática. El resto de estándares se presentan en este módulo.

- **OCL** (*object constraint language*) como lenguaje de restricciones y consulta de modelos.

- **QVT** (*query-view-transformation*) como lenguaje de transformación de modelos.

- **XMI** (*XML metadata interchange*) como lenguaje de intercambio de información.

**Nota**

Metamodelar consiste en definir lenguajes de modelado.

**MDE** (*model-driven engineering*) es un paradigma dentro de la ingeniería del software que aboga por el uso de los modelos y las transformaciones entre ellas como piezas clave para dirigir todas las actividades relacionadas con la ingeniería del software.

En este sentido, MDE es un término más amplio que MDD, puesto que MDD se centra fundamentalmente en las labores de diseño y desarrollo de aplicaciones, mientras que MDE abarca también el resto de las actividades de ingeniería software: prototipado y simulación, análisis de prestaciones, migración de aplicaciones, reingeniería de sistemas heredados, interconexión e interoperabilidad de sistemas de información, etc.

En otras palabras, la diferencia entre MDD y MDE es la misma que hay entre desarrollo e ingeniería.

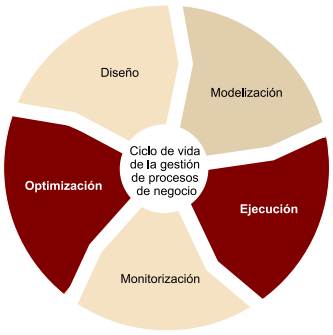
**MBE** (*model-based engineering*) es un término general que describe los enfoques dentro de la ingeniería del software que usan modelos en algunos de sus procesos o actividades.

En general, MBE es un término más amplio que MDE, y que lo engloba como paradigma. Aunque la diferencia es sutil, y principalmente de perspectiva, en MDE son los modelos, y las transformaciones entre ellos, los principales "motores" que guían los procesos de desarrollo, verificación, reingeniería, evolución o integración. Sin embargo, en MBE los modelos y las transformaciones no tienen por qué tener ese rol "motriz".

> Un proceso que use modelos para representar todos los artefactos software pero cuya ejecución se realice de forma completamente manual podría considerarse de MBE pero no de MDE.

Otra diferencia es que MDE suele centrarse en actividades de ingeniería del software, mientras que MBE puede referirse también a ingeniería de sistemas o industriales, cuando estos usan modelos para representar sus artefactos o procesos.

**BPM** (*business process modeling*) es una rama del *model-based engineering* que se centra en el modelado de los procesos de negocio de una empresa u organización, de forma independiente de las plataformas y las tecnologías utilizadas.




Existen varias notaciones actualmente para expresar modelos de negocio, siendo las principales BPMN 2.0 (*business process model and notation*) y SPEM 2.0 (*software and systems process engineering metamodel specification*), ambas actualmente responsabilidad de la OMG. Los diagramas de actividad de UML también pueden usarse para describir este tipo de procesos, aunque de forma un poco más genérica.

**Webs recomendadas**

Puedes encontrar toda la información de BPMN 2.0 y de SPEM 2.0 en sus webs oficiales. Para BPMN 2.0 en http://www.bpmn.org y http://www.omg.org/spec/BPMN/2.0/ y para SPEM 2.0 en http://www.omg.org/spec/SPEM/2.0/.

Los modelos de negocio de alto nivel pueden usarse para múltiples funciones que cada vez están cobrando mayor relevancia en MDE, como son los de la ingeniería de procesos y la adquisición basada en modelos.

- Dentro de lo que se denomina "ingeniería de procesos de negocio", disponer de modelos de procesos permite realizar ciertos análisis muy útiles para estudiar sus prestaciones (*performance*), detectar cuellos de botella o incluso situaciones de bloqueos (*deadlocks*).

- Los modelos de los procesos de negocio permiten describir los requisitos del sistema que se pretende construir con un nivel de abstracción adecuado para servir como "pliegos de requisitos" en los procesos de adquisición de sistemas de información. En este contexto, una empresa o administración genera los modelos de negocio del sistema que pretende adquirir, y los proveedores usan esos modelos como base para desarrollar las aplicaciones en sus plataformas. Esta práctica ha dado lugar a lo que se conoce como *model-based acquisition* (MBA).

Además de los acrónimos mencionados hasta el momento, es posible encontrar muchos más en la literatura relacionada con la ingeniería del software dirigida por modelos. En esta sección se destacan dos: ADM y MDI.

*Architecture-driven modernization* (**ADM**) es una propuesta de la OMG para implementar prácticas de ingeniería inversa utilizando modelos.

**Web recomendada**

Información detallada sobre la propuesta ADM de la OMG, incluyendo los estándares relacionados, herramientas y casos de uso de éxito en empresas, puede consultarse en http://adm.omg.org/

El objetivo de ADM es extraer modelos a diferentes niveles de abstracción de un código o aplicación existente, con el ánimo de extraer la información independiente de los lenguajes y plataformas tecnológicas utilizadas. Contar con esa información va a permitir utilizar luego técnicas de MDE para analizar el

sistema, generarlo en plataformas diferentes, etc. La palabra *modernización* se debe al objetivo que perseguía al principio ADM, que no era sino el de migrar aplicaciones existentes hacia nuevas plataformas.

**Ejemplos**

El Departamento de Defensa americano se vio en la necesidad de migrar los programas de sus aviones de combate, que usaban microprocesadores de los años setenta y para los cuales no había recambios. Igualmente, la NASA en el 2002 usaba procesadores 8086 en muchos de sus equipos del Space Shuttle Discovery, y temía que un proceso de migración manual del software pudiera ocasionar una significante pérdida de calidad y unos costes desorbitados, aparte del tiempo y esfuerzo necesarios para repetir todas las pruebas requeridas para un software crítico como el de control de aviones de combate o aeronaves tripuladas. Incluso si la migración fuese un éxito, dentro de unos cuantos años volverían a verse en la misma situación –a menos que idearan una nueva forma de migrar sus sistemas. La solución vino de la idea de extraer automáticamente modelos independientes de la plataforma de sus programas, y definir transformaciones que "compilaran" esos modelos (independientes de la plataforma y de cualquier tecnología) en el código apropiado a los nuevos procesadores y sistemas a utilizar.

*Model driven-interoperability* (**MDI**) es una iniciativa para implementar mecanismos de interoperabilidad entre servicios, aplicaciones y sistemas usando modelos y técnicas de MBE.

Por **interoperabilidad** se entiende la habilidad de dos o más entidades, sistemas, herramientas, componentes o artefactos para intercambiar información y hacer un uso adecuado de la información intercambiada para trabajar de forma conjunta.

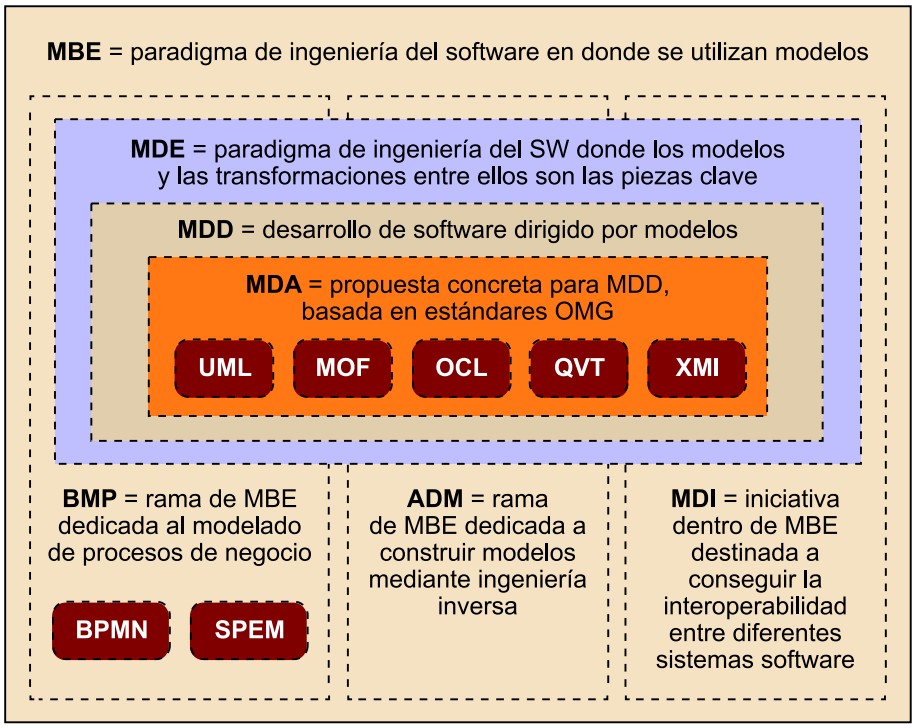
Aunque la interoperabilidad siempre ha sido un requisito esencial en cualquier sistema formado por dos o más componentes, este problema nunca había sido resuelto satisfactoriamente por las dificultades para resolver las diferencias entre los sistemas, la falta de acuerdo en los estándares a utilizar de forma conjunta y la falta de mecanismos y herramientas apropiadas.

**Mejoras de interoperabilidad**

La comunidad MDE ha comenzado a trabajar en estos temas, organizando talleres y conferencias internacionales donde expertos en diferentes materias pueden discutir sobre ellos y tratar de llegar a acuerdos. Es interesante el planteamiento del problema presentado por Jean Bezivin en su presentación sobre MDI. Por su parte, la OMG ha creado un grupo de trabajo (Model-Interchange Working Group) para resolver los problemas de interoperabilidad entre las distintas herramientas que trabajan con modelos, y en particular las herramientas UML. Los principales fabricantes de herramientas de modelado forman parte de este grupo, con el objetivo de aunar esfuerzos y ser compatibles entre sí.

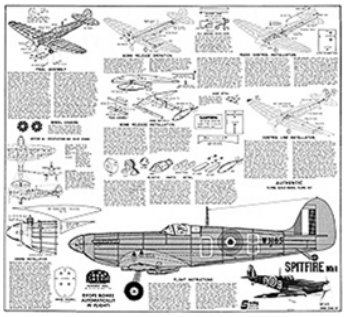
A modo ilustrativo, la figura 3 muestra todos los conceptos definidos en esta sección.

Figura 3. Terminología relacionada con el desarrollo de software dirigido por modelos

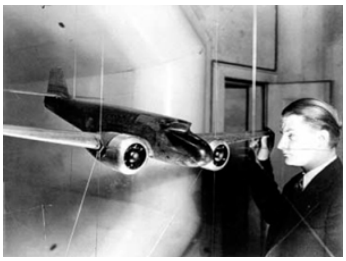



## 1.4. Los modelos como piezas clave de ingeniería

Pensemos un momento en cómo se diseñan y desarrollan los sistemas complejos en otras ingenierías tradicionales, como la ingeniería civil, la aeronáutica, la arquitectura, etc. Este tipo de disciplinas llevan siglos construyendo con éxito rascacielos, cohetes espaciales o increíbles puentes colgantes, cuya complejidad es sin duda tan elevada o más que la de nuestros sistemas de software.




Algo que diferencia a estas disciplinas de la ingeniería del software, al menos en el proceso de diseño y construcción, es que cuentan con lenguajes y notaciones para producir modelos de alto nivel, independientes de las plataformas y tecnologías de desarrollo y de la implementación final, y con procesos bien definidos que son capaces de transformar esos modelos en las construcciones finales de una forma **predecible**, **fiable** y **cuantificable**.



> Si el arquitecto que diseña un rascacielos de más de 100 pisos tuviera que tener en cuenta en el diseño del edificio detalles de muy bajo nivel, como la madera de los marcos de las puertas, el tipo de grifería a usar o el tipo de clavos con el que se van a clavar los cuadros, y además tener que diseñar y construir esos marcos, grifos y clavos, el proyecto sería impracticable.

También es importante considerar el hecho de que los ingenieros y arquitectos tradicionales no usan solo un plano de lo que tienen que construir, sino que normalmente desarrollan varios, cada uno expresado en un lenguaje diferente

y a un nivel de abstracción distinto. El tipo de plano va a depender del tipo de propiedades que quieran especificar, así como del tipo de preguntas que quieren ser capaces de responder con ese plano.

Los tipos de planos pueden usar diferentes lenguajes, notaciones e incluso materiales. Por ejemplo, para un avión se puede usar un plano para la estructura básica, otro con el despiece de sus partes mecánicas a escala, otro para las conexiones eléctricas, una maqueta en plástico para estudiar la aerodinámica en el túnel de viento, un prototipo en metal para estudiar la amortiguación y cargas, un modelo de tamaño reducido para mostrar a los clientes, etc.

Cada modelo (plano, maqueta, prototipo) es el más adecuado para una función, que es lo que se denomina el **propósito del modelo**, y cada uno será útil en un momento diferente del proceso de desarrollo.

Así, el plano con la estructura va a servir para construir la maqueta y para posteriormente guiar el resto de los planos. El modelo a escala va a servir durante la fase de análisis para comprobar si satisface los requisitos de los clientes y del departamento de marketing. La maqueta va a permitir estudiar los aspectos aerodinámicos y refinar el modelo antes de la fase de diseño detallado. Los planos de la estructura y despiece van a servir a los mecánicos para construir el avión, etc.

Algo muy importante a señalar es que los diferentes planos no son completamente independientes, sino que todos representan el mismo sistema aunque desde diferentes puntos de vista y a distintos niveles de abstracción.

Sin embargo, esta forma de proceder de otras ingenierías no es común en el desarrollo de software. Los ingenieros de software no suelen producir modelos de alto nivel de los procesos de negocio de sus sistemas de información, ni siquiera de la arquitectura detallada de sus aplicaciones. Eso hace que la única documentación que queda en la empresa del sistema de información que ha adquirido sea el código final. Incluso en aquellos casos en los que el arquitecto software produjo los modelos de la estructura de la aplicación (usando, por ejemplo, UML), durante el resto de los procesos de desarrollo (codificación, pruebas y mantenimiento) normalmente se alteran muchos detalles que no se reflejan en estos "planos". Esto hace que los modelos originales queden pronto desactualizados y, por tanto, inútiles.

Estos problemas son los que hicieron surgir a principios de los años 2000 un movimiento que promulgaba el uso de los modelos como piezas clave del desarrollo de software, y cuyo primer exponente fue la propuesta denominada MDA (*model-driven architecture*), de la organización OMG (Object Management Group). Esta propuesta, que detallaremos en las siguientes secciones, promulgaba el uso de diferentes tipos de modelos, cada uno a distintos niveles de abstracción.

**Nota**

OMG es un consorcio industrial, formado por más de 700 empresas, cuyo propósito es definir estándares para la interoperabilidad de sistemas software. Inicialmente centrado en sistemas orientados a objetos (p. ej., CORBA), hoy se centra en estándares para modelado de sistemas y procesos (UML, MOF, OCL, BPMN, QVT, etc.).

La idea básica de MDA consiste en elaborar primero modelos de muy alto nivel, denominados modelos independientes de las plataformas o PIM (*platform-independent models*) y completamente independientes de las aplicaciones informáticas que los implementarán o las tecnologías usadas para desarrollarlos o implementarlos. MDA define entonces algunos procesos para ir refinando

esos modelos, particularizándolos progresivamente en modelos específicos de la plataforma a usar o PSM (*platform-specific models*) cada vez más concretos conforme se van determinando los detalles finales.

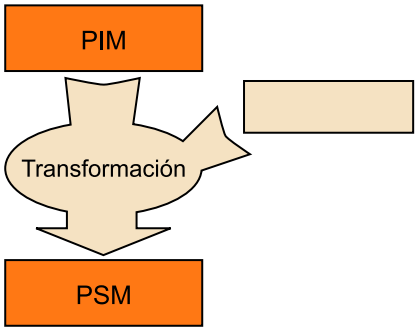
Lo más interesante de MDA es que las plataformas que detallan los servicios a usar (*middlewares*, sistemas operativos, hardware o algunas más abstractas como las que proporcionan requisitos de seguridad) vienen también descritas mediante modelos, y que los procesos que van refinando los modelos de alto nivel hasta llegar a la implementación final vienen descritos por transformaciones de modelos, que no son sino modelos a su vez.

Esta idea se conoce como el "patrón MDA", que gráficamente se muestra en la figura 4, en donde puede verse un modelo PIM que se transforma en un modelo PSM mediante una transformación automatizada. Dicha transformación tiene como entrada otro modelo, que puede ser el de la plataforma concreta, donde se ha de generar el modelo PSM, el de ciertos parámetros de configuración del usuario, etc. Dicho modelo de entrada "parametriza" la transformación, y está representado en la figura 4 por una caja vacía.

**Ved también**

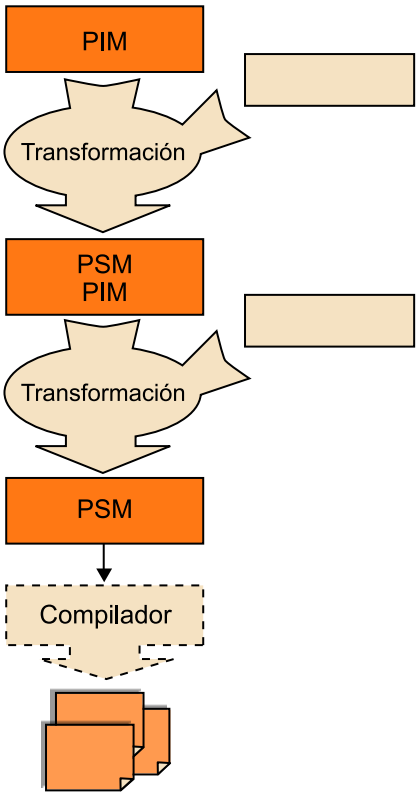
Los elementos de este patrón se verán con más detalle en el apartado "Transformaciones de modelos".

Figura 4. El "patrón" MDA




Utilizando esta idea, en MDA el proceso de desarrollo del software es por tanto un proceso iterativo de transformación de modelos (figura 5).

Figura 5. El proceso de desarrollo en MDA




En la figura 5 podemos observar cómo cada paso transforma un PIM del sistema considerado a un cierto nivel en un PSM del nivel siguiente, hasta que se alcanza la implementación final del sistema, con la particularidad de que cada PSM resultado de una transformación puede convertirse en el PIM para la transformación siguiente (con respecto a otra plataforma o nivel de abstracción).

Es importante señalar que, en este contexto, una implementación no es más que el modelo final de un sistema conteniendo toda la información necesaria para su implantación y puesta en funcionamiento, es decir, otra forma de PSM; quizás el PSM de más bajo nivel.

**Ejemplos**

A partir de un modelo con la especificación básica del diseño de una base de datos y de una interfaz gráfica de acceso, es posible generar todo el código de la aplicación en plataformas diferentes, usando distintas tecnologías de bases de datos y plataformas software. Otro ejemplo del uso de esta cadena de transformaciones MDA se da en lo que se conoce como *model-driven web engineering*, donde a partir del PIM de un sistema web compuesto por un modelo de los datos persistentes de la aplicación (el modelo del *contenido*), de un modelo de navegación, y de un modelo de presentación abstracto, es posible definir transformaciones automáticas a plataformas concretas de componentes (como EJB o .NET) que usen interfaces de usuario basadas en diferentes tecnologías (como PHP o Silverlight). Esto es posible con algunas de las propuestas como UWE o WebML.

Este planteamiento de usar modelos independientes de la tecnología presenta numerosas ventajas. Entre ellas, que las empresas pueden contar con modelos de alto nivel de sus procesos de negocio, de la información que manejan y de los servicios que ofrecen a sus clientes y los que requieren de sus proveedores, de forma independiente de la tecnología que los sustente y de las plataformas

hardware y software que usen en un momento dado. De esta forma, dichos modelos se convierten en los verdaderos activos de las compañías, con una vida muy superior a los sistemas de información que los implementan, ya que las tecnologías evolucionan mucho más rápido que los procesos de negocio de las empresas.

El modelo de negocio de una compañía que ofrece servicio de traducción jurada de documentos puede describirse independientemente de si el sistema de información que lo soporta fue desarrollado para un *mainframe* en los años sesenta, usando una arquitectura cliente-servidor en los ochenta, mediante una aplicación web en el 2000 o en la "nube" en la actualidad.

Por otro lado, también es posible reutilizar los modelos de las plataformas tecnológicas o de implementación, así como de las transformaciones que convierten ciertos modelos de alto nivel a estas plataformas.

Otra ventaja muy importante es que este enfoque MDA permite el tratamiento unificado de todos los artefactos de software que intervienen en el diseño, desarrollo, pruebas, mantenimiento y evolución de cualquier sistema, puesto que todos pasan a ser modelos, incluso las propias transformaciones o el código de la aplicación.

**Lectura recomendada**

**Jean Bezivin** (2005). "The Unification Power of Models". *SoSym* (vol. 2, núm. 4, págs. 171-188).

En MDD el proceso de desarrollo de software se convierte por tanto en un proceso de modelado y transformación, mediante el cual el código final se genera mayormente de forma automática a partir de los modelos de alto nivel, de los modelos de las plataformas destino y de las transformaciones entre ellos.

**Nota**

Del lema de los 90 "Everything is an object" se pasa en el 2003 a "Everything is a model."

Además de conseguir elevar el nivel de abstracción notablemente frente a los lenguajes de programación convencionales y sus procesos de desarrollo habituales, también se obtienen mejoras sustanciales en otros frentes:

**1)** En primer lugar, la gestión de los cambios en la aplicación, así como su mantenimiento y evolución se modularizan de una forma estructurada. Un cambio en la tecnología de implementación de la aplicación solo afectará normalmente a plataformas de nivel más bajo y a las transformaciones que generan los modelos PSM a este nivel, dejando inalterados los modelos superiores (modelos de negocio, requisitos de seguridad, etc.). Estos modelos de alto nivel van a tener una vida más larga y se vuelven menos dependientes en los cambios en los niveles inferiores.

**2)** Por otro lado, el que la propagación de los cambios y la generación de código lo lleven a cabo las transformaciones de modelo hacen que la regeneración de la aplicación pueda realizarse de forma casi automática, ahorrando costes y esfuerzo, además de poder garantizarse una mayor calidad.

Haciendo una analogía con la programación convencional, podemos considerar los modelos de alto nivel como los programas, y la implementación de la aplicación como el código máquina. Las transformaciones de modelos son las que sirven para compilar los programas y generar el código máquina correspondiente. Sin embargo, el planteamiento del proceso de desarrollo de software en términos de modelos a distintos niveles de abstracción y transformaciones entre ellos nos permite dar un salto cualitativo, eliminando muchos de los problemas que planteaban los lenguajes de programación en cuanto a expresividad, insuficiente nivel de abstracción, complejidad accidental, etc. Esto va a permitir al ingeniero software centrarse en la especificación de los modelos en el dominio del problema en vez de en el dominio de la solución, y a un nivel de abstracción adecuado al problema que pretende resolver. Por "dominio del problema" entendemos el espacio o ámbito de aplicación del sistema. Por "dominio de la solución" se entiende el conjunto de lenguajes, metodologías, técnicas y herramientas que se usan para resolver un problema concreto.

**3)** Finalmente, el contar con diferentes modelos del sistema, cada uno centrado en un punto de vista determinado y a un nivel de abstracción bien definido abre un abanico de posibilidades y líneas de actuación muy interesantes y que discutimos en las siguientes secciones.

### 1.5. Tipos de modelos en MDA

Como se ha explicado antes en la sección 1.4, la propuesta de MDA se organiza básicamente en torno a modelos independientes de la plataforma (PIM, *platform independent models*), modelos específicos de la plataforma (PSM, *platform specific models*) y transformaciones entre modelos.

**Ved también**

Véase la figura 4.

Un **modelo independiente de la plataforma** o PIM es un modelo del sistema que concreta sus requisitos funcionales en términos de conceptos del dominio y que es independiente de cualquier plataforma.

Generalmente, la representación de un modelo PIM está basada en un lenguaje específico para el dominio modelado, donde los conceptos representados exhiben un cierto grado de independencia respecto a diversas plataformas; ya sean estas plataformas tecnológicas concretas (como CORBA, .NET o J2EE) o plataformas más abstractas (como por ejemplo, requisitos de seguridad o de fiabilidad). Es así como MDA puede asegurar que el modelo independiente de la plataforma (PIM) sobrevivirá a los cambios que se produzcan en futuras plataformas de tecnologías y arquitecturas software.

**Ejemplo**

El modelo de la cadena de montaje mostrado en la figura 8 es un ejemplo de PIM.

Un **modelo específico para una plataforma** (PSM) es un modelo resultado de refinar un modelo PIM para adaptarlo a los servicios y mecanismos ofrecidos por una plataforma concreta.

Aparte de modelos PIM y PSM, en MDA también son importantes las plataformas, que a su vez se describen como modelos.

En MDA, una **plataforma** es un conjunto de subsistemas y tecnologías que describen la funcionalidad de una aplicación a través de interfaces y patrones específicos, facilitando que cualquier sistema que vaya a ser implementado sobre dicha plataforma pueda hacer uso de estos recursos sin tener en consideración aquellos detalles que son relativos a la funcionalidad ofrecida por la plataforma concreta.

Partiendo de la representación de un PIM y dado un modelo de definición de una plataforma (PDM, *platform definition model*), el PIM puede traducirse a uno o más modelos específicos de la plataforma (PSM) para la implementación correspondiente usando nuevamente lenguajes específicos del dominio, o lenguajes de propósito general como Java, C#, Python, etc.

En un principio, los perfiles UML (*UML profiles*) fueron la alternativa más utilizada como lenguaje de modelado para crear modelos PIM y PSM, pero ahora han ganado en aceptación nuevos lenguajes definidos a partir de lenguajes de metamodelado como MOF o Ecore.

**Ved también**

Los perfiles UML se estudian con detalle en el apartado 3, al hablar de cómo se pueden definir lenguajes específicos de dominio.

Un **perfil UML** se define como una extensión de un subconjunto de UML orientada a un dominio. Se describe a partir de una especialización de dicho subconjunto y utilizando los conceptos que incorpora el mecanismo de extensión de UML: **estereotipos**, **restricciones** y **valores etiquetados**. Como resultado se obtiene una variante de UML para un propósito específico.

En la actualidad existen perfiles adoptados por OMG para plataformas como CORBA, Java, EJB o C++. Más adelante explicaremos con detalle los perfiles de UML, que van a permitir definir tanto la sintaxis como la semántica de un lenguaje, mediante un refinamiento de las de UML. Además, el uso de iconos específicos va a hacer posible dotar a los modelos de notaciones gráficas más usables y atractivas.

**Ejemplo**

Un ejemplo de iconos específicos se muestra en la figura 9.

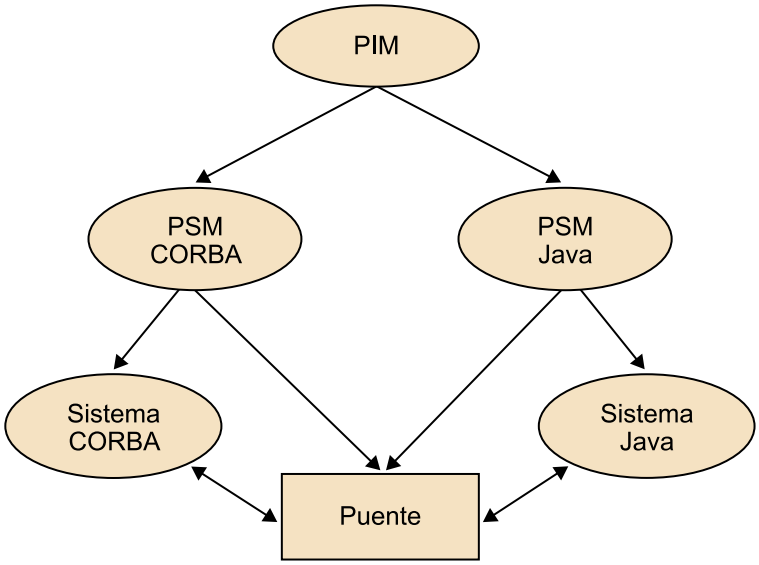
### 1.6. El proceso de desarrollo MDA

Tal y como mencionamos en la sección “Conceptos básicos”, en MDA el proceso de desarrollo de software es un proceso de transformación de modelos iterativo en el que en cada paso se transforma un PIM del sistema considerado a un cierto nivel en un PSM del nivel siguiente, hasta que se alcanza la implementación final del sistema (véase la figura 5). Las transformaciones de modelos son las que se encargan de “dirigir” el proceso. Aunque este es el caso más básico, hay otros procesos MDA de interés, como los que hemos mencionado anteriormente de modernización de sistemas (ADM) y de interoperabilidad basada en modelos (MDI).

**Ved también**

Los procesos ADM y MDI fueron introducidos en la sección “Terminología”.

Figura 6. Puentes de interoperabilidad en MDA




Otro caso interesante en donde los modelos pueden proporcionar importantes beneficios es en la integración de sistemas. Generalmente la implementación de un sistema requiere del uso y cooperación entre varias tecnologías software para lograr la funcionalidad especificada. Tal es el caso de las aplicaciones que se implementan reutilizando componentes o sistemas externos. Así, la integración entre diferentes tecnologías puede ser especificada a nivel de modelos en el contexto de MDA mediante la definición explícita de puentes (*bridges*), como se observa en la figura 6.

Un **puente** (*bridge*) se define como un elemento de modelado que permite comunicar e integrar sistemas heterogéneos entre sí.

En MDA, la implementación de puentes se simplifica enormemente, siendo las herramientas de transformación las que se encargan de generar los puentes de forma automática uniendo las diferentes implementaciones y los distintos modelos de las plataformas.

## 1.7. Actuales retos de MDD y MDA

Los primeros resultados de la adopción de las prácticas de MDD son prometedores, y están demostrando ser realmente efectivos en aquellos casos en los que se ha aplicado adecuadamente. Por supuesto, como cualquier otra tecnología o aproximación, MDA no está exenta de riesgos. Aparte de introducir numerosas ventajas y oportunidades, es importante ser consciente de los retos que deberemos afrontar en nuestro negocio al plantear un nuevo desarrollo adoptando esta propuesta.

- Costes en la adopción de las técnicas y herramientas MDD, por la curva de aprendizaje e inversión que representan para cualquier organización.

- Problemas de adopción por parte de los actuales equipos de desarrollo, al tratarse de prácticas bastante diferentes a las tradicionales.

- Problemas a nivel organizativo, ya que muchos gestores no están dispuestos a apoyar inversiones cuyo beneficio a muy corto plazo no es trivial (aunque a largo plazo puedan mejorar significativamente la productividad de la empresa).

- Falta de madurez en algunas de las herramientas MDD (editores de modelos, generadores de código, máquinas de transformación), principalmente debido a la reciente implantación de esta disciplina.

- Falta de buenas prácticas y de procesos de desarrollo MDD genéricos, lo que no permite adoptar esta práctica de forma unificada y *off-the-shelf*.

**Web recomendada**

La adopción de MDA cuenta ya con numerosos casos de éxito en empresas y organizaciones, incluyendo DaimlerChrysler, Lockheed Martin, Deutsche Bank, ABB, National Cancer Institute, o Credit Suisse. La descripción de muchos de estos casos puede consultarse en http://www.omg.org/mda/products_success.htm.

A modo de resumen, el siguiente cuadro muestra algunas circunstancias en las que es conveniente aplicar las técnicas MDD en una empresa, frente a aquellas que quizá lo desaconsejen.

| Es conveniente aplicar MDD cuando... | No es conveniente aplicar MDD cuando... |
|---|---|
| El producto se desplegará en múltiples plataformas o interesa desligarlo de una tecnología concreta porque tendrá una vida útil prolongada. | El producto se desplegará en una sola plataforma que está prefijada de antemano y su vida útil será muy corta. |
| El equipo puede invertir esfuerzos en adaptar o desarrollar herramientas propias para obtener beneficios a medio/largo plazo. | El proyecto necesita disponer de una tecnología madura para obtener resultados inmediatos. |
| Se dispone de un equipo con experiencia en MDD o bien recursos y tiempo para formarse. | No se dispone de experiencia en MDD ni oportunidades para formarse. |
| Los gerentes y directivos de la empresa apoyan el cambio de paradigma de desarrollo. | Existe mucho recelo o incluso rechazo por parte de la dirección de la organización. |

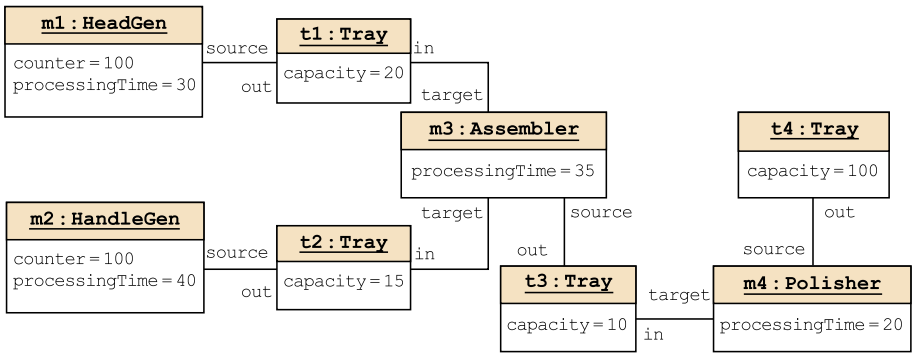
Algo que en general está dando muy buen resultado en muchas empresas es la adopción de procesos MDD en algunos equipos de desarrollo y para proyectos muy concretos, a modo de experimento controlado en donde poder probar los resultados y valorar estas prácticas.

## 1.8. Un caso de estudio

Para ilustrar los conceptos vistos en este apartado, y como hilo argumental para el resto del módulo, vamos a modelar un sistema, concretamente una cadena de montaje de una fábrica de martillos (en adelante nos referiremos a él como "CadenaDeMontaje"). El sistema tiene cuatro tipos de máquinas: generadores de mangos (`HandleGen`), generadores de cabezales (`HeadGen`), ensam-

bladores (`Assembler`) y pulidoras (`Polisher`). Las dos primeras generan las piezas básicas, que han de ser posteriormente ensambladas para formar martillos, y que finalmente serán embellecidos en la pulidora. Las máquinas se conectan mediante bandejas (`Tray`), desde las que toman sus piezas de entrada y en donde colocan las piezas producidas. Cada máquina tarda un tiempo en procesar una pieza, y cada bandeja tiene una capacidad limitada. Una posible disposición de la cadena de montaje se muestra en el diagrama de clases de la figura 7.

Figura 7. Un modelo de una cadena de montaje de martillos

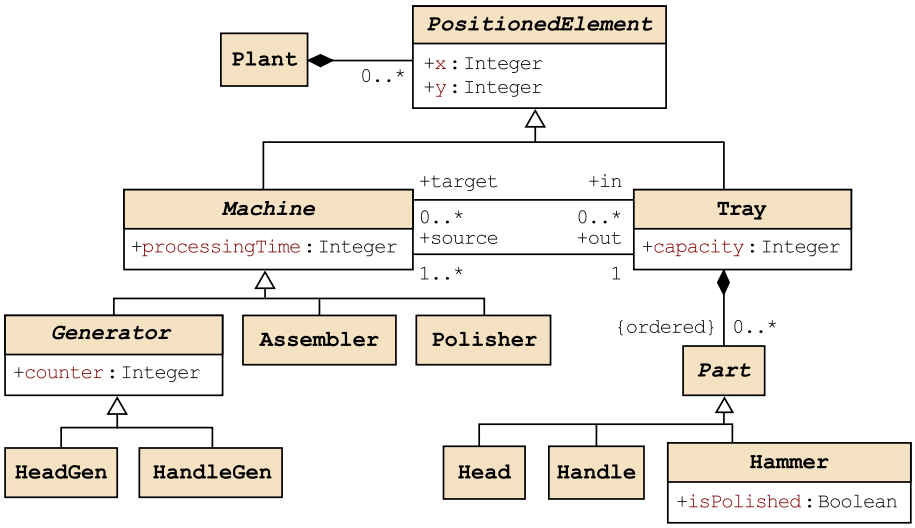



La de la figura 7 es una posible configuración del sistema, aunque podrían establecerse otras dependiendo del número de máquinas generadores y ensambladoras que se utilicen para acelerar el proceso, la calidad de las máquinas (hay máquinas en el mercado con menores tiempos de procesado, pero cuestan más), la capacidad de las bandejas (de nuevo, a más capacidad mayor precio), etc.

> Lo que se busca con este sistema es maximizar la producción tratando de optimizar los costes y los tiempos.

Como podemos observar, el modelo mostrado en la figura 7 es un modelo PIM, independiente de cualquier otra plataforma concreta. Nos podemos preguntar cuál es su metamodelo, que no es sino otro modelo pero que contiene los conceptos del dominio del problema y las relaciones entre ellos. Un posible metamodelo de ese modelo es el que se muestra en la figura 8.

Figura 8. Metamodelo de la cadena de montaje de martillos




En este caso el modelo y su metamodelo están expresados en UML.

El metamodelo describe las clases de los objetos que forman el modelo y las relaciones entre ellos, y el modelo describe los elementos que forman parte del sistema que queremos representar.

**UML y las restricciones OCL**

En general, UML no va a ser suficiente para definir correctamente metamodelos, sino que va a ser también preciso incluir algunas restricciones de integridad. Por ejemplo, el número de partes que contiene una bandeja ha de ser siempre menor que su capacidad. En UML, esas restricciones de integridad se expresan haciendo uso de OCL, un lenguaje que estudiaremos en el apartado 2 de este módulo.

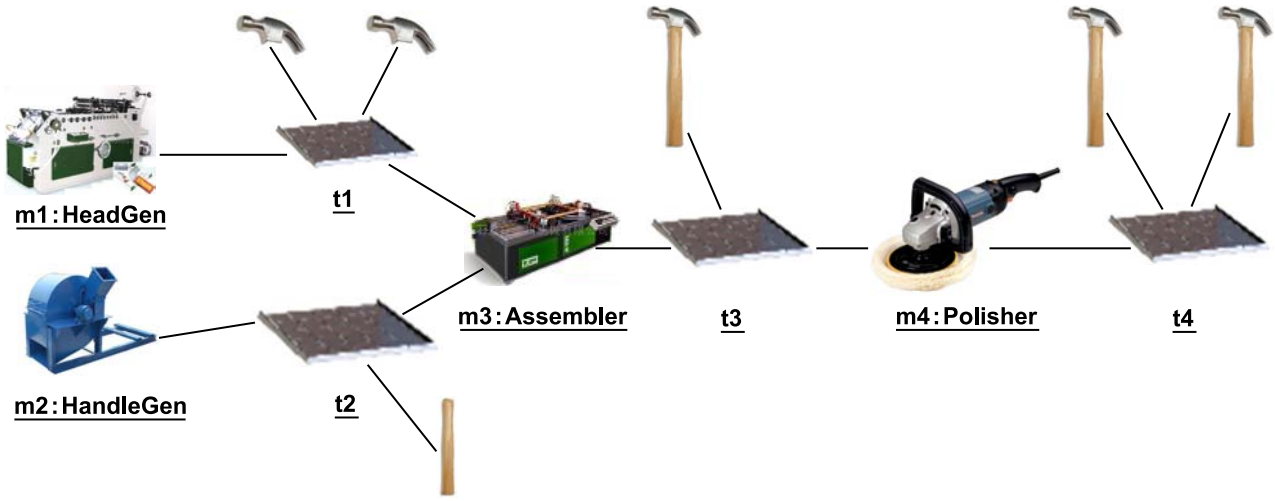
El propósito del modelo es ofrecer una descripción de alto nivel de los elementos básicos del sistema y su estructura, con el objetivo de su comprensión por parte de los usuarios finales. Sin embargo, este modelo ignora otros muchos aspectos del sistema por no ser relevantes desde ese punto de vista.

El modelo no describe de forma explícita el comportamiento del sistema, ni la tecnología con la que está implementado, ni la estructura de la base de datos que almacena la información que queremos que sea persistente, etc.

Otros modelos serán los encargados de especificar estos y otros aspectos, como veremos más adelante.

Usando un perfil UML adecuado, es posible expresar el modelo mostrado en la figura 7 como aparece en la figura 9 (hemos incluido algunas piezas en las bandejas para mostrar cómo se verían con esta notación, representando no el modelo inicial del sistema, sino un modelo en un instante dado en el tiempo).

Figura 9. El modelo de la cadena de montaje expresado con iconos




## 1.9. Conclusiones

En este apartado hemos presentado una descripción del estado actual del desarrollo de software dirigido por modelos, especialmente en el contexto de MDA. Para ello, hemos analizado los conceptos y mecanismos que aporta esta propuesta y cómo puede beneficiar al proceso de desarrollo de grandes sistemas software por parte de las empresas del sector.

Una vez hemos visto en este apartado los conceptos generales de MDA, en los siguientes vamos a estudiar en detalle algunos de los lenguajes y tecnologías asociadas: OCL para especificar con precisión los modelos UML, las técnicas de metamodelado para definir lenguajes específicos de dominio, y los lenguajes y herramientas de transformación de modelos.

## 2. El lenguaje de restricciones y consultas OCL

**Ved también**

Podéis encontrar una introducción a OCL en la asignatura *Ingeniería de requisitos* del grado de Ingeniería Informática.

En el apartado anterior hemos comentado la importancia que tiene para MDE el que los modelos sean **precisos** y **completos**. La precisión implica que su interpretación no admita ambigüedad alguna, lo que garantiza que todos los que van a utilizar los modelos (diseñadores, desarrolladores, analistas, usuarios finales y, sobre todo, ¡otros programas!) van a interpretarlos de la misma forma. El ser completos implica que han de contener todos los detalles relevantes para representar el sistema desde el punto de vista adecuado.

Aunque existen notaciones tanto gráficas como textuales para ello, cuando se piensa en representación de modelos, sobre todo de sistemas software, suele pensarse en notaciones gráficas como UML. Dichas notaciones son muy apropiadas para representar modelos de una forma visual y atractiva, pero no poseen la expresividad suficiente para describir toda la información que debe contener un modelo.

El modelo de la figura 8 del apartado anterior, que describe el metamodelo de la cadena de montaje, no describe algunas propiedades muy importantes del sistema, como que el número de partes que contiene una bandeja ha de ser siempre menor que su capacidad, o que los contadores de piezas han de ser positivos. Este tipo de condiciones no pueden expresarse solo con la notación que proporciona UML.

Tradicionalmente, en el caso de UML, lo que suele hacerse es completar los modelos con descripciones en lenguaje natural sobre este tipo de condiciones y cualquier otra información adicional sobre el modelo que sea relevante.

**Ejemplos**

La semántica del modelo y su entorno, las restricciones de integridad, la especificación de los valores iniciales de los atributos, el comportamiento de las operaciones, etc.

El problema es que las descripciones en lenguaje natural, aunque fáciles de escribir y leer por parte de cualquier persona, son normalmente ambiguas, y desde luego no son manipulables por otros programas, que es lo que precisamente requiere MDE. Por otro lado, también sabemos que existen los lenguajes denominados "formales", que no tienen ambigüedades pero que son difíciles de usar por personas que no tengan un cierto bagaje matemático, tanto a la hora de escribir especificaciones de sistemas como de comprender las escritas por otros. Otra opción, por supuesto, es usar el código fuente, que no es sino un modelo final del software que contiene todos los detalles. Sin embargo, su bajo nivel de detalle y su elevada complejidad no lo convierten en la mejor opción para comprender el sistema y razonar sobre él.

**Consulta recomendada**

Axel van Lamsweerde (2000) propone una definición de lenguaje formal en: "Formal specification: a roadmap *". Proceedings of the Conference on The Future of Software Engineering* (FOSE'00) (págs. 147-159). ACM, NY. http://doi.acm.org/10.1145/336512.336546.

OCL (*object constraint language*) es un lenguaje textual, con base formal, y que además posee mecanismos y conceptos muy cercanos a los de UML, lo cual facilita su uso por parte de los modeladores. Por ejemplo, la siguiente expresión OCL indica que el número de partes que hay en una bandeja en cualquier momento ha de ser inferior a su capacidad:

```
context Tray inv NoOverflow: self.part->size() <= self.capacity
```

OCL es un lenguaje formal, basado en teoría de conjuntos y lógica de primer orden, que fue inicialmente desarrollado como parte del método de diseño de sistemas denominado Syntropy, de Steve Cook y John Daniels, para ser posteriormente adoptado por la División de Seguros de IBM como lenguaje preciso de modelado de sus procesos de negocio.

**Consulta recomendada**

Syntropy es un método de diseño de sistemas orientado a objetos, sobre el que se basó UML. Está descrito en el libro S. Cook, J. Daniels (1994). *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice Hall. http://www.syntropy.co.uk/syntropy/.

Cuando UML necesitó un lenguaje para expresar restricciones de integridad sobre sus modelos, la comunidad MDA vio a OCL como el candidato idóneo. Sus mecanismos para hacer navegar modelos encajaban de forma natural con la estructura de UML, y ambos lenguajes se complementaban muy bien. Una de las principales aportaciones de la OMG fue la completa integración de OCL con UML y con MOF a la hora de incorporarlo a su catálogo de estándares de modelado y de MDE, haciendo compatibles sus sistemas de tipos y las formas de asignar nombres a sus elementos. Además, cada uno de estos lenguajes aporta sus ventajas al ser combinados, llegando al punto de que en el contexto de MDA no pueden vivir el uno sin el otro: para obtener un modelo completo, son necesarios tanto los diagramas UML como las expresiones OCL.

**Web recomendada**

OCL2.0 fue adoptado en el 2003 por el grupo OMG como parte de UML 2.0. El estándar de OCL publicado por la OMG está disponible en www.omg.org/spec/OCL/. La versión descrita aquí corresponde a OCL 2.3.1.

## 2.1. Características de OCL

En primer lugar, OCL es un lenguaje **fuertemente tipado**. En otras palabras, toda expresión OCL es de un determinado tipo. Para ello, los sistemas de tipos de UML y OCL son completamente compatibles: el conjunto de tipos básicos de OCL se corresponde con los que define UML (`Integer`, `String`, `Set`, `Sequence`, etc.), cada clase UML define un tipo OCL, y la herencia en UML se interpreta como subtipado en OCL.

Otra característica interesante de OCL es que es un lenguaje de especificación que **no tiene efectos laterales**. Esto quiere decir que la evaluación de cualquier expresión OCL no puede modificar el estado del modelo, sino solo consultarlo y devolver un valor (de un cierto tipo, precisamente el tipo de la expresión OCL). Por tanto, no puede considerarse a OCL como un lenguaje de programación: sus sentencias no son ejecutables y solo permiten consultar el estado del sistema, evaluar una expresión o afirmar algo sobre él[2].

(2)Por ejemplo, un invariante que ha de ser siempre cierto, o el estado del sistema tras una operación.

Es importante señalar también que la evaluación de una expresión OCL se supone **instantánea**. Eso quiere decir que los estados de los objetos de un sistema no pueden cambiar mientras se evalúa una expresión.

Por último, OCL cuenta con diversas herramientas que permiten comprobar sus especificaciones sobre diferentes tipos de modelos. Entre ellas destacamos las siguientes:

- **Dresden OCL Toolkit** proporciona un conjunto de herramientas para el análisis sintáctico y evaluación de restricciones OCL en modelos UML, EMF y Java, a la vez que proporciona herramientas para la generación de código Java/AspectJ y SQL. Es posible utilizar las herramientas de Dresden OCL como una biblioteca en otros proyectos o como un *plugin* Eclipse para manipular expresiones OCL.

- **USE**, herramienta desarrollada por la Universidad de Bremen, está diseñada para especificar tanto modelos UML como sus restricciones de integridad usando OCL.

- **MDT/OCL** para modelos Ecore de Eclipse.

**Nota**

El OCL Portal ofrece referencias a las distintas herramientas para OCL disponibles en la actualidad, así como referencias a trabajos y otras páginas web donde las herramientas son comparadas y analizadas (http://st.inf.tu-dresden.de/ocl/).

**Web recomendada**

Más información sobre USE: http://www.sourceforge.net/apps/mediawiki/useocl/.

**Web recomendada**

Página web de MDT/OCL: www.eclipse.org/projects/project.php?id=modeling.mdt.ocl.

Sin embargo, no demasiadas herramientas de modelado (editores de modelos, entornos de desarrollo de código, etc.) dan soporte actualmente a la especificación y validación de expresiones OCL. Y de entre aquellas que lo permiten, muy pocas son capaces de utilizarlo en toda su potencia (por ejemplo, para la generación de código). De todas formas, la necesidad de utilizar los modelos de forma precisa está haciendo madurar las herramientas OCL rápidamente.

### 2.2. Usos de OCL

El lenguaje OCL admite numerosos usos en el contexto de MDE:

**a)** Como lenguaje de consulta sobre modelos.

**b)** Para especificar invariantes y restricciones de integridad sobre las clases y los tipos de un modelo de clases UML.

**c)** Para especificar las pre- y poscondiciones de las operaciones.

**d)** Para describir guardas en las máquinas de estados.

**e)** Para especificar conjuntos de destinatarios de los mensajes y acciones UML.

**f)** Para especificar restricciones en operaciones.

**g)** Para especificar reglas de derivación de atributos y cuerpos de las operaciones.

**Web recomendada**

Toda la información sobre OCL 2.3 puede consultarse en la especificación de la OMG: www.omg.org/spec/OCL/2.3.1

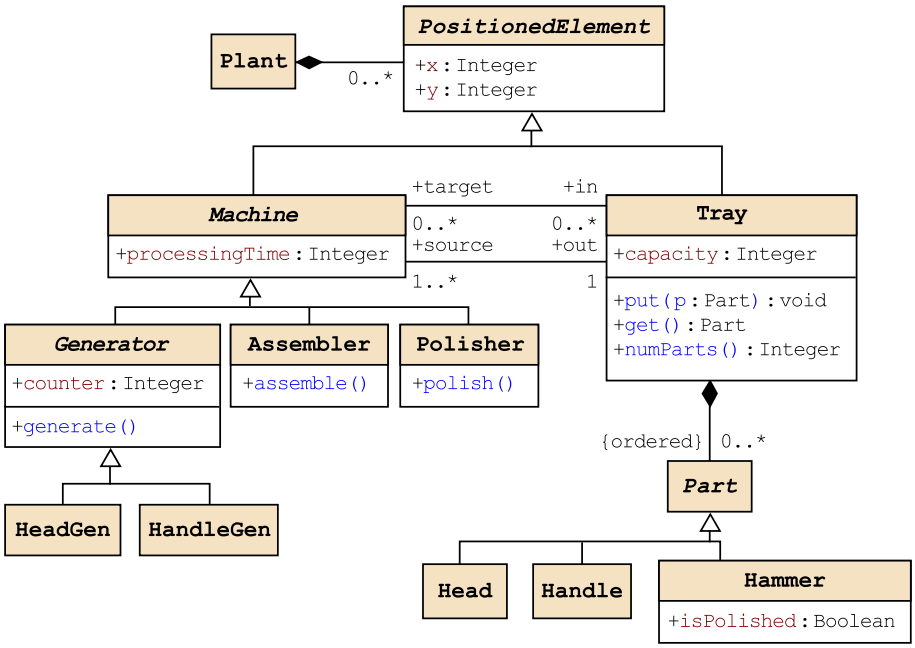
En las asignaturas del Área de Ingeniería del Software del grado de Ingeniería Informática se introdujo el lenguaje OCL junto con sus tipos y operaciones básicas. En este apartado vamos a presentar el lenguaje desde el punto de vista de su uso en el contexto de MDE, e introduciremos algunas funciones algo más avanzadas, como la especificación de la semántica de operaciones, las restricciones de integridad sobre los modelos y la creación de nuevos atributos y operaciones.

**Ved también**

Ver el módulo "Documentación de requisitos" de la asignatura *Ingeniería de requisitos* del Grado de Ingeniería Informática.

Para ilustrar el uso de OCL utilizaremos el metamodelo de la cadena de montaje que ya presentamos en el apartado 1 (figura 4), y que mostramos en la figura 10, enriquecido con algunas operaciones en las clases para mostrar también cómo se especifica su comportamiento con OCL.

Figura 10. El metamodelo de una cadena de montaje




## 2.3. Restricciones de integridad en OCL

**Ejemplo**

Ejemplos de este tipo de restricciones son las multiplicidades de los extremos de una asociación UML.

Las restricciones de integridad (o invariantes) sobre un metamodelo expresan las condiciones que deben cumplir los modelos que representen sistemas válidos. En la figura 10 podemos ver, por la multiplicidad de la asociación, que una máquina de cualquier modelo que represente una cadena de montaje debe tener exactamente una bandeja de salida (`out`). Igualmente, toda bandeja tiene que tener al menos una máquina que deposite partes en ella (`source`). Esta restricción, que se expresa en términos de la multiplicidad de los extremos de una asociación, también puede ser expresada en OCL usando **invariantes**:

```
context Tray inv: self.source->size() > 0
context Machine inv: self.out->size() = 1
```

**Nota**

Obviamente, teniendo la posibilidad de expresar multiplicidades directamente en los diagramas de clases, no tiene mucho sentido utilizar restricciones OCL como estas, aunque, como siempre, es decisión del modelador, y a nosotros nos sirve para ilustrar su naturaleza.

Recordemos que el **contexto** de una expresión OCL indica un tipo del modelo a cuyas instancias hace referencia la expresión. Dentro de un contexto, la palabra reservada `self` se usa para referirse a una instancia de dicho tipo. La operación `size()` sobre colecciones (`Set`, `Sequence`, `OrderedSet`, `Bag`) permite conocer el tamaño de las mismas. El punto `(".")` y la flecha `("->")` permiten navegar por los atributos y asociaciones de los elementos, dependiendo de si son elementos individuales o colecciones, respectivamente.

Como hemos dicho anteriormente, hay sin embargo otras restricciones que UML no es capaz de expresar directamente con su notación. Por ejemplo, la siguiente expresión OCL indica que el contador de cualquier generador debe ser positivo:

```
context Generator inv PositiveCounter: self.counter >= 0
```

Obsérvese cómo es posible asignar nombres a los invariantes (en este caso `PositiveCounter`) y al resto de condiciones OCL, para poder referirnos a ellos posteriormente.

Otra restricción interesante en nuestro ejemplo es la siguiente, llamada `NoOverflow`, que impone que las bandejas no pueden contener más piezas de las indicadas por su atributo `capacity`:

```
context Tray inv NoOverflow: self.part->size() <= self.capacity
```

Al igual que en UML, en OCL es posible navegar por asociaciones con extremos sin nombre utilizando el nombre de la clase al otro extremo de la asociación, en minúscula (en este caso, `self.part`).

**Ved también**

Podéis encontrar más información sobre navegación en colecciones en el apartado "Especificación de la semántica de operaciones".

Los invariantes también nos sirven para definir apropiadamente las características de los tipos del modelo.

Por ejemplo, en la figura 10 se indica que hay tres tipos de máquinas, pero no queda reflejado que cada una de ellas puede imponer una restricción diferente sobre el número de bandejas a las que puede estar conectada: pulidoras y ensambladoras deberían tener al menos una bandeja de entrada (como veremos a continuación, las ensambladoras tendrían dos si no permitimos bandejas con elementos de distinto tipo), mientras que las generadoras no deberían tener ninguna. De igual forma, podríamos permitir que cada máquina tuviera varias

bandejas de salida. Sin embargo, dado que el número de bandejas de salida está limitado a uno, vamos a exigir que las generadoras no tengan bandejas de entrada, que las ensambladoras tengan exactamente dos y que las pulidoras tengan exactamente una.

Esto se puede especificar mediante los tres invariantes siguientes:

```
context Generator inv NoInputTrays:  self.in->isEmpty()
context Assembler inv TwoInTrays:  self.in->size() = 2
context Polisher inv OneInTray:  self.in->size() = 1
```

OCL admite numerosas formas de expresar invariantes. Por ejemplo, los tres anteriores pueden especificarse en uno solo, sobre la clase `Machine`, como sigue:

```
context Machine inv MachinesAndTrays:
   self.oclIsTypeOf(Generator) implies self.in->isEmpty() and
   self.oclIsTypeOf(Assembler) implies self.in->size() = 2 and
   self.oclIsTypeOf(Polisher) implies self.in->size() = 1
```

También es posible hacerlo usando una estructura condicional:

```
context Machine inv MachinesAndTrays2:
   self.in->size() = if self.oclIsTypeOf(Assembler) then 2
                     else if self.oclIsTypeOf(Polisher) then 1
                          else 0
                          endif
                     endif
```

OCL distingue entre las operaciones `oclIsTypeOf(T)` y `oclIsKindOf(T)`. La diferencia es sutil pero importante: la primera determina si la clase a la que pertenece un objeto (es decir, su clasificador) es `T`, mientras que la segunda determina si la clase a la que pertenece un objeto es `T`, o cualquier superclase de `T`.

Por ejemplo, si `g` es un objeto de clase `HeadGen`, entonces son ciertas las siguientes expresiones: `g.oclIsTypeOf(HeadGen)`, `g.oclIsKindOf(HeadGen)`, `g.oclIsKindOf(Generator)`, `g.oclIsKindOf(Machine)` y `g.oclIsKindOf(PositionedElement)`. Sin embargo, no lo son `g.oclIsTypeOf(Generator)` ni `g.oclIsTypeOf(Machine)`.

Si establecemos el número de bandejas de entrada de cada una de las máquinas tenemos que asegurarnos de que contengan partes de los tipos apropiados. Podemos tener varias máquinas colocando piezas en una misma bandeja, pero todas las partes de una misma bandeja deberían ser del mismo tipo, lo que podría expresarse como sigue:

```
context Assembler inv AllInGenerators:
   self.in->forAll(t | t.source->forAll(m | m.oclIsTypeOf(Generator)))
```

Sin embargo, una restricción de este tipo no sería suficiente pues no podemos permitir que, por ejemplo, una pulidora tenga una bandeja de mangos como entrada. Necesitamos que todas las máquinas poniendo piezas en la bandeja de entrada de una ensambladora sean o generadoras de mangos o cabezas de martillo, y además tener de ambos tipos:

```
context Assembler inv HeadAndHandleGens:
   self.in->forAll(t | t.source->forAll(m | m.oclIsTypeOf(Generator))
                and t.source->exists(m | m.oclIsTypeOf(HandleGen))
                and t.source->exists(m | m.oclIsTypeOf(HeadGen)))
```

Dado que hemos fijado en dos el número de bandejas de entrada a una máquina ensambladora y que todas las partes de una misma bandeja han de ser del mismo tipo, el invariante `HeadAndHandleGens` podría simplificarse:

**Ejercicio propuesto**

Hemos restringido el tipo de máquinas de entrada a otras máquinas, o para ser más precisos, las máquinas que pueden colocar partes en las bandejas de entrada de cada máquina. ¿Es necesario también establecer restricciones sobre las de salida?

```
context Assembler inv HeadAndHandleGens2:
   self.in->forAll(t | t.source->exists(m | m.oclIsTypeOf(HandleGen))
                and t.source->exists(m | m.oclIsTypeOf(HeadGen)))
```

De forma similar, una pulidora solo puede tener ensambladoras como máquinas que ponen partes en su bandeja de entrada:

```
context Polisher inv :
   self.in->forAll(t | t.source->forAll(m | m.oclIsTypeOf(Assembler)))
```

Con las restricciones establecidas hasta ahora podemos tener situaciones donde nos encontremos con varias bandejas finales de las que ninguna máquina coge piezas. Podemos imponer que haya una única bandeja final, en donde se depositarán todas las piezas producidas, con el siguiente invariante:

```
context Tray inv OneFinalTray:
   Tray.allInstances()->one(target->isEmpty())
```

En ese invariante hemos hecho uso de la operación `allInstances()` que, aplicada a un nombre de un tipo, devuelve el conjunto de instancias que actualmente existen en el modelo de ese tipo. Por su parte, el cuantificador `one()`, aplicada a una colección, devuelve *true* si existe exactamente un único elemento que cumple una propiedad. OCL también ofrece la función `any()`, que devuelve alguno de los elementos de un conjunto que satisface una propiedad. Si hay más de uno que la satisface, puede devolver cualquiera de ellos. Si no hubiera ninguno, devolvería el valor `null`.

### Ejercicio propuesto

Supongamos que eliminamos el invariante `OneFinalTray`. Poned un ejemplo de sistema que cumpla con el resto de los invariantes y que tenga más de una bandeja final. Discutid la necesidad de añadir el invariante `OneFinalTray`.

En OCL hay que distinguir entre los valores `null` e `invalid`. El primero es un valor que indica "ausencia de valor". Por ejemplo, un extremo de una asociación con multiplicidad `0..1` que no está asociado a ninguna clase toma el valor `null`. Por otro lado, `invalid` es un valor que indica que la operación que se trata de ejecutar es incorrecta, como por ejemplo cuando se hace una división por cero. Es importante destacar que `null` es un valor válido que puede ser almacenado en colecciones, algo que no ocurre con `invalid`. Lo que no pueden es ejecutarse operaciones sobre `null`, salvo las correspondientes a `Bag` (por ejemplo, `null->isEmpty()` devuelve `true`, y `null->notEmpty()` devuelve false) sin embargo, `isOclType(T)` devuelve `invalid` si `T` es `null`. OCL proporciona las operaciones `oclIsInvalid()` para saber si un elemento es `invalid`, y `oclIsUndefined()` para saber si es `null` o `invalid`.

La operación `allInstances()` es muy útil cuando queremos imponer una restricción sobre el número de objetos que puede haber en un modelo. Por ejemplo, podemos querer indicar que el número de máquinas de un sistema debe coincidir con el número de bandejas:

```
context Plant inv SameNumbers:
   Machine.allInstances()->size() = Tray.allInstances()->size()
```

Obsérvese que este invariante es quizás más restrictivo de lo necesario. Aunque es por supuesto decisión del modelador añadirlo o no, en principio no hay problema en que varias máquinas utilicen una misma bandeja como bandeja de salida.

### Ejercicio propuesto

El invariante `SameNumbers` limita drásticamente las configuraciones de máquinas y bandejas permitidas, poned dos ejemplos de configuraciones con distinto número de máquinas.

La siguiente restricción `NoOverflow2` es equivalente a la restricción `NoOverflow` que vimos anteriormente:

```
context Tray inv NoOverflow2:
   Tray.allInstances()->forAll(t | t.part->size() <= t.capacity)
```

Es importante también establecer que no haya dos máquinas o bandejas en la planta con la misma posición:

```
context PositionedElement inv NoOverlap:
   PositionedElement.allInstances()->forAll(m1,m2 |
         m1<>m2 implies (m1.x<>m2.x  or  m1.y<>m2.y) )
```

Una situación muy común en los diagramas UML es que pueden permitir algunos ciclos que no deben darse en los sistemas que modelan. Por ejemplo, sin considerar las restricciones establecidas hasta ahora, el diagrama UML de la figura 10 admite ciclos que permiten a una máquina estar conectada a una misma bandeja como bandeja de entrada y de salida. Para evitar esta situación podemos imponer el siguiente invariante:

```
context Machine inv NoCycles:
   self.out->forAll(t | t.target->excludes(self)) and
   self.in->forAll(t | t.source->excludes(self))
```

### Ejercicio propuesto

¿Es realmente el invariante `NoCycles` deducible del resto de los invariantes? Justificad vuestra respuesta.

Como es habitual, este invariante admite diversas formulaciones equivalentes, como por ejemplo:

```
context Machine inv NoCycles2:
   self.out->intersection(self.in)->isEmpty()
```

Normalmente, el mejor contexto es aquel con el que el invariante se puede expresar de la manera más simple, o bien con el que el invariante es más simple de comprobar; en cuanto a la completitud de nuestras especificaciones, será en última instancia responsabilidad del modelador, y para ello la experiencia de este será fundamental.

**Lectura recomendada**

**Jordi Cabot, Ernest Teniente**. (2007). "Transformation techniques for OCL constraints". *Science of Computer Programming* (vol. 3, núm. 68, págs. 179-195).

### 2.4. Creación de variables y operaciones adicionales

Algo que también permite OCL es definir nuevos atributos y operaciones en el modelo, usando la palabra reservada `def`. El objetivo de estas operaciones y atributos será normalmente el simplificar las expresiones OCL necesarias. Las dos siguientes expresiones declaran, respectivamente, un nuevo atributo (`isFinal`) para la clase `Tray`, y una operación para la clase `PositionedElement`. El atributo determina si una bandeja es final o no, y la operación calcula la distancia de Manhattan desde otro objeto de esa clase hasta él.

```
context Tray
   def: isFinal: Boolean = self.target->isEmpty()
context PositionedElement
   def: distanceTo(p : PositionedElement) : Integer =
            (p.x - self.x).abs() + (p.y - self.y).abs()
```

OCL requiere que los nuevos atributos sean siempre derivados, y que las operaciones definidas así sean solo de consulta.

### 2.5. Asignación de valores iniciales

Algo fundamental para nuestros diagramas UML es tener la posibilidad de añadir reglas que asignen valores iniciales a los atributos y a los extremos de las asociaciones.

Las reglas que expresan valores iniciales son muy simples, pues basta con detallar el contexto, el atributo y el valor inicial que deseamos que tome en el momento en el que se cree un objeto de ese tipo. En nuestro ejemplo, podemos suponer que inicialmente las máquinas generadoras parten de 100 piezas cada una, y que los martillos cuando se crean no están pulidos:

```
context Generator::counter : Integer
   init = 100
context Hammer::isPolished : Boolean
   init = false
```

OCL también permite expresar los valores de los atributos derivados, con la cláusula `derive`. El diagrama de la figura 10 no contiene atributos derivados, pero supongamos que la clase `Tray` tuviera uno, denominado `connectedMachines`, que contuviese el número de máquinas de entrada y de salida que en cada momento tiene conectada una bandeja. El valor de dicho atributo derivado se especificaría en OCL de la siguiente manera:

```
context Tray::connectedMachines : Integer
   derive: self.source->size() + self.target->size()
```

## 2.6. Colecciones

Cuando se trabaja con colecciones de objetos en OCL, hay que tener en cuenta la diferencia entre las cuatro colecciones disponibles. En un `Set`, cada elemento solo puede aparecer una vez. En un `Bag`, los elementos pueden aparecer más de una vez. Un `Sequence` es un `Bag` donde los elementos están ordenados. Un `OrderedSet` es un `Set` donde los elementos están ordenados.

**Ved también**

Las colecciones se introducen en la asignatura *Ingeniería de requisitos* del grado de Ingeniería Informática.

Cada una de estas colecciones admite una serie de operaciones básicas para manejarlas (`size()`, `select()`, `collect()`, `sum()`, etc.). Sin embargo, hay algunos aspectos menos conocidos del comportamiento de las colecciones, sobre todo en lo que se refiere a cómo se construyen cuando se navega a través de las asociaciones de un diagrama de clases de UML o de MOF, y que merece la pena presentar aquí.

**Ved también**

Ver la descripción de las operaciones básicas en el módulo "Documentación de requisitos" de la asignatura *Ingeniería de requisitos*.

### 2.6.1. Navegaciones que resultan en `Sets` y `Bags`

En OCL existe la regla de que cuando se navega a través de más de una asociación con multiplicidad mayor que `1`, se obtiene un `Bag`.

> Por ejemplo, cuando se va desde una instancia de clase `A` a través de más de una instancia de clase `B` hasta más de una instancia de clase `C`, el resultado es un `Bag` de objetos de tipo `C`. Sin embargo, cuando se navega solo en una de estas asociaciones, el resultado de la navegación es un `Set`.

Para poder entender por qué es importante este hecho, supongamos que queremos conocer cuántas máquinas están conectadas a una máquina dada a través de sus bandejas, y para ello creamos la siguiente expresión OCL:

```
context Machine def: neighbours : Integer =
   self.in.source->union(self.out.target)->size()
```

Esta expresión plantea, sin embargo, un problema. Una máquina puede estar conectada a más de una bandeja de entrada, y por tanto se podría repetir una referencia al mismo objeto de la clase `Machine` en esas colecciones ya que son `Bags` y no `Sets`. Así, en la expresión escrita anteriormente, una máquina puede ser contada dos veces, lo cual no sería correcto. Para eso, OCL ofrece operaciones que transforman uno de los tipos de colecciones en cualquiera de los otros, como por ejemplo `asSet()` o `asBag()`. En este caso, usando una de estas operaciones corregimos la definición anterior:

```
context Machine def: neighbours : Integer =
   self.in.source->union(self.out.target)->asSet()->size()
```

### 2.6.2. Navegaciones que resultan en `OrderedSets` y `Sequences`

Cuando se navega con una asociación marcada como `ordered`, la colección resultante es de tipo `OrderedSet`. Asimismo, cuando se navega a través de más de una asociación, y una de ellas es marcada como `ordered`, la colección resultante es de tipo `Sequence`. Muchas operaciones estándar tienen en cuenta el orden de la colección, como por ejemplo: `first()`, `last()` o `at()`.

## 2.7. Especificación de la semántica de las operaciones

UML permite especificar de diferentes maneras el comportamiento de un sistema. Una de las más usuales es mediante la especificación de operaciones asociadas a las clases, las cuales sirven para describir el comportamiento de los objetos del sistema. OCL permite especificar la semántica de las operaciones en términos de sus pre- y poscondiciones.

**Lectura recomendada**

El uso de pre- y poscondiciones e invariantes para la especificación de operaciones software ha sido común en informática por mucho tiempo, aunque fue popularizado por lo que Bertrand Meyer denomina "Diseño por Contrato". Véase: Bertrand Meyer (1997), *Object-Oriented Software Construction* (2.ª ed.). Prentice Hall.

El propósito de una precondición es indicar las restricciones que deben satisfacerse para que la operación pueda ejecutarse correctamente. El propósito de una poscondición es especificar las condiciones que deben satisfacerse tras la ejecución de la operación (supuesta su precondición al comenzar la ejecución). En otras palabras, las precondiciones representan los estados válidos del sistema para los que está garantizado que la operación puede ejecutarse correctamente, y si lo hace ha de terminar en un estado en el que se satisfaga la poscondición.

Obsérvese que las pre- y poscondiciones determinan **qué** ha de hacer una operación, pero no indican **cómo** tiene que hacerlo. De otra forma estaríamos detallando la implementación concreta del sistema, lo que no es deseable para un modelo, y menos aún para un modelo independiente de la plataforma (PIM).

> Por ejemplo, la poscondición de una operación que ordena un vector de enteros debe determinar que el vector ha de estar ordenado tras la ejecución de la operación, pero no tiene que indicar si la ordenación se realiza utilizando Quicksort, Inserción o cualquier otro método particular de ordenación.

En OCL las precondiciones se indican con la palabra reservada `pre` y las poscondiciones con `post`. La palabra reservada `result` representa la variable especial que almacena el resultado de una operación que devuelve un valor. La siguiente expresión OCL especifica el comportamiento de la operación `numParts()` que devuelve el número de piezas de una bandeja, describiendo lo que debe devolver tras su ejecución.

```
context Tray::numParts() : Integer
post: result = self.part->size()
```

OCL también permite una forma alternativa de modelar consultas simplemente especificando el valor de la consulta como el cuerpo de la operación, con la cláusula `body`:

```
context Tray::numParts() : Integer
body: self.part->size()
```

La ausencia de una precondición (como es el caso en las expresiones anteriores) equivale a indicar que cualquier estado inicial es válido para la operación. De la misma forma, es posible especificar más de una pre- o poscondición para una misma operación, en cuyo caso todas han de cumplirse.

En una poscondición, además de la variable especial `result`, puede usarse el sufijo `@pre` para referirse al valor de una variable justo antes de la ejecución de la operación, es decir, el valor que tenía al evaluarse la precondición. También existe la operación `oclIsNew()` para consultar si un determinado objeto no existía antes de la ejecución de la operación; en otras palabras, es nuevo y lo ha creado la operación. El sufijo `@pre` puede ser útil, por ejemplo, para especificar el comportamiento de la operación `get()` de las bandejas:

```
context Tray::get() : Part
pre:  self.part->notEmpty()
post: result = self.part@pre->first() and self.part->excludes(result)
```

**Web recomendada**

OCL define numerosas operaciones sobre colecciones (`first()`, `excludes()`, `etc`.). La definición de cada una de ellas puede consultarse en el propio estándar de OCL: www.omg.org/spec/OCL/2.3.1

En este caso hemos pedido que la pieza que obtengamos sea la primera de la bandeja. También sería posible indicar que podemos obtener cualquiera de ellas:

```
context Tray::get() : Part      --- alternative specification
pre:  self.part->notEmpty()
post: self.part@pre->size() = self.part->size() + 1
      and self.part@pre->includes(result)
      and self.part->excludes(result)
```

De forma similar, una posible especificación OCL de la operación `put()` es la siguiente:

```
context Tray::put(p : Part)
pre:  self.part->size() < self.capacity and self.part->excludes(p)
post: self.part = self.part@pre->append(p)
```

Esta especificación hace uso de la operación `append()` sobre colecciones ordenadas. En el caso de que queramos no fijar que la operación coloque la pieza al final de la bandeja, la especificación podría realizarse de la siguiente forma:

```
context Tray::put(p:Part)     --- alternative specification
pre:  self.part->size() < self.capacity and self.part->excludes(p)
post: self.part->size() = self.part@pre->size() + 1
      and self.part->includes(p)
```

Es importante comentar que una pre- o poscondición puede estar especificada de forma incompleta. Por ejemplo, las poscondiciones anteriores de la operación `put()` no indican explícitamente que el resto del `Tray` no cambia.

## 2.8. Invocación de operaciones y señales

OCL dispone de un operador denominado `hasSent` (abreviado "`^`") para indicar que se ha llevado a cabo una comunicación entre dos objetos. Por comunicación OCL entiende la emisión de una señal, o la invocación de una operación. Este operador solo puede usarse en las poscondiciones de las operaciones.

Por ejemplo, la expresión `self.out^put(h)` afirma que durante la ejecución de la operación `genHandle()` del generador de mangos se ha debido invocar la operación `put()` de su bandeja de salida con el mango `h`. En otro caso, la expresión es `false`. Por ejemplo:

```
context       HandleGen::generate1()
post: let h : Handle = … in self.out^put(h)
```

La clausula `let...in...` permite definir variables que pueden usarse varias veces en una misma expresión OCL.

También es posible dejar sin especificar el valor concreto de los parámetros de una invocación, indicando que solo nos importa comprobar que se ha invocado la operación, independientemente de los parámetros que se hayan usado. Esto se hace usando un signo de interrogación (`"?"`) como nombre del parámetro:

```
context HandleGen::generate2()
post: … self.out^put(? : Handle)  …
```

OCL define un tipo denominado `OclMessage` al que pertenecen todos los mensajes usados en la emisión de señales y la invocación de operaciones. También se define el operador `SentMessages` (abreviado "`^^`"), que devuelve el conjunto de mensajes enviados. Por ejemplo, en la poscondición de la operación `generate2()` anterior se podría indicar que se ha invocado solo un mensaje `put()`, como sigue:

```
self.out^^put(? : Part)->size() = 1
```

Podemos completar las especificaciones de las operaciones `generate()` de `HandleGen` y `HeadGen` proporcionando las precondiciones adecuadas y asegurándonos en las poscondiciones de que las piezas devueltas son objetos nuevos.

```
context HeadGen::generate()
pre:  self.counter > 0 and self.out.numParts() < self.out.capacity
post: let msg : OclMessage = self.out^put(? : Head)  in
        counter = counter@pre - 1  and  msg.p.isOclNew()

context HandleGen::generate()
pre:  self.counter > 0 and self.out.numParts() < self.out.capacity
post: let msg : OclMessage = self.out^put(? : Handle)  in
        counter = counter@pre - 1  and  msg.p.isOclNew()
```

Obsérvese cómo se accede al parámetro del mensaje usando el nombre definido para él en la operación (en este caso el parámetro de `put()` se denominaba `p`, y por tanto accedemos a él como `msg.p`).

Igualmente, podríamos acceder a las secuencias de mensajes `get()` y `put()` usados por la máquina ensambladora en su operación `assemble()` y comprobar que se han emitido exactamente dos mensajes `get()` y un `put()`, y que el parámetro del `put()` es precisamente de tipo `Hammer`:

```
context Assembler::assemble()
pre: -- hay piezas en las 2 bandejas de entrada
      self.in->forAll(numParts()>0)
      and -- la bandeja de salida no está llena
      self.out.numParts() < self.out.capacity
post: let t1 : Tray =self.in->asSequence()->first() in  -- una bandeja
      let t2 : Tray =self.in->asSequence()->last() in-- otra bandeja
      let msg : OclMessage = self.out^put(? : Part) in -- emitido un put()
      let h : Hammer = msg.p in       -- el arg. del put es un martillo
         self.in^^get()->size() = 2   -- se han emitido dos get()
     and t1^get()  and  t2^get()     -- un get() con cada bandeja
     and m1.hasReturned()        -- ambos retornaron correctamente
     and m2.hasReturned()
     and self.out^^put(? : Part)->size() = 1 -- se ha emitido un solo put
     and h.isOclNew()               -- el martillo del put es nuevo
     and not h.isPolished           -- y no estaba pulido
```

Obsérvese el uso en esta expresión del operador `^^` para referirse a todos los mensajes de un tipo enviados o recibidos a un determinado tipo de objetos. Obsérvese también cómo se accede al parámetro del mensaje (`msg.p`) usando el nombre definido para él en la operación (`p`). Por último, obsérvese el uso de las operaciones `first()` y `last()`. Sobre estructuras ordenadas (`Sequence` y `OrderedSet`), estas operaciones proporcionan, respectivamente, el primero y el último de sus elementos. Dado que las bandejas de entrada de una máquina son un conjunto no ordenado, debemos aplicar antes la operación `asSequence()` (o `asOrderedSet()` dependiendo del caso) para obtener el elemento deseado. Dado que por otras restricciones OCL establecidas sabemos que una máquina ensambladora tiene exactamente dos máquinas generadoras como máquinas colocando elementos en sus bandejas de entrada, nos basta con coger una y otra y estará asegurado el que sean de elementos distintos.

Con los operadores `^` y `^^` sabemos cómo referirnos a las señales emitidas y operaciones invocadas durante la ejecución de una operación. Las señales son comunicaciones unidireccionales, y no devuelven ningún valor; sin embargo, las operaciones implican una invocación y un retorno (o terminación), que contiene el valor devuelto por la operación.

OCL ofrece dos funciones para gestionar el retorno de una operación y el valor devuelto por ella: `hasReturned()` y `result()`. Ambas pertenecen a la clase `OclMessage`. Por supuesto, la operación `result()` solo devuelve un valor bien definido si la operación llamada ha terminado y devuelto un valor. Esto es precisamente lo que se comprueba con `hasReturned()`. Si la operación no ha terminado, `hasReturned()` devuelve `false` y `result()` está indefinido (devuelve `invalid`).

Como parte de la especificación de la máquina pulidora, usando estas funciones podemos especificar que se debe haber enviado un mensaje `get()` a la bandeja de entrada, que esta ha devuelto un martillo y que este ha sido enviado en un mensaje `put()`, tras pulirlo, a la bandeja de salida. La siguiente expresión OCL especifica esta operación:

```
context Polisher::polish()
pre:  (self.in.numParts()>0) and (self.out.numParts()<self.out.capacity)
post: let   inM : OclMessage = self.in^get() in
      let   h: Hammer= inM.result() in -- el get() nos da un martillo
         inM.hasReturned()    -- mensaje enviado y respuesta recibida
         and h.isPolished        -- el martillo ha sido pulido y está listo
         and self.out^put(h)  -- el objeto devuelto por get() es el
                  -- mismo que luego se envía a la bandeja de salida
```

## 2.9. Algunos consejos de modelado con OCL

En general, modelar un sistema no es una tarea simple. Aparte de la complejidad intrínseca que pueda tener la especificación del propio sistema, también nos encontramos con la dificultad "accidental" que ocasiona el uso de notaciones como UML u OCL. Por ejemplo, en este apartado hemos visto cómo una misma condición sobre un modelo puede especificarse con varias expresiones OCL equivalentes (desde contextos distintos, usando `self` o `allInstances()`, por ejemplo). Los siguientes apartados contienen algunas recomendaciones sobre el uso de UML y OCL para la especificación de sistemas.

### 2.9.1. Uso de la representación gráfica de UML frente a OCL

Hemos visto que hay muchas restricciones que admiten dos representaciones, una de forma gráfica en UML y otra como invariantes en OCL. Los ejemplos más típicos de este tipo de restricciones son las multiplicidades de los extremos de las asociaciones.

**R1**: Expresar las restricciones de forma gráfica siempre que sea posible.

En otras palabras, es mejor utilizar OCL solo para las expresiones que no puedan expresarse de ninguna forma con la notación gráfica de UML.

### 2.9.2. Consistencia de los modelos

Cuando estamos definiendo los modelos es posible que introduzcamos demasiadas restricciones, o que las que especifiquemos sean incompatibles entre sí. Eso es algo que debemos tener muy en cuenta.

**R2**: Asegurar que el modelo propuesto es consistente.

Una buena práctica es comprobar, siempre que introduzcamos una nueva restricción, que hay al menos una instancia del modelo que puede satisfacer todas las restricciones de integridad al mismo tiempo.

### 2.9.3. Compleción de los diagramas UML

Es importante incluir en los diagramas UML toda la información de los elementos de un modelo, y especialmente las asociaciones:

**R3**: Especificar de forma explícita los nombres de todos los extremos de las asociaciones entre las clases, así como sus multiplicidades.

En UML, cuando hay más de una asociación entre dos clases es obligatorio asignarles nombres a los extremos de las asociaciones para poder distinguirlos. Sin embargo, cuando solo hay una asociación entre dos clases, UML no obliga a hacerlo. En este caso, cuando OCL necesita referirse a los objetos que están relacionados con otros a través de una asociación sin nombre, utiliza la convención de usar como nombre del extremo de la asociación el nombre de la clase pero en minúscula. Esto lo hemos ilustrado omitiendo el nombre de los extremos de la asociación entre las clases `Tray` y `Part`. De todas formas, es conveniente indicar el nombre de todos los extremos de las asociaciones entre las clases.

**Nota**

Recordad que el nombre de un extremo de asociación representa el rol que los objetos de esa clase tienen en la asociación, y suele expresarse con un nombre (y no con un verbo). Desde el punto de vista de OCL, ese nombre se convierte además en el nombre de un atributo de la clase del extremo opuesto, por lo cual ha de tener sentido como nombre de atributo.

Igualmente, es importante especificar de forma explícita la multiplicidad de toda asociación. Por defecto, UML supone que la multiplicidad de un extremo de asociación es 1 a menos que se indique otro valor. Sin embargo, es un error común pensar que la multiplicidad por defecto es `"*"`. Para evitar malentendidos es preferible no dejar nada sin detallar de forma explícita.

Por supuesto, es importante detallar otras características importantes de las asociaciones, como por ejemplo si están ordenadas (`isOrdered`), si no permiten que un mismo elemento aparezca dos veces (`isUnique`), etc. Es importante desde el punto de vista del modelado que nos planteemos estas cuestiones para cada asociación.

### 2.9.4. La importancia de ser exhaustivo

Además de tener muy en cuenta las propiedades de las asociaciones y de las clases que conforman el modelo (cuáles son abstractas y cuáles no, etc.) es preciso también contemplar todas las restricciones de integridad de los modelos. En este apartado hemos especificado varios invariantes sobre el modelo usando OCL. Sin embargo, nos surge la duda de si habrá más que hayamos podido olvidar, o no hemos tenido en cuenta.

**R4**: Identificar todas las restricciones que deben cumplir los modelos que han de representar modelos válidos del sistema.

Esta no es desde luego una tarea fácil, y para la que no existen tampoco recetas mágicas. Uno de los problemas habituales es que el modelador suele olvidarse de incluir restricciones que especifican situaciones obviamente imposibles en la realidad, pero que el modelo UML sí permite. Es por ello por lo que es importante tratar de analizar todas las asociaciones por si pudieran darse ciclos no deseados, analizar los valores de los atributos por si requiriesen ser restringidos, las del modelo completo por si fueran precisas restricciones de unicidad (por ejemplo, que el valor de algunos atributos sea único), los valores iniciales de los atributos, etc.

**Nota**

Suele ser habitual olvidarse de imponer restricciones del tipo "dos elementos de una planta de montaje no pueden compartir una misma posición física".

### 2.9.5. Navegación compleja

A la hora de escribir expresiones OCL solemos "navegar" por las asociaciones del modelo para referirnos a los elementos que están relacionados con uno dado.

En OCL sería posible escribir todas las expresiones comenzando desde un contexto cualquiera, pero eso podría producir expresiones muy largas y complejas de entender, ya que en algunos casos obligaría a incluir caminos de navegación muy largos para referirse a objetos relacionados de forma distante.

**R5**: Hay que tratar de escoger el mejor contexto para cada expresión, y de usar expresiones de navegación lo más sencillas posibles.

A la hora de escoger el contexto de un invariante, hay que tener en cuenta estas indicaciones:

- Si el invariante restringe el valor de un atributo de una clase, esa clase constituye el contexto óptimo para el invariante.

- Si el invariante restringe el valor de los atributos de varias clases, es mejor escoger como contexto la clase desde donde sea más fácil expresar la restricción OCL.

- Es mejor tratar de usar expresiones con `self` en vez de con `allInstances()` siempre que se pueda, para simplificar las expresiones OCL.

- Utilizar variables y operaciones auxiliares (definidas con cláusulas `def`) para simplificar las expresiones que resulten muy complicadas. Este tipo de variables pueden verse como una generalización de la expresión `let` para variables que se utilizan en más de una sola expresión OCL.

### 2.9.6. Modularización de invariantes y condiciones

Suele ser habitual expresar ciertos invariantes, precondiciones y poscondiciones con muchas restricciones unidas por cláusulas `and`. En vez de esto, y dada la facilidad para componer condiciones mediante conjunción que tiene OCL (basta con incluirlas unas tras otras), es mejor descomponer las condiciones en expresiones más pequeñas que pueden combinarse luego mediante conjunciones, y así modularizar las especificaciones.

**R6**: Modularizar los invariantes y las condiciones tanto como sea posible.

El hecho de definir expresiones más simples y manejables va a mejorar su legibilidad, comprensibilidad y mantenimiento notablemente.

## 2.10. Conclusiones

Tradicionalmente, el modelado de sistemas software ha sido sinónimo de especificación de diagramas. La mayoría de los modelos consistían en figuras con forma de cajas y flechas, y con algún texto que las acompañaba. La información recopilada por este tipo de modelos tiene la tendencia a ser incompleta, imprecisa, informal e, incluso a veces, inconsistente.

Es por ello por lo que es preciso contar con lenguajes precisos y con base formal para especificar apropiadamente los modelos. Esto es incluso más importante cuando el modelo no ha de ser leído por humanos, sino por máquinas que han de llevar a cabo todas las misiones que define MDE: generar automáticamente el código final de la implementación, las pruebas, comprobar la consistencia del modelo, simularlo, analizar sus propiedades, etc. El problema es que automatizar este trabajo solo es posible si el modelo contiene toda la información necesaria. Una máquina no puede interpretar reglas escritas en lenguaje humano, o realizar suposiciones por muy obvias que nos parezcan a nosotros[3].

(3)Por ejemplo, que los contadores de los generadores han de ser positivos, o que dos máquinas no pueden estar situadas en las mismas coordenadas en la planta.

Sin embargo, un modelo escrito en un lenguaje que usa expresiones puede no ser fácil de escribir, entender, modificar o mantener. Por ello la comunidad de MDA ha optado por la combinación de UML y OCL como lenguajes para especificar modelos de la forma más intuitiva y precisa a la vez. En este apartado hemos presentado los principales conceptos y mecanismos del lenguaje OCL, usando ejemplos para ello, y hemos dado algunas indicaciones de cómo usar OCL de la mejor forma posible. A continuación vamos a estudiar cómo es posible también definir nuevos lenguajes de modelado. Esto es necesario cuando UML, al ser un lenguaje de propósito muy general, no proporciona la notación adecuada para modelar nuestros sistemas concretos.

# 3. Lenguajes específicos de dominio

## 3.1. Introducción

En los apartados anteriores hemos discutido la necesidad de describir, de forma precisa y a un nivel de abstracción adecuado, todos los aspectos de un sistema que son relevantes desde un determinado punto de vista. También hemos destacado la importancia de utilizar el lenguaje más adecuado para la descripción de dichos aspectos, que cuente con la expresividad más apropiada para especificarlos. Para ello pueden usarse tanto lenguajes de modelado de propósito general (por ejemplo UML) como también lenguajes específicos de dominio (o DSL, *domain-specific languages*).

**Experto del dominio**

Un experto del dominio es aquella persona que es especialista en el dominio del problema, aunque no necesariamente en el dominio de la aplicación. Por ejemplo, un banquero conoce bien la terminología que se utiliza en la banca y un marinero la de la navegación, aunque ninguno de ellos puede tener los conocimientos para diseñar una aplicación bancaria o de navegación usando UML o implementarla haciendo uso de tecnologías Java o .NET.

Un **lenguaje específico de dominio (DSL)** es un lenguaje que proporciona los conceptos, notaciones y mecanismos propios de un dominio en cuestión, semejantes a los que manejan los expertos de ese dominio, y que permite expresar los modelos del sistema a un nivel de abstracción adecuado.

Puede haber DSL de muy distinta naturaleza y grado de complejidad, desde muy técnicos[4] a aquellos de muy alto nivel orientados a personas de negocio o científicos[5].

(4)Por ejemplo, los que permiten configurar una red de conmutación de paquetes o especificar los parámetros de una línea de productos software.

(5)Por ejemplo, los que permiten modelar procesos de negocio, cadenas de producción de una fábrica de ensamblaje, moléculas de ADN, etc.

Los DSL proporcionan los conceptos esenciales de un dominio de aplicación como elementos de primera clase de dicho lenguaje.

***Domain specific modeling***

La proliferación de los DSL ha propiciado la aparición del término DSM (*domain specific modeling*) que se refiere a la disciplina dentro de MBE que usa DSL para modelar dominios de aplicación concretos.

De esta forma, los DSL van a permitir a los expertos del dominio expresar el modelo de su sistema usando el vocabulario que normalmente utilizan, y de forma independiente de las plataformas de implementación.

**Web recomendada**

Consúltese http://www.dsmforum.org

En general, los DSL se diseñan y desarrollan para que sean los expertos del dominio quienes modelen sus propios sistemas, y que luego esos modelos puedan ser procesados de forma automática por ordenadores.

Esto obliga a conjugar simplicidad con precisión: los modelos han de ser fácilmente entendibles y manejables por los expertos del dominio, pero a su vez han de ser lo suficientemente precisos para que sean ejecutables y procesables por los sistemas de información.

La conexión entre los modelos expresados en un DSL y los sistemas en donde se ejecutan y analizan se realiza mediante las transformaciones de modelos, que son las encargadas de convertir los modelos en los correspondientes artefactos software (código fuente, programas, componentes, aplicaciones, etc.) en las plataformas software destino.

**Ved también**

Las transformaciones de modelos se estudian con más detalle en el apartado 4.

Este apartado está dedicado a dichos lenguajes, sus características y propiedades, así como a las formas actualmente existentes para definirlos de forma que describan correctamente los modelos de nuestros sistemas.

### 3.2. Componentes de un DSL

Aunque, como hemos mencionado antes, existen muy variados tipos de DSL dependiendo de los dominios para los que son creados, todos comparten una serie de componentes comunes que pasamos a describir.

#### 3.2.1. Sintaxis abstracta

La **sintaxis abstracta** de un lenguaje describe el vocabulario con los conceptos del lenguaje, las relaciones entre ellos, y las reglas que permiten construir las sentencias (programas, instrucciones, expresiones o modelos) válidas del lenguaje.

Dichas reglas restringen los posibles elementos de los modelos válidos, así como aquellas combinaciones entre elementos que respetan las reglas semánticas del dominio.

Como comentamos en el apartado 1, los **metamodelos** constituyen la forma más natural de describir la sintaxis abstracta de un lenguaje.

> El metamodelo de la figura 8 del primer apartado representaba los conceptos que se manejan a la hora de diseñar una cadena de montaje de una fábrica de martillos y las relaciones que pueden establecerse entre ellos para formar modelos válidos. En el apartado 2 vimos cómo también es preciso incluir restricciones OCL para especificar algunas reglas que no pueden describirse con una notación gráfica como la que proporciona UML.

Es importante señalar que la sintaxis abstracta es independiente de la **notación** que se use para representar los modelos (sea textual o gráfica) y de la semántica o **interpretación** que se les quiera dar, tanto a nivel de la estructura del sistema como de su comportamiento.

#### 3.2.2. Sintaxis concreta

Todos los lenguajes (no solo de modelado o de programación) cuentan con una notación que permite la representación y construcción de sus modelos.

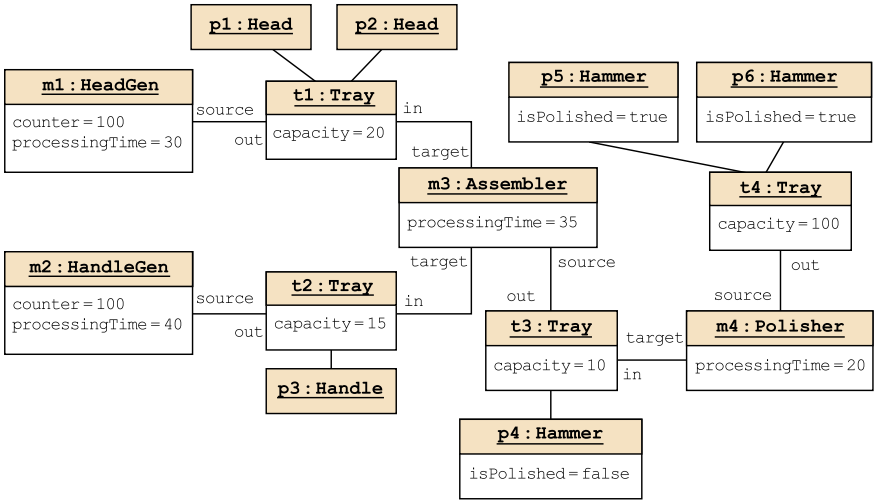
La **sintaxis concreta** de un lenguaje define la notación que se usa para representar los modelos que pueden describirse con ese lenguaje.

Principalmente hay dos tipos de sintaxis concretas: textuales y gráficas.

La **sintaxis gráfica** (o **visual**) describe los modelos de forma diagramática, usando símbolos para representar sus elementos y las relaciones entre ellos.

Por ejemplo, en UML se utilizan cajas rectangulares para representar clases e instancias, y líneas para representar asociaciones y enlaces. Así, las figuras 7, 8 y 9 del primer apartado representan visualmente un modelo y su metamodelo (recordemos que un metamodelo es también un modelo). La figura 11 muestra la representación de la figura 9 usando la sintaxis concreta de UML.

Figura 11. El modelo de la cadena de montaje de martillos de la figura 9 en UML




La mayor ventaja de una sintaxis gráfica es que permite representar mucha información de forma intuitiva y fácilmente comprensible. Uno de sus principales inconvenientes es que no permite especificar los sistemas con demasiado nivel de detalle, pues los diagramas se complican excesivamente. Además, la expresividad de las notaciones gráficas suele ser limitada; por ello suelen complementarse con especificaciones descritas en notaciones textuales, como vimos en el apartado anterior con UML y OCL.

La **sintaxis textual** permite describir modelos usando sentencias compuestas por cadenas de caracteres, de una forma similar a como hacen la mayoría de los lenguajes de programación.

**El modelo de la figura 11 representado usando la notificación HUTN**

Un ejemplo de notación textual es la que usa OCL, y que vimos en el apartado anterior. XMI es otra notación textual, que usa XML para representar modelos. OMG también ha definido otra notación textual para describir modelos y metamodelos, denominada HUTN (*human-usable textual notation*), que no usa XML por lo farragosa y compleja que resulta ser esta notación para ser entendida y utilizada por humanos. Usando HUTN, el modelo de la figura 11 se representa como sigue:

**Consulta recomendada**

**OMG** (2004). *Human-Usable Textual Notation* (HUTN) Specification v1.0. OMG doc. formal/04-08-01.

```
HeadGen "m1" { counter: 100 processingTime: 30 out: "t1" }
HandleGen "m2" { counter: 100 processingTime: 40 out: "t2" }
Assembler "m3" { processingTime: 35 in: "t1", "t2" out: "t3" }
Polisher "m4" { processingTime: 20 in: "t3" out : "t4" }
Tray "t1" { capacity: 20 source: "m1" target: "m3"
          part: Handle "p1" {}, Handle "p2" {} }
Tray "t2" { capacity: 15 source: "m2" target: "m3" part: Handle "p3" {}
Tray "t3" { capacity : 10 source : "m3" target: "m4"
          part: ~isPolished Hammer "p4" {} }
Tray "t4" { capacity : 100 source : "m4"
          part: isPolished Hammer "p5" {}, isPolished Hammer "p6" {} }
```

Obsérvese cómo se trata de facilitar en HUTN la comprensión por parte de los usuarios (humanos) de las especificaciones, mediante algunos mecanismos concretos. Por ejemplo, los nombres de los objetos aparecen detrás de sus tipos; o los atributos booleanos que pueden mostrarse como adjetivos de los objetos delante de ellos. Así, en el código fuente que aparece arriba se usa `isPolished Hammer"p5"` para indicar que el atributo isPolished de ese martillo toma el valor `true`, mientras que `~isPolished Hammer "p4"` indica que el valor de `isPolished` es `false` en el objeto cuyo nombre es `p4`.

### Ejercicio propuesto

Tratad de leer y comprender la descripción XMI que genera cualquier herramienta UML del modelo representado en la figura 11, y comparadla con su representación en HUTN mostrado en el código fuente que aparece arriba.

Además de los definidos por OMG, existen otros lenguajes que permiten definir DSL y dotarlos con una sintaxis concreta textual. Uno de ellos es Xtext, un proyecto de Eclipse de amplio uso por los mecanismos y herramientas que proporciona para definir editores y validadores de forma automática.

**Web recomendada**

Para saber más sobre Xtext consultad su página web: http://www.eclipse.org/Xtext

La principal ventaja de las notaciones textuales es que permiten describir expresiones y relaciones complejas entre los elementos del modelo, con un mayor nivel de detalle. Sin embargo, los modelos descritos así enseguida se convierten en inmanejables e incomprensibles por los humanos, debido a su extensión y complejidad.

En la práctica, la mejor solución es utilizar una combinación de notaciones gráficas y textuales, con lo que es posible obtener las ventajas de ambas opciones. De esta forma la representación de alto nivel del sistema se realiza utilizando una notación gráfica, mientras que las propiedades de grano fino, que es preciso describir con mayor detalle, se especifican utilizando una notación textual.

Comentamos anteriormente que la sintaxis abstracta debe ser independiente de cualquier notación y de cualquier semántica que se le asigne. Algo similar debería ocurrir con la sintaxis concreta, pero en la práctica esto es más complicado: aunque en teoría una sintaxis concreta es independiente de cualquier sintaxis abstracta y de cualquier semántica (significado), los humanos normalmente solemos asociar significados concretos a símbolos que vemos, sobre todo en ciertos contextos.

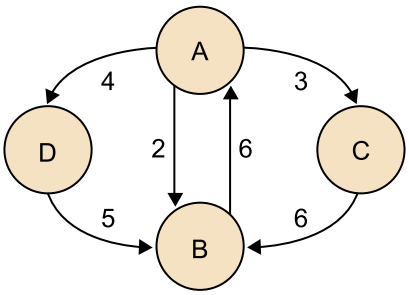
**Nota**

A la acción de asignar un significado a un símbolo dado se la conoce como "interpretación".

Por ejemplo, un rectángulo suele significar una clase UML o una entidad en un diagrama de entidad-relación, mientras que las líneas representan relaciones. Por su parte, los rombos se asocian a decisiones en los modelos de flujo, tanto en UML como en otras notaciones que representan procesos (Flowchart, BPMN, etc.). Un ejemplo donde una imagen representa claramente a la clase asociada a ella ocurre con el modelo de la figura 9: a un elemento de tipo `Hammer` se le ha asignado la figura de un martillo como su sintaxis gráfica.

El problema surge cuando dos personas asignan dos significados diferentes a un mismo símbolo.

Figura 12. ¿Qué significa este modelo?




El modelo mostrado en la figura 13 admite diversas interpretaciones, todas ellas válidas:

**1)** Distancias entre ciudades expresadas en kilómetros.

**2)** Transiciones entre estados de un diagrama de estados, donde los números indican el tiempo en milisegundos que tarda en llevarse a cabo cada transición.

**3)** Flujos de migración mensuales entre aeropuertos, expresados en millares de personas.

**4)** Deudas entre entidades bancarias, expresadas en millones de euros.

**Consulta recomendada**

Este ejemplo se debe al profesor Gonzalo Génova (UC3M): "Modeling and Metamodeling in Model Driven Development (Parte 1: What is a model: syntax and semantics"). http://www.ie.inf.uc3m.es/ggenova/Warsaw/Part1.pdf

**Ejercicio propuesto**

Definid cuatro metamodelos, cada uno representando la sintaxis abstracta de las cuatro interpretaciones diferentes del modelo mencionadas en el texto y compararlos entre sí. ¿Cuáles son las diferencias entre ellos?

### 3.2.3. Semántica

Tanto la sintaxis abstracta como la sintaxis concreta permiten describir modelos de un sistema. Sin embargo, como ejemplifica la figura 12, la información que proporcionan estas dos sintaxis puede no ser suficiente para comprender de forma precisa el significado de un modelo, ni para poder razonar sobre él y sobre su comportamiento. Por tanto, la definición de cualquier lenguaje debe proporcionar también información sobre su **semántica**, además de sobre sus sintaxis abstracta y concreta. Sin esta información puede que dos personas (o

herramientas) interpreten y manipulen un mismo modelo de dos formas distintas, lo que puede llevar a malentendidos, razonamientos incorrectos sobre el sistema o a implementaciones erróneas.

Las operaciones admisibles sobre el modelo de la figura 12 (así como su comportamiento) dependen mucho de la interpretación que hagamos de él. Así, si el modelo representara la deuda entre entidades bancarias, podríamos reducir ese mismo modelo a otro equivalente, en el que el banco A deba 3 millones a B, D deba 1 a B, y el banco C deba 3 a B, mediante sucesivas redenciones de las deudas (si A debe 2 a B, y B debe 6 a A, eso es equivalente a que B deba 4 a A). Sin embargo, esto no tendría sentido alguno si el modelo de la figura 12 representara transiciones de una máquina de estados, o distancias entre poblaciones. Tampoco tendría sentido en este caso que el grafo admitiera arcos de un nodo a sí mismo, algo que sí tendría mucho sentido si el grafo representara un diagrama de estados.

En general, lo que **parece representar** un modelo no es lo mismo que lo que **realmente significa**.

Hay varias formas de definir la semántica de un lenguaje, cada una más apropiada al tipo de uso que se le pretenda dar a esa semántica.

- La semántica **denotacional** traduce (o transforma) cada sentencia o modelo del lenguaje en una sentencia o modelo de otro lenguaje que tiene una semántica bien definida. Esta forma de dar semántica corresponde a una interpretación o compilación del modelo, pero donde el lenguaje destino no suele ser un lenguaje de modelado o programación, sino un formalismo matemático (denominado *dominio semántico*). Para transformar un modelo de un lenguaje en un modelo de otro lenguaje con una semántica bien definida se utilizan las transformaciones horizontales entre modelos, que se verán en el apartado 4.

**Nota**

Para dotar de semántica denotacional a un DSL suelen utilizarse transformaciones de modelos que convierten los modelos del lenguaje en modelos del formalismo destino.

- La semántica **operacional** describe el comportamiento de los modelos del sistema de forma explícita, mediante un lenguaje de acciones o definiendo operaciones y especificando su comportamiento.

En el apartado 2 vimos un ejemplo de semántica operacional para los modelos de cadenas de montaje, mediante el cual enriquecíamos el metamodelo con las operaciones que admitían los diferentes elementos, y definíamos el comportamiento de dichas operaciones mediante pre- y poscondiciones expresadas en OCL.

En UML también es posible especificar comportamiento mediante acciones, y poder ejecutar los modelos como si fueran programas. Esto es solo posible si la descripción del comportamiento está suficientemente detallada, y además existe una máquina o plataforma que sea capaz de ejecutar tales especificaciones de comportamiento. La primera propuesta para especificar un modelo UML ejecutable vino de la mano de Mellor y Balcer. Posteriormente, Ed Seidewitz ha definido un subconjunto de UML (denominado fUML) con semántica bien definida y un lenguaje de acciones para él (denominado Alf, *action language for fUML*), que han sido recientemente estandarizados por la OMG. La sintaxis concreta de Alf es textual, una mezcla de OCL con las instrucciones de asignación y de iteración comunes a los lenguajes de programación imperativos.

**Consulta recomendada**

**Mellor, S; Balcer, M.** (2002). *Executable UML: A foundation for model-driven architecture*. Addison Wesley.

**Webs recomendadas**

http://www.omg.org/spec/FUML/Current

http://www.omg.org/spec/ALF/Current

Implementación de fUML: http://fuml.modeldriven.org

Parser para Alf: http://lib.modeldriven.org/MDLibrary/trunk/Applications/Alf-Reference-Implementation/dist/

Presentaciones: www.slideshare.net/seidewitz

- La semántica **axiomática** describe un conjunto de reglas, axiomas o predicados que deben cumplir las sentencias del lenguaje (es decir, los modelos bien formados), y las interpreta en una lógica donde es posible razo-

nar sobre ellas. La lógica de Hoare para lenguajes de programación es un ejemplo de este tipo de semántica, así como la lógica de reescritura para gramáticas de grafos.

Por ejemplo, para modelos como el de la figura 13, suponiendo que modelen deudas entre entidades bancarias, podemos pensar en los siguientes cuatro axiomas (o reglas) que definen su comportamiento y permiten derivar unos modelos a partir de otros:

**1)** [Suma] Dos arcos que comparten el mismo origen y el mismo destino pueden ser reemplazados por un arco cuya etiqueta es la suma de las dos.

**2)** [Resta] Dos arcos con orígenes y destinos opuestos pueden reemplazarse por un arco orientado como el de mayor valor y etiquetado con la diferencia de las etiquetas.

**3)** [Nulo] Un arco etiquetado con 0 puede ser eliminado.

**4)** [Ciclos] Las etiquetas de un conjunto de arcos que formen un ciclo pueden ser incrementadas o decrementadas en una misma cantidad (la regla 2 es un caso particular de esta, teniendo en cuenta también la regla 3; en otras palabras, la regla 2 puede ser derivada de las reglas 3 y 4).

Estas reglas definen el comportamiento del sistema mediante un sistema de reescritura de grafos.

Hay numerosas herramientas que permiten especificar el comportamiento de un sistema en términos de reescritura o transformación de grafos, como por ejemplo GrGen.Net, ATOM3, FuJaBa, AGG, PROGRES, MotMot, e-Motions o Henshin. Cada una permite además realizar diferentes tipos de análisis (confluencia, terminación, alcanzabilidad), simular o ejecutar los modelos o incluso generar código directamente para diferentes plataformas como C++ o .NET.

A la hora de especificar la semántica de un lenguaje no es solo esencial que sea precisa, sino que además sea útil para realizar las labores que se pretenden con ella. Así, una descripción matemática del sistema será muy útil para razonar sobre el sistema si los diseñadores pueden entenderla, puesto que permite realizar los análisis que se pretenden, y se cuenta con herramientas para realizar dichos análisis. En otro caso su utilidad no estaría en absoluto justificada.

Por ejemplo, ¿para qué se desea una semántica de nuestro sistema definida en términos de redes de Petri estocásticas si no somos expertos en ese formalismo o no tenemos herramientas para manejar este tipo de especificaciones? Igualmente, contar con una especificación operacional del sistema en término de pre- y poscondiciones en OCL no es muy útil si lo que queremos es analizar algunas propiedades como el rendimiento (*throughput*) del sistema, su fiabilidad u otras características de calidad de servicio.

En general, no hay una forma mejor o peor de especificar la semántica de un lenguaje, sino que depende mucho del uso que queramos hacer de esa semántica y de nuestro conocimiento sobre ella.

### 3.2.4. Relaciones entre sintaxis abstracta, sintaxis concreta y semántica

Para finalizar esta sección destacaremos algunas relaciones entre la sintaxis abstracta, la sintaxis concreta y la semántica de un lenguaje, así como buenas prácticas para su uso correcto.

- Mientras que la sintaxis abstracta especifica cuáles son los modelos válidos, la sintaxis concreta permite representar dichos modelos. Por su parte, la semántica les asigna un significado preciso y no ambiguo.

- En MBE, la forma natural de definir la sintaxis abstracta de los DSL es mediante metamodelos.

- La relación entre la sintaxis concreta de un lenguaje se suele definir a partir de su sintaxis abstracta, mediante una relación (*concrete syntax mapping*) que asigna un símbolo de la sintaxis concreta a cada concepto de la sintaxis abstracta. Dicha asignación debe respetar la semántica que suelen tener los símbolos.

**Ejemplo**

Por ejemplo, el color rojo suele indicar peligro o prohibición, no es apropiado para representar permisos u obligaciones. Igualmente las líneas suelen representar arcos de un grafo o asociaciones entre elementos, y no deberían utilizarse para representar clases o entidades.

- Una misma sintaxis abstracta puede tener asociada más de una sintaxis concreta, como por ejemplo una textual y otra gráfica. En MDD, todos los DSL tienen al menos una sintaxis concreta, que es la que ofrece por defecto el lenguaje en el que están definidos sus metamodelos.

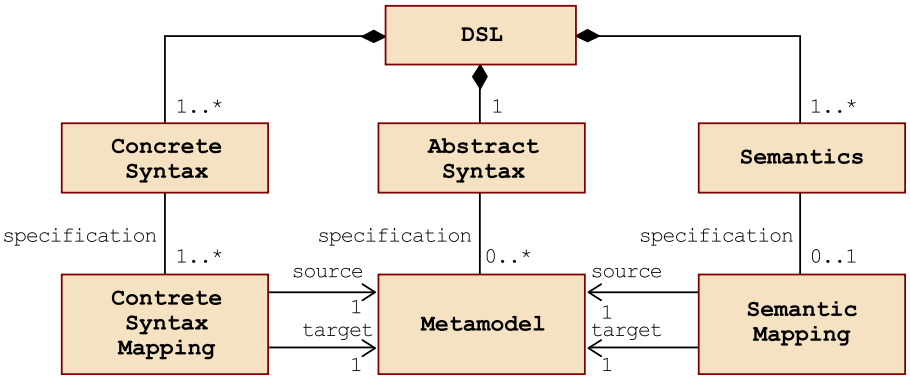
- El *concrete syntax mapping* suele estar definido por dos funciones: de la sintaxis concreta a la sintaxis abstracta (lo que se conoce como *parsing function*) y de la sintaxis abstracta a la concreta (*rendering function*).

- En cuanto a la semántica, a un lenguaje podemos asignarle varios significados, cada uno definido por su interpretación en un dominio semántico concreto. Cada interpretación va a venir descrita por una transformación (*mapping*) entre el metamodelo de su sintaxis abstracta y el metamodelo del dominio semántico destino, que asigna un significado a cada uno de los modelos válidos del DSL.

- El tipo de semántica que hay que definir para un lenguaje va a depender del uso que quiera hacerse de ella, y del tipo de razonamientos y herramientas disponibles en el dominio semántico destino. Siempre es posible definir diversas interpretaciones para un mismo lenguaje, aunque en este caso es preciso cuidar la *consistencia* entre ellas (de forma que todo lo que se infiera en un dominio semántico sea cierto también en el resto, y no haya contradicciones).

En MDE, los *mappings* suelen definirse e implementarse mediante transformaciones de modelos. La figura 13 muestra a modo de resumen todos estos conceptos de una forma gráfica. Las secciones siguientes ahondan un poco más en algunos aspectos concretos de ellos.

Figura 13. Partes de un DSL y las relaciones entre ellas




## 3.3. Metamodelado

Hemos visto la importancia que tienen los metamodelos para representar los conceptos de un lenguaje, las relaciones entre ellos y las reglas estructurales que restringen los posibles elementos de los modelos válidos. Es por ello por los que los metamodelos se convierten en artefactos esenciales de MDE.

Ya comentamos en el apartado 1 que los metamodelos son a su vez modelos, y por tanto vienen definidos por su meta-metamodelo. Según la propuesta de OMG, esta estructura recursiva (denominada arquitectura de metamodelado) termina en el nivel 3, pues los meta-metamodelos pueden usarse para definirse a sí mismos.

**Ved también**

Ver la figura 1 del apartado 1.

Esta arquitectura de metamodelado es una pieza clave en MDA. De hecho, la existencia de un meta-metamodelo como MOF es la que va a permitir definir de forma unificada todos los metamodelos de los distintos lenguajes de modelado que intervienen en MDA, y por tanto hacerlos compatibles entre sí.

Por supuesto, además de MOF hay otras propuestas para implementar las técnicas de MDE, como por ejemplo EMF (*eclipse modeling framework*) de Eclipse, DSL Tools de Microsoft, GME (*generic modeling environment*) de la Universidad de Vanderbilt, o la plataforma AMMA del INRIA y la Universidad de Nantes. Cada uno de ellos define su propio meta-metamodelo para definir sus lenguajes, como por ejemplo Ecore para EMF o KM3 para AMMA. La ventaja de que estos artefactos de software sean a su vez modelos permite su interconexión mediante transformaciones de modelos, logrando así tender puentes entre estos llamados "espacios técnicos" (*technical spaces*).

**Lectura recomendada**

**H. Brunelière; J. Cabot; C. Clasen; F. Jouault; J. Bézivin** (2010). "Bridging Eclipse and Microsoft Modeling Tools". *Towards Model Driven Tool Interoperability*. Proc. of ECMFA.

En cuanto a la especificación del metamodelo de un sistema, es una tarea que no es simple y que en general debe ser acometida por equipos multidisciplinares que combinen expertos del dominio de la aplicación que conocen el vocabulario y notaciones habituales con expertos en modelado conceptual que sepan representar estos conceptos en términos de metamodelos concretos.

**Consultas recomendadas**

Para saber más sobre la definición de DSL se recomiendan las siguientes lecturas: Mernik y otros (2005); Strembeck; Zdun (2009), Kelly; Tolvanen (2008); Cook y otros (2007); Fowler (2011); Kleppe (2008).

Finalmente, siempre que se habla de metamodelos surge su relación con las **ontologías**, que son otro de los mecanismos disponibles en la actualidad para representar dominios de aplicación de forma independiente de las plataformas tecnológicas y de las implementaciones subyacentes. Desde nuestro punto de

vista, la principal diferencia entre un metamodelo y las ontologías es que estas últimas no solo permiten capturar los conceptos del dominio y las relaciones entre ellas, sino que también hacen uso de un sistema de inferencia (normalmente basado en lógica de primer orden) para dotar de cierta semántica a dichos elementos y razonar sobre ellos.

Por ejemplo, suelen distinguir entre diferentes tipos de asociaciones ("is-a", "is-part-of", "refines",...) con significados bien definidos.

Así, además de lo que puede expresarse con un metamodelo, con una ontología también es posible obtener nuevas relaciones derivadas de las existentes, comprobar la consistencia entre relaciones específicas, etc.

Por ejemplo, si un modelo A está relacionado con un modelo B mediante una asociación de tipo "is-a", y B está relacionado con un tercer modelo C por ese mismo tipo de asociación, entonces puede concluirse que A también está relacionado con C por una asociación "is-a", debido a la propiedad de transitividad de ese tipo de asociación.

**Nota**

Una ontología es un conjunto de conceptos, sus definiciones y las relaciones entre ellos, que permiten expresar ideas válidas en un determinado dominio.

Por otro lado, las ontologías no tienen por qué contar con lenguajes precisos y bien definidos para ser descritas, como le ocurre a los metamodelos, que han de estar descritos en el lenguaje de su meta-metamodelo, además de ser capaces de definir modelos válidos conforme a la sintaxis abstracta que definen.

**Lectura recomendada**

**Pidcock, Woody** (2003). *What are the differences between a vocabulary, a taxonomy, a thesaurus, an ontology, and a meta-model?* http://infogrid.org/wiki/Reference/PidcockArticle

En esta sección vamos a a definir primero la sintaxis concreta para metamodelos y daremos algunas propuestas. A continuación, describiremos una de estas propuestas, que es la que ofrece OMG para definir la sintaxis concreta de DSL internos mediante extensiones del metamodelo de UML, lo que permite definir notaciones gráficas haciendo uso de las herramientas de modelado UML existentes. Por supuesto, existen otras alternativas, y esta propuesta tiene ventajas e inconvenientes cuando la comparamos con ellas, las cuales trataremos de analizar a lo largo de esta sección.

### 3.3.1. Sintaxis concreta para metamodelos

Una vez contamos con el metamodelo que define la sintaxis abstracta de un dominio de aplicación para modelar sistemas de ese dominio, es preciso definir una sintaxis concreta para él. Dicha sintaxis concreta va a proporcionar la notación que permita a los diseñadores describir los modelos.

Tal y como hemos comentado en la sección 3.2.2, existen diferentes tipos de sintaxis concreta (textual, gráfica) y diferentes propuestas y herramientas para definirlas.

Asimismo, la sintaxis concreta puede definirse desde cero, o bien haciendo uso de una notación existente, extendiéndola con los conceptos de nuestro lenguaje. Esto distingue dos tipos de lenguajes específicos de dominio: internos y externos.

Un **DSL interno** utiliza la notación de un lenguaje existente (el lenguaje "anfitrión") como base para definir sus conceptos y su sintaxis concreta.

Por ejemplo, en la sección siguiente veremos cómo definir lenguajes específicos de dominio extendiendo UML con los nuevos conceptos de nuestro dominio. En este sentido, UML se comporta como lenguaje anfitrión. Esto es muy útil y cómodo porque puede reutilizarse tanto la sintaxis general de UML como sus herramientas de modelado. También es posible definir DSL internos con lenguajes textuales, y por ejemplo Ruby es uno de los más utilizados para estas tareas por las facilidades de extensión que proporciona. Lisp es otro lenguaje que se ha utilizado para definir DSL internos, así como los lenguajes de programación Java y C#. A los DSL internos se les denomina también DSL empotrados (*embedded*).

**Ejemplo**

Un ejemplo de DSL interno construido sobre Ruby es RubyTL, un lenguaje muy potente para la transformación de modelos: http://rubytl.rubyforge.org/

Un **DSL externo** define su sintaxis concreta de forma independiente de cualquier otro lenguaje.

La ventaja de los DSL externos es que no dependen de otros, y permiten ser más pequeños, flexibles y centrados en el dominio. Por otro lado, obligan a tener que desarrollar todas las herramientas para definirlos y manipularlos, como por ejemplo editores, *parsers*, validadores, etc. Sin embargo, cada vez hay en el mercado más herramientas que permiten realizar estas labores tanto para DSL externos gráficos como textuales, tal y como mencionaremos más adelante al final de la sección "Definición de plataformas".

### 3.3.2. Formas de extender UML

El hecho de que UML sea un lenguaje de propósito general proporciona una gran flexibilidad y expresividad a la hora de modelar sistemas. Sin embargo, hay numerosas ocasiones en las que es mejor contar con algún lenguaje de propósito más específico para modelar y representar los conceptos de ciertos dominios particulares. Esto sucede cuando:

- la sintaxis o la semántica de UML no permiten expresar los conceptos específicos del dominio, o
- cuando se desea restringir y especializar los constructores propios de UML, que suelen ser demasiado genéricos y numerosos.

OMG define tres posibilidades a la hora de definir lenguajes específicos de dominio (véase la figura 14):

**Ved también**

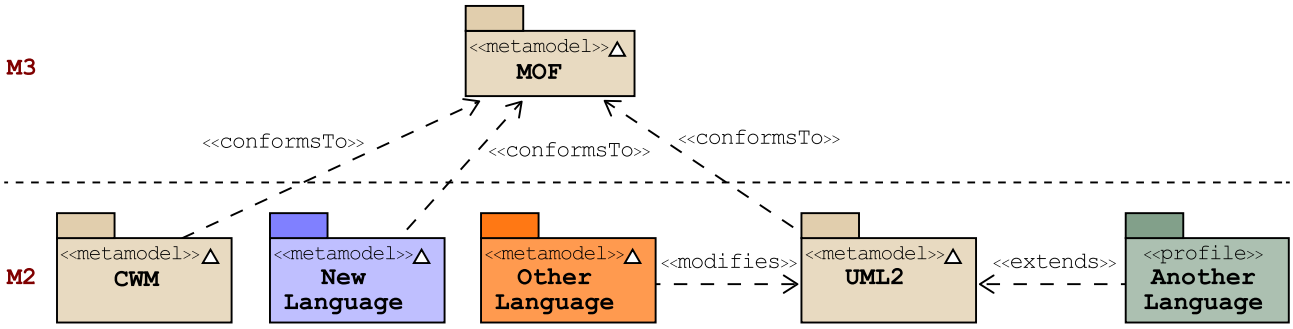
MOF es el lenguaje de metamodelado definido por la OMG para describir lenguajes de modelado. Véanse los apartados 1.2.7 y 3.3.

**a)** Se define un lenguaje completamente nuevo a partir de MOF.

**b)** Se modifica el metamodelo de UML añadiendo nuevas clases, modificando otras y/o borrando algunas. Esto se conoce como una extensión "pesada" de UML (*heavyweight extension*).

c) Se extiende el propio UML, especializando algunos de sus conceptos y restringiendo otros, pero respetando la semántica original de los elementos de UML (clases, asociaciones, atributos, operaciones, transiciones, etc.). Es decir, la estructura original de UML no se modifica, sino que se añaden cosas (clases, relaciones...) a partir de ella. Esta se conoce como una extensión "ligera" de UML (*lightweight extension*).

Figura 14. Formas propuestas por la OMG para definir lenguajes




En la figura 14, M2 y M3 corresponden a los niveles homónimos de la "torre de modelos de la OMG", tal y como se describe en las figuras 1 y 2 del apartado 1.

**Web recomendada**

Para más información sobre CWM (*common warehouse metamodel*) consúltese http://www.omg.org/cwm.

La primera opción es la adoptada por lenguajes como CWM, puesto que la semántica de algunos de sus constructores no coincide con la de UML y sus creadores prefirieron no basarse en UML sino definir un lenguaje nuevo. Para ello solo hay que describir el metamodelo del nuevo utilizando MOF. En este caso estaríamos hablando de DSL externos. La principal ventaja de definir un nuevo lenguaje de esta forma es que permite un mayor grado de expresividad y correspondencia con los conceptos del dominio de aplicación particular, al igual que cuando nos hacemos un traje a medida. Sin embargo, el hecho de no respetar el metamodelo de UML va a impedir que puedan utilizarse las herramientas UML existentes en el mercado (editores, validadores, simuladores, etc.) puesto que no pueden manejar los nuevos conceptos.

La segunda opción consiste en tratar de reutilizar el metamodelo de UML adaptándolo a nuestros requisitos particulares, pero pudiendo modificar las reglas y restricciones de integridad de UML, así como su semántica.

> Por ejemplo, podríamos querer poder definir asociaciones entre elementos que no sean clases, sino también entre paquetes o entre operaciones. O que los elementos de nuestros modelos puedan tener varios nombres (sinónimos) y no solo uno.

La principal ventaja de esta alternativa es que podemos reutilizar muchas partes de UML tal y como están definidas, lo que supone no tener que redefinir algunos subconjuntos del metamodelo de UML que son de mucha utilidad (por ejemplo, máquinas de estados o diagramas de componentes). Ahora bien, el resultado obtenido ya no es UML, y por tanto, en este caso también perde-

mos la posibilidad de usar las herramientas existentes para UML. También se pierde la comprensión esperada de los diagramas UML, al haber cambiado su significado.

Finalmente, hay situaciones en las que es suficiente con extender el lenguaje UML añadiendo elementos o especializando los existentes, pero respetando su semántica. El lenguaje UML permite este mecanismo de extensión utilizando lo que se denominan perfiles UML (*UML profiles*).

El hecho de que dichas extensiones sean conservativas (es decir, que respeten la semántica de UML) va a permitirnos usar los entornos u herramientas existentes para UML con nuestros modelos, pudiendo crearlos, validarlos o intercambiarlos con otras herramientas UML sin problemas.

En el caso de la segunda y tercera opciones, estaríamos hablando de DSL internos, cuyo lenguaje anfitrión es UML.

En general, resulta difícil decidir cuándo es mejor crear un nuevo metamodelo, modificar el de UML o definir una extensión de UML usando los mecanismos de extensión estándares definidos con ese propósito. Quizá la posibilidad de poder usar las herramientas UML existentes sea uno de los principales puntos a considerar, porque la generación de editores y validadores para modelos no es una tarea fácil. Sin embargo, la aparición de este tipo de herramientas[6] han favorecido notablemente la primera opción. Estas herramientas permiten definir metamodelos y construir editores visuales de una forma elegante y asequible. Incluso muchos de ellos cuentan con otras herramientas MDA para la generación de código en diferentes plataformas a partir de alto nivel (como por ejemplo la herramienta Acceleo, de Obeo). Por otro lado, GMF (*grapical modelling framework*) es un proyecto de Eclipse para la generación de herramientas visuales para DSL. Aunque muy potente y expresiva, requiere un conocimiento y grado de experiencia elevado y no resulta fácil crear editores de DSL con GMF.

(6)Como por ejemplo DSL Tools (de Microsoft), MetaEdit+ (de Metacase) u Obeo Designer (de Obeo).

Por las razones expuestas y por ser la forma más utilizada en la actualidad, nos vamos a centrar a continuación en analizar los mecanismos de extensión que proporciona UML para ser extendido mediante perfiles UML. Asimismo, también discutiremos la utilidad y relevancia de estos perfiles UML en el contexto de MDA, en donde juegan un papel destacado.

### 3.3.3. Los perfiles UML

Los perfiles UML aparecieron en la versión 1 de UML, puesto que ya desde el principio se pensaba en UML como una familia extensible de lenguajes, más que como una notación única. En la versión 2 de UML los perfiles fueron modificados ligeramente para mejorar algunos aspectos.

Por ejemplo la forma de acceder desde las metaclases a los estereotipos y viceversa, o los mecanismos para extender y adaptar las metaclases de un metamodelo cualquiera (y no solo el de UML) a las necesidades concretas de una plataforma, como puede ser .NET o J2EE, o de un dominio de aplicación (tiempo real, modelado de procesos de negocio, etc.).

**Web recomendada**

La relación completa de los perfiles UML definidos por la OMG está disponible en http://www.omg.org/technology/documents/profile_catalog.htm

UML 2.0 señala varias razones por las que un diseñador puede querer extender y adaptar un metamodelo existente, sea el del propio UML, el de CWM, o incluso el de otro perfil:

- Disponer de una terminología y vocabulario propio de un dominio de aplicación o de una plataforma de implementación concreta[7].
- Definir una sintaxis para construcciones que no cuentan con una notación propia (como sucede con las acciones).
- Definir una nueva notación para símbolos ya existentes, más acorde con el dominio de la aplicación objetivo[8].
- Añadir cierta semántica que no aparece determinada de forma precisa en el metamodelo[9].
- Añadir restricciones a las existentes en el metamodelo, restringiendo su forma de utilización[10].
- Añadir información que puede ser útil a la hora de transformar el modelo a otros modelos o a código.

(7)Por ejemplo, poder manejar dentro del modelo del sistema terminología propia de EJB como "Home interface", "entity bean", "JAR", etc.

(8)Poder usar, por ejemplo, una figura con un ordenador para representar computadores en una red en lugar del símbolo para representar un nodo que ofrece UML.

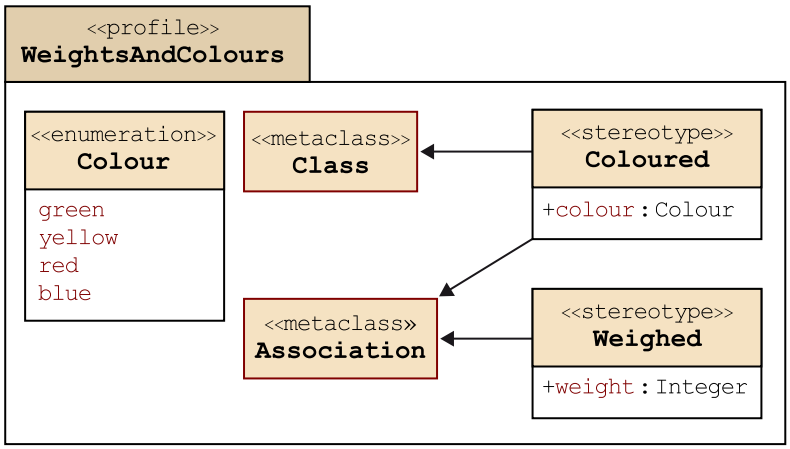
(9)Por ejemplo, la incorporación de prioridades en la recepción de señales en una máquina de estados de UML, relojes, tiempo continuo, etc.

(10)Por ejemplo, impidiendo que ciertas acciones se ejecuten en paralelo dentro de una transición, o forzando la existencia de ciertas asociaciones entre las clases de un modelo.

Un perfil se define en un paquete UML, estereotipado, *profile*, que extiende a un metamodelo existente o a otro perfil. Tres son los mecanismos que se utilizan para definir perfiles: estereotipos (*stereotypes*); definiciones de etiquetas (*tag definitions*) y valores etiquetados (*tagged values*), y restricciones (*constraints*).

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

Figura 15. Un perfil UML para extender UML con colores y pesos




En primer lugar, un estereotipo viene definido por un nombre y por una serie de elementos del metamodelo a los que puede asociarse. Gráficamente, los estereotipos se definen como otras clases UML, pero estereotipadas (etiqueta «`stereotype`»). En nuestro ejemplo, el perfil UML `WeightsAndColours` define dos estereotipos, `Coloured` y `Weighed`, que proporcionan color y peso a un elemento UML. Todos los estereotipos van asociados a una o más metaclases del metamodelo de UML, que son las que el estereotipo extiende.

En el ejemplo de la figura 15 (solo) las clases y las asociaciones de UML pueden colorearse y (solamente) las asociaciones pueden tener asociado un peso. Obsérvese cómo el perfil especifica los elementos del metamodelo de UML (clases estereotipados, *metaclass*) sobre los que se pueden asociar los estereotipos que define el perfil mediante flechas continuas de punta triangular negra.

A los estereotipos es posible asociarles restricciones (*constraints*), que imponen condiciones sobre los elementos del metamodelo que han sido estereotipados. De esta forma pueden describirse, entre otras, las condiciones que ha de verificar un modelo "bien formado" de un sistema en un dominio de aplicación. Por ejemplo, supongamos que el metamodelo de nuestro dominio de aplicación impone la restricción de que si dos o más clases están unidas por una asociación coloreada, el color de las clases debe coincidir con el de la asociación. Dicha restricción se traduce en la siguiente restricción del perfil UML:

```
context Coloured inv sameColour:
   self.base_Association.memberEnd->
      forAll(c | c.extension_Coloured->notEmpty()) implies
         c.extension_Coloured.colour=self.colour
```

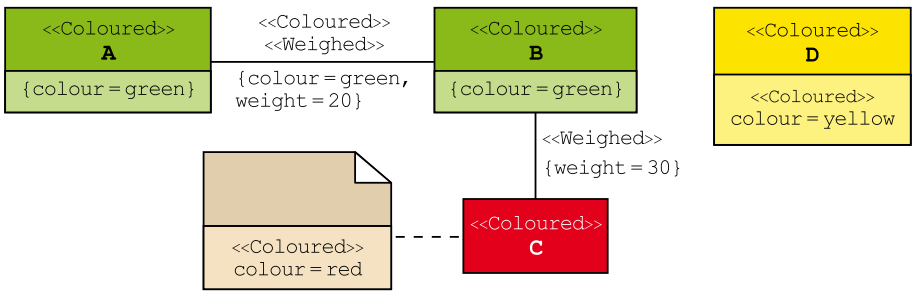
Obsérvese cómo es posible referirse desde un estereotipo a la metaclase que extiende mediante `base_<M>`, donde `<M>` es el nombre de la metaclase (por ejemplo, `base_Class` o `base_Association`). Igualmente, desde una metaclase es posible referirse a un estereotipo de nombre `<X>` como `extension_<X>`, y por tanto preguntar si ese estereotipo está asociado a la clase (la multiplicidad es [0-1] y por tanto basta con preguntar si `notEmpty()`).

En el ejemplo vemos cómo se usa el atributo `memberEnd` de la metaclase `Association` de UML para referirse a las clases relacionadas mediante esa asociación. Para cada una de ellas se pregunta si está estereotipada como `Coloured (c.extension_Coloured->notEmpty())` y, en ese caso, se comprueba si el color es el mismo que el de la asociación `(c.extension_Coloured.colour=self.colour)`.

Finalmente, una definición de etiqueta (*tag definition*) define un meta-atributo adicional que se asocia a una metaclase del metamodelo extendido por un perfil. Toda definición de etiqueta ha de contar con un nombre y un tipo, y se define dentro de un determinado estereotipo. De esta forma, el estereotipo «`Weighed`» de nuestro ejemplo cuenta con un atributo denominado "`weight`", de tipo `Integer`, que indica el peso de cada asociación que haya sido estereotipada como «`Weighed`». Las definiciones de etiquetas se representan de forma gráfica como atributos de la clase que define el estereotipo. Cuando un estereotipo se aplica a un elemento del modelo, los valores de las definiciones de etiquetas se definen mediante valores etiquetados (*tagged values*) que no son sino pares (etiqueta=valor). Por ejemplo, `weight=30`.

La figura 16 muestra un ejemplo del uso de este perfil sobre un modelo UML, donde algunas clases han sido estereotipadas como coloreadas, y a las asociaciones se les ha añadido información sobre su peso y color.

Figura 16. Un ejemplo de uso del perfil UML con pesos y colores




La figura muestra cómo es posible añadir más de un estereotipo a un elemento, así como mostrar los *tagged values* de diferentes maneras: como restricciones (caso de la clase marcada con color verde), en una subcaja (como se muestra en la clase marcada como amarilla) o en una nota aparte (como en el caso de la clase marcada con el color rojo).

Es importante señalar que estos tres mecanismos de extensión no permiten cambiar o eliminar elementos del metamodelo que extienden, solo añadirles elementos y restricciones, pero respetando su sintaxis y semántica original. Esto es lo que permite que las herramientas UML existentes puedan usarse sin problemas con los metamodelos extendidos por un perfil.

Actualmente hay definidos varios perfiles UML, algunos de los cuales han sido incluso estandarizados por la OMG: los perfiles UML para CORBA y para CCM (CORBA *component model*), el perfil UML para EAI (*enterprise application integration*), el perfil UML para MARTE (para tiempo real y sistemas empotrados), el de ingeniería de servicios (SysML) o el de UPDM para la especificación

de marcos arquitectónicos de aplicaciones de defensa. Otros muchos perfiles UML se encuentran actualmente en proceso de definición y normalización por parte de la OMG.

**Ejemplos**

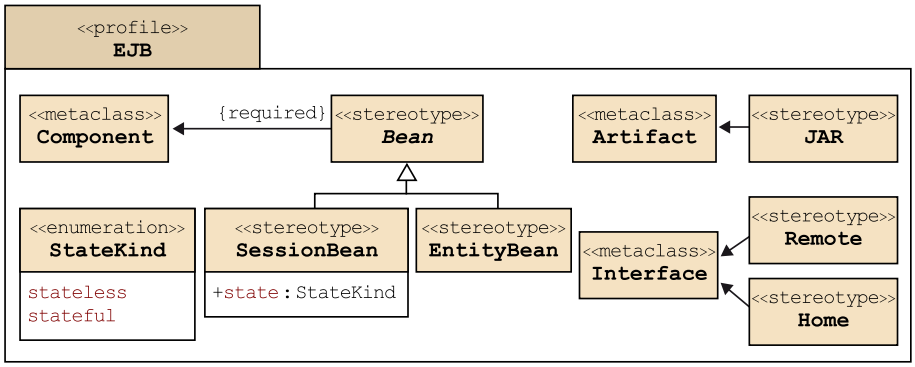
Además de los perfiles UML que ha definido y estandarizado OMG (véase http://www.omg.org/technology/documents/profile_catalog.htm) también hay perfiles UML definidos por otras organizaciones y empresas que, aun no siendo estándares oficiales, están disponibles de forma pública y son comúnmente utilizados (convirtiéndose por tanto en estándares *de facto*). Un ejemplo de tales perfiles es el definido por ISO e ITU-T para la especificación de sistemas de información que siguen el estándar RM-ODP. También se dispone de perfiles UML para determinados lenguajes de programación, como pueden ser Java o C#. Cada uno de estos perfiles define una forma concreta de usar UML en un entorno particular. Así, por ejemplo, el perfil UML para CORBA define una forma de usar UML para modelar interfaces y artefactos de CORBA, mientras que el perfil UML para Java define una forma concreta de modelar código Java usando UML.

**Ved también**

Consultar el estándar *reference model of open distributed processing* (RM-ODP) en la asignatura *Ingeniería del software de componentes y sistemas distribuidos* del grado de Ingeniería Informática.

A modo de ejemplo, la figura 17 muestra un simple perfil para EJB con algunos de los conceptos de este modelo de componentes (*session bean, entity bean*, JAR, *remote interface*, *home interface*) y las metaclases de UML que extienden estos estereotipos.

Figura 17. Un perfil UML muy simple para EJB




### 3.3.4. Cómo se definen perfiles UML

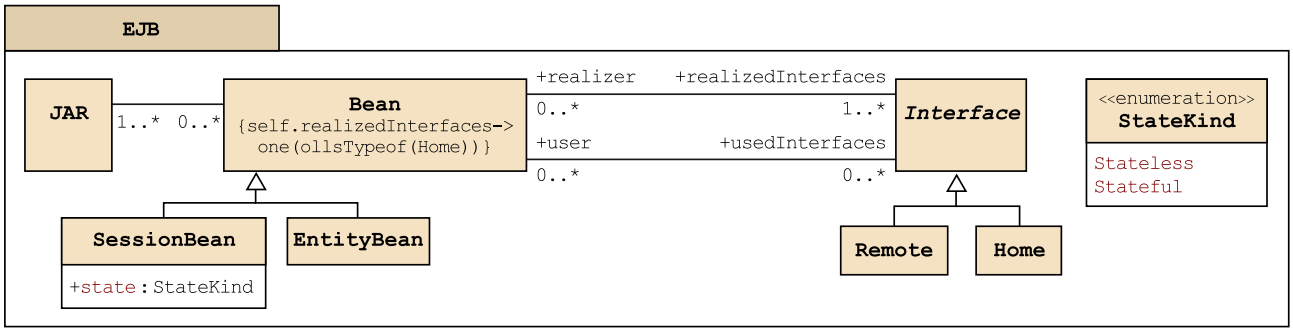
En esta sección describiremos un posible método de elaboración de perfiles UML. La especificación de UML 2.0 solo se limita a definir el concepto de perfil y sus contenidos, sin describir cómo se elaboran. Sin embargo, también es necesario poder contar con ciertas guías y buenas prácticas que sirvan de ayuda a la hora de definir un perfil UML para una plataforma o dominio de aplicación concreto. Por ello, a la hora de definir un perfil UML pueden seguirse los siguientes pasos:

**1)** Antes de comenzar, es preciso disponer de la correspondiente definición del metamodelo de la plataforma o dominio de aplicación a modelar con un perfil. Si no existiese, entonces definiríamos dicho metamodelo utilizando los mecanismos de MOF o del propio UML (clases, relaciones de herencia, asocia-

ciones, etc.) de la forma usual. Deberemos incluir la definición de las entidades propias del dominio, las relaciones entre ellas, así como las restricciones que limitan el uso de estas entidades y de sus relaciones.

Por ejemplo, un posible metamodelo para modelar sistemas de componentes con EJB es el mostrado en la figura 18, que incluye una restricción OCL que expresa que un `Bean` debe implementar exactamente una interfaz `Home`. Este metamodelo es el que dará lugar al perfil mostrado en la figura 17.

Figura 18. Un metamodelo para sistemas EJB




**2)** Una vez disponemos del metamodelo del dominio, pasaremos a definir el perfil. Dentro del paquete *profile* incluiremos un estereotipo para cada uno de los elementos del metamodelo que deseamos incluir en el perfil. Estos estereotipos tendrán el mismo nombre que los elementos del metamodelo, estableciéndose de esta forma una relación entre el metamodelo y el perfil. En principio cualquier elemento que hubiésemos necesitado para definir el metamodelo puede ser etiquetado posteriormente con un estereotipo.

**3)** Es importante tener claro cuáles son los elementos del metamodelo de UML sobre los que vamos a aplicar los distintos estereotipos[11]. Cada estereotipo se aplicará a la metaclase de UML que mejor capture la semántica del correspondiente elemento del metamodelo del dominio.

[11]Ejemplo de tales elementos son las clases, asociaciones, atributos, operaciones, interfaces, transiciones, paquetes, etc.

**4)** Proporcionar definiciones de etiquetas de los elementos del perfil correspondientes a los atributos que aparezcan en el metamodelo. Incluir la definición de sus tipos y sus posibles valores iniciales.

Esto es lo que sucede con el atributo `state` de la clase `SessionBean` del metamodelo, que se convierte en una definición de etiqueta del mismo nombre de la clase `SessionBean` del perfil.

**Nota**

En el caso del perfil para EJB que se mostró en la figura 18, el estereoptipo `Bean` extiende de la metaclase `Component`, `JAR` la metaclase `Artifact` e `Interface` se reutiliza de UML. También se reutilizan las relaciones que define el propio UML entre `Component`, `Artifact` e `Interface`.

**5)** Definir las restricciones que forman parte del perfil, a partir de las restricciones del dominio. Por ejemplo, las multiplicidades de las asociaciones que aparecen en el metamodelo del dominio o las propias reglas de negocio de la aplicación deben traducirse en la definición de las correspondientes restricciones.

Para establecer estas restricciones suele ser preciso navegar por el metamodelo de UML, y por ello hay que conocerlo bien. Por ejemplo, la restricción mencionada anteriormente de que un `Bean` debe implementar exactamente una interfaz `Home` se traduce en el perfil UML en la siguiente restricción `OCL` sobre los componentes estereotipados como `Bean`:

```
context Bean inv oneRealizedHomeInterface:
   self.base_Component.realizedInterfaces()->
      one(extension_Home->notEmpty())
```

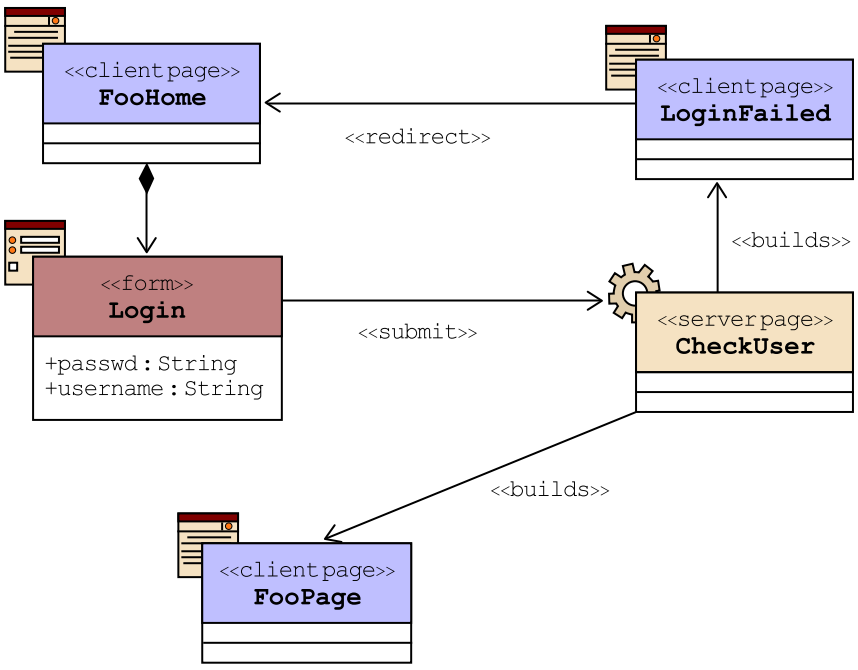
En esa expresión se hace uso de una operación OCL definida en el metamodelo de UML (UML 2.3, §8.3.1, pág. 152) que calcula el conjunto de interfaces realizadas por un clasificador:

```
context Classifier def: realizedInterfaces () : Set(Interface) =
   self.clientDependency->
      select(dependency | dependency.oclIsKindOf(Realization) and
            dependency.supplier.oclIsKindOf(Interface)))->
               collect(dependency|dependency.client)
```

Finalmente, mencionar que es posible asociar un icono a cada estereotipo, el cual será mostrado en lugar del icono estándar que define UML para la metaclase extendida. Gracias a esto fue posible, por ejemplo, usar iconos con formas de máquinas, martillos y bandejas para representar la figura 9 con la cadena de montaje.

Otro ejemplo interesante de perfil UML es el definido por Jim Conallen para modelar aplicaciones web, y que proporciona conceptos como “client page”, “server page”, “target”, “java script” o “form” como elementos nativos del lenguaje y permite usarlos desde cualquier herramienta UML. La figura 19 muestra un modelo UML que hace uso de ese perfil para modelar una parte del sistema que pide *login* con autenticación antes de permitir acceder a la página “FooPage”.

Figura 19. Ejemplo de uso del perfil UML para modelar aplicaciones web




**Consulta recomendada**

**Jim Conallen** (2003). *Building Web Applications with UML* (2.ª ed.). Addison-Wesley.

## 3.4. MDA y los perfiles UML

Además de servir para definir nuevos DSL a partir del metamodelo de UML, los perfiles UML juegan un papel muy importante en MDA para realizar dos tareas principales.

### 3.4.1. Definición de plataformas

En primer lugar, los perfiles UML proporcionan mecanismos muy adecuados para describir los modelos de las plataformas de implementación, tal y como hemos visto de forma muy sucinta con el perfil para EJB. A partir de estos modelos es posible establecer una correspondencia entre cada uno de los elementos del modelo PIM y los de la plataforma, la cual determina de forma unívoca la transformación del PIM al correspondiente PSM.

La idea es utilizar los estereotipos del perfil de una plataforma para "marcar" los elementos de un modelo PIM de un sistema y producir el correspondiente modelo PSM, ya expresado en términos de los elementos de la plataforma destino. Una marca representa un concepto en el modelo PSM, que se aplica a un elemento del modelo PIM para indicar cómo debe transformarse dicho elemento en el modelo PSM destino. Las marcas pueden especificar también requisitos extra-funcionales o de calidad de servicio sobre la implementación final[12].

(12)Por ejemplo, algunos de los elementos del modelo PIM pueden marcarse como persistentes, o sujetos a determinadas condiciones de seguridad.

El conjunto de las marcas y las reglas de transformación que gobiernan el uso de las mismas deben ser especificadas de forma estructurada y normalmente se proporcionan junto con el perfil UML de la plataforma destino. Si el perfil UML de la plataforma incluye la especificación de operaciones, entonces las reglas de transformación deberían incluir también el tratamiento de dichas operaciones.

### 3.4.2. Marcado de modelos

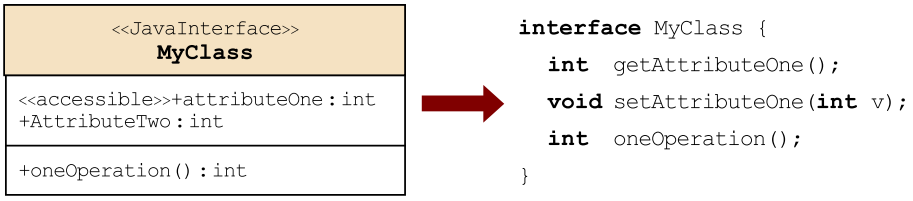
El segundo uso de los perfiles es para "decorar" (o "marcar" según la terminología MDA) un modelo con información adicional sobre ciertos de sus elementos, para indicar decisiones de diseño o implementación sobre ellos.

En MDA, una **marca** representa un concepto del PSM que puede ser aplicado a un elemento del PIM para indicar cómo hay que transformar ese elemento.

Para ilustrar este concepto, la figura 20 muestra una clase del PIM denominada `MyClass` con dos atributos y una operación, que ha sido marcada con dos estereotipos, «`JavaInterface`» y «`accessible`», para transformarla adecuadamente a un modelo PSM. El primer estereotipo sirve para indicar las clases

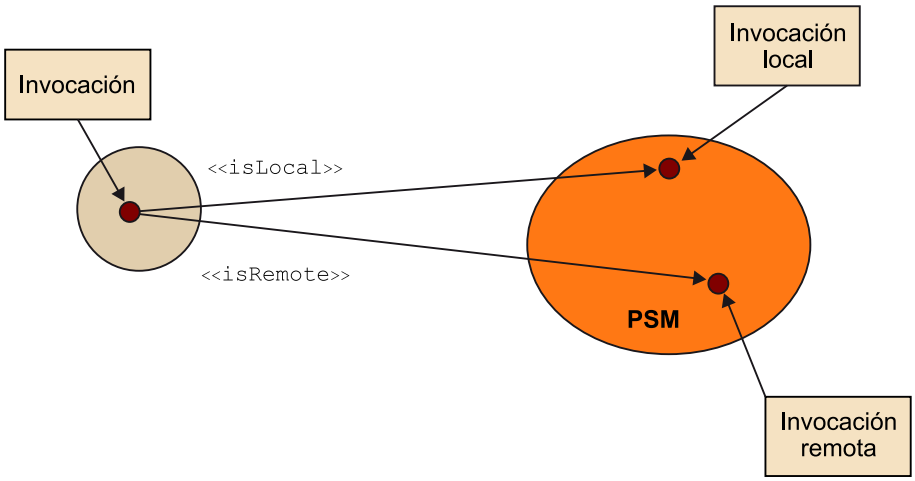
del PIM que han de transformarse en interfaces Java, y el segundo permite seleccionar los atributos de esas clases que han de ser visibles en el PSM. Como puede observarse en la figura, el segundo atributo no aparece en el PSM transformado.

Figura 20. Un ejemplo de PIM marcado y su transformación a Java




De forma análoga, la figura 21 muestra gráficamente el uso de estereotipos para indicar en el PIM si una operación ha de transformarse en otra cuya invocación ha de realizarse de forma local o remota en el PSM.

Figura 21. Otro ejemplo de marcas en el PIM para decidir cómo han de transformarse elementos concretos en el correspondiente PSM



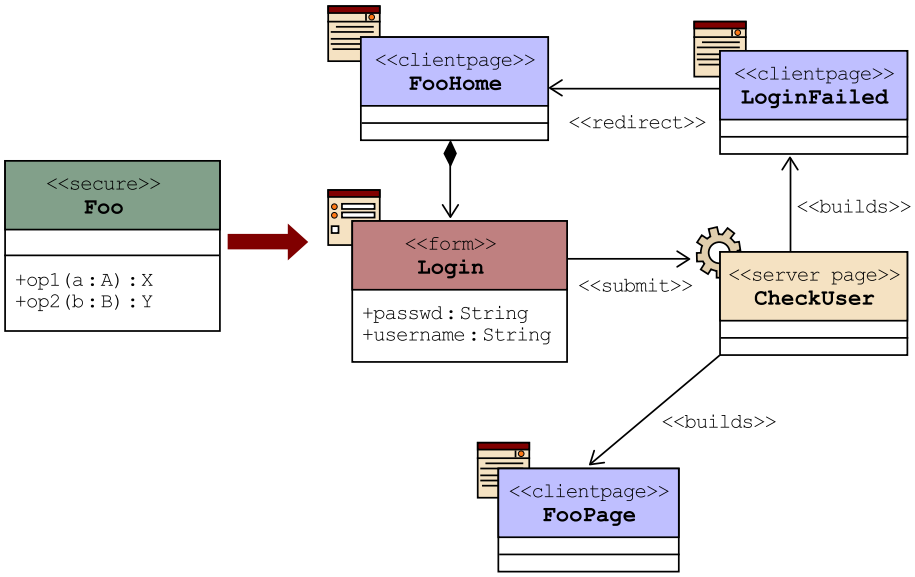

### 3.4.3. Plantillas

Además de marcas, MDA también define el concepto de plantilla (*template*):

> En MDA, una **plantilla** (*template*) es un modelo parametrizado que especifica la forma y contenido de ciertas transformaciones.

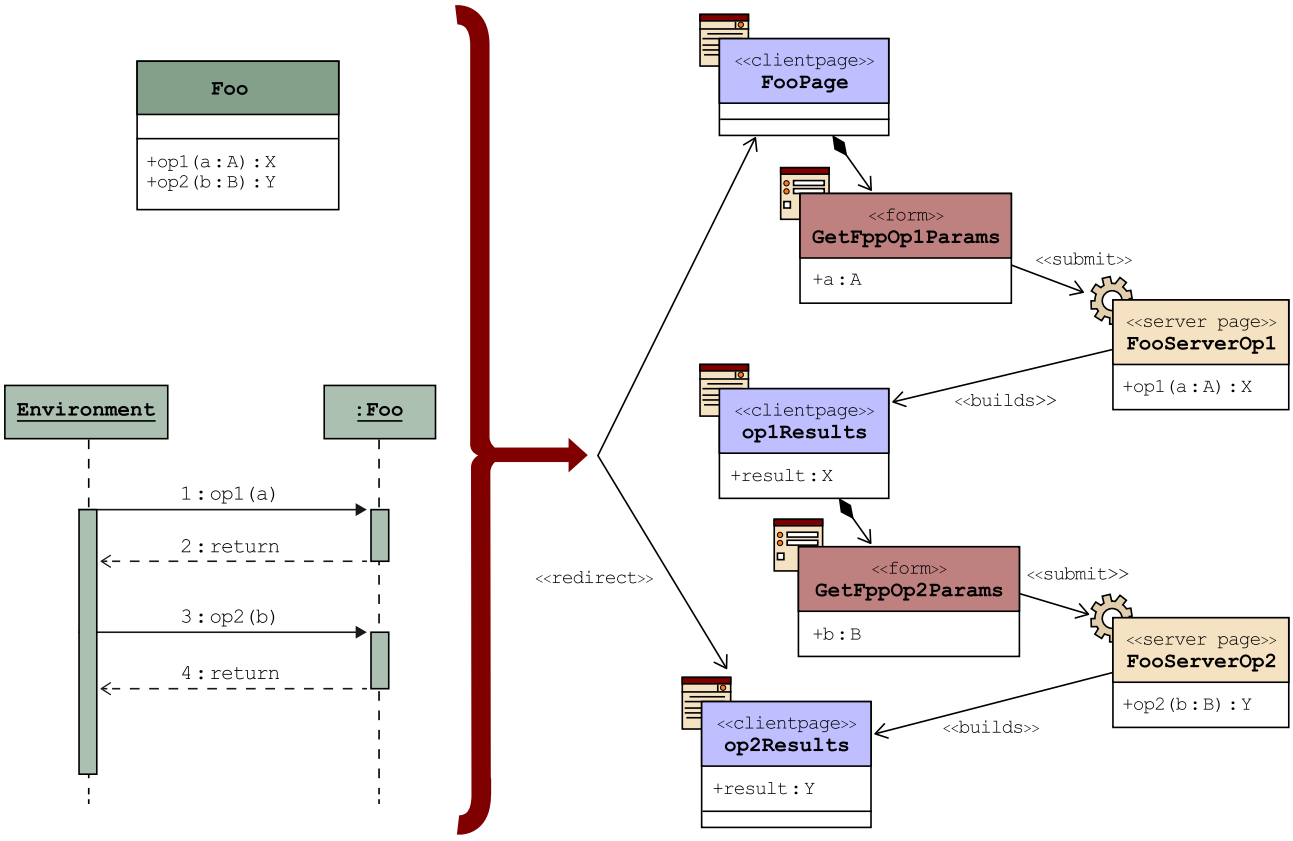
Por ejemplo, la figura 22 muestra el comportamiento de una transformación de modelos que permite transformar clases marcadas como «secure» (`Foo`, en el ejemplo) en un conjunto de clases que representan a la correspondiente página web (`FooPage`) junto a las páginas necesarias para autenticar a los usuarios antes de acceder a ella.

Figura 22. Un ejemplo de plantillas para transformar modelos




La figura 23 muestra otro ejemplo del uso de plantillas para definir la transformación de determinados elementos de los modelos marcados con estereotipos en conjuntos de elementos del modelo destino. En este caso, el PIM proporciona información sobre una clase y sus operaciones, así como la secuencia válida en la que deben invocarse dichas operaciones. A partir de esa información una transformación de modelos es la encargada de generar el modelo de la aplicación Web que invoca dichas operaciones desde la página inicial correspondiente.

Figura 23. Otro uso de plantillas para transformar modelos en MDA



## 3.5. Consideraciones sobre la definición de DSL

Tal y como mencionamos anteriormente, existen numerosas alternativas para definir DSL, o bien usar lenguajes existentes para expresar los modelos de un sistema. En esta sección analizamos las principales opciones, y discutimos las ventajas e inconvenientes de cada una.

### 3.5.1. Uso de DSL frente a lenguajes de uso general ya conocidos

La primera decisión a considerar es si merece la pena definir un nuevo DSL o usar algún lenguaje de modelado o programación existente.

Entre las ventajas de definir un nuevo DSL destacamos las siguientes:

- Los DSL permiten expresar el diseño del sistema usando una notación muy cercana y natural para los expertos del dominio, cuyos conceptos son los que normalmente manejan. Esto les permite entender, escribir y validar sus propios modelos.

- Los DSL permiten incrementar la usabilidad, mantenibilidad y reutilización de los diseños y modelos por parte de los expertos del dominio, de nuevo porque usan una notación conocida por ellos.

- Los DSL permiten expresar sistemas de forma independiente de la tecnología usada o de los lenguajes de implementación, y al nivel de abstracción adecuado.

- La validación de los modelos puede realizarse desde el propio dominio del problema, ya que un DSL puede controlar que se construyan modelos válidos.

- Los DSL suelen ser mucho más "compactos" y menos complejos que los lenguajes de modelado de propósito general, con muchas menos clases y relaciones entre ellas (y que, en la mayoría de los casos, no suelen utilizarse).

**Nota**

De hecho, la "regla del 80-20" dice que para el 80% de los diseños UML no es preciso más del 20% del lenguaje. Piénsese que el metamodelo de UML 2 tiene más de 800 metaclases.

- El nivel de expresividad y precisión que se consigue con un DSL suele ser mucho más elevado que el que se logra con un lenguaje de propósito general.

- En muchos casos es más sencillo desarrollar herramientas (o integrar algunas de las existentes) para manejar o analizar modelos desarrollados con DSL que con lenguajes de propósito general, precisamente por su genericidad y amplio espectro.

Algunas desventajas de definir DSL de manera *ad-hoc*:

- El coste de tener que aprender un lenguaje distinto para cada dominio de aplicación no es despreciable, especialmente si el dominio de aplicación es bastante restringido.

- El diseño, implementación y mantenimiento de un DSL no son tareas fáciles. Y más si también tenemos que diseñar, implementar y mantener muchas de las herramientas asociadas a cada nuevo lenguaje (editores, validadores, etc.).

- Los mecanismos y herramientas existentes para reutilizar (subconjuntos de) lenguajes y combinarlos entre sí son todavía muy rudimentarios, lo que obliga a comenzar casi desde cero cada vez que se desea definir y diseñar un nuevo lenguaje.

- Aunque se gane notablemente en expresividad, puede que se pierda en otros aspectos, como eficiencia o reutilización, por ejemplo.

- La definición descontrolada de nuevos DSL puede producir incompatibilidades entre empresas que definan DSL diferentes para modelar el mismo tipo de sistemas, o incluso que una misma empresa defina dos DSL diferentes (e incompatibles entre sí) para modelar lo mismo.

- La experiencia muestra que no todos los expertos de un dominio usan de igual forma los conceptos del dominio, e incluso pueden usarlos de forma diferente. Esto implica posibles problemas de interoperabilidad y consistencia entre modelos (y DSL) desarrollados por distintos expertos.

- No todos los expertos de un dominio están dispuestos (o preparados) para escribir y modificar modelos usando estos DSL.

- Puede existir una mayor dificultad para integrar distintos DSL con los sistemas y aplicaciones de una compañía que si se usaran lenguajes de propósito general.

- La documentación y ejemplos existentes sobre el diseño, desarrollo y uso de DSL es bastante más escasa que la existente sobre lenguajes de propósito general.

### 3.5.2. Uso de perfiles UML

Una alternativa interesante es definir un DSL mediante perfiles UML, lo que trata de conjugar las ventajas del uso de DSL con las que proporcionan los lenguajes de propósito general como UML. Sin embargo, el uso de una exten-

sión ligera como los perfiles UML frente al uso de definición de lenguajes a partir de MOF (o Ecore, en el caso de Eclipse) tampoco es una decisión fácil de tomar, pues ambas opciones presentan ventajas e inconvenientes.

Los perfiles UML cuentan con algunas ventajas muy atractivas:

- Son fáciles de definir y manejar si se está acostumbrado a manejar modelos y herramientas UML.

- Hay un gran número de personas formadas ya en UML. En este caso, puede reutilizarse el conocimiento y experiencia existente en la empresa sobre UML y sobre sus herramientas.

- No hay que olvidar que UML es un estándar y esto facilita enormemente la interoperabilidad con los sistemas de información y herramientas de otras empresas.

- No es preciso aprender otros lenguajes de modelado, ni adquirir otras herramientas.

- Son muy útiles a la hora de construir prototipos de lenguajes de una forma fácil, rápida y sin demasiado esfuerzo.

Por otro lado, los perfiles UML también presentan algunos inconvenientes:

- Su expresividad es limitada, al tener que respetar la semántica de UML.

- En general son difíciles de definir de una forma rigurosa, sobre todo por la complejidad que representa especificar sus restricciones de integridad en base a los elementos del metamodelo de UML.

- Los perfiles no permiten "ocultar" nada de UML, por lo que siempre se arrastra todo su metamodelo. Esto tiene dos problemas principales: por un lado la complejidad del metamodelo resultante, y por otro que no se puede evitar que los usuarios del perfil usen el resto de los elementos de UML en sus modelos, algo que en numerosas ocasiones no es deseable.

- Manipular modelos desarrollados con perfiles UML no es simple (por ejemplo, a la hora de transformarlos), por la forma en la que se representan internamente las extensiones a UML.

- Los modelos desarrollados con perfiles UML heredan también algunos de los problemas del propio UML, como su falta de precisión.

- Por último, la notación que se consigue finalmente con un perfil UML no es tan compacta y atractiva como la que puede conseguirse con un DSL cuya sintaxis concreta se haya desarrollado a medida[13].

(13)Aunque es verdad que el coste de definir la sintaxis concreta para un perfil UML es significativamente menor que el coste que implica definir la sintaxis concreta de otros DSL.

La siguiente tabla muestra algunos consejos que pueden ayudar a tomar una decisión en uno u otro sentido, dependiendo de las circunstancias particulares del diseñador:

| Es conveniente definir el DSL a partir de MOF (o Ecore) cuando... | Es conveniente usar perfiles UML cuando... |
|---|---|
| El dominio de aplicación está bien definido y consolidado, y los conceptos están ampliamente aceptados. | El dominio no es muy estándar o estable, y por tanto, es preciso realizar prototipos rápidos del lenguaje y de sus herramientas asociadas. |
| No se necesita poder combinar fácilmente modelos de aplicaciones de diferentes dominios. | Es preciso combinar fácilmente modelos de diferentes dominios de aplicación. |
| La semántica del dominio no es compatible con la de UML. | Es posible respetar la semántica (y *look-and-feel*) de UML para modelar sistemas de ese dominio. |
| La empresa no cuenta con demasiada experiencia en UML ni con sus herramientas asociadas. | La empresa cuenta con herramientas UML que normalmente utiliza en sus proyectos, y experiencia con UML. |
| La sintaxis concreta que se necesita para expresar los modelos es muy diferente a la de UML o no puede conseguirse con los perfiles UML. | La notación que se precisa para el lenguaje no es muy diferente de la de UML, y puede obtenerse a partir de ella mediante pequeños cambios. |
| Las herramientas que se necesitan para editar, manipular y analizar los modelos no admiten una integración fácil con UML/XMI. | Las herramientas asociadas a los modelos se integran bien con las herramientas disponibles en la empresa para manejar modelos UML. |

### 3.5.3. Lenguajes de modelado frente a lenguajes de programación

Una distinción que tradicionalmente se ha hecho es la de los lenguajes de modelado frente a los de programación. Normalmente se ha considerado que los lenguajes de modelado se usan de manera informal, no tienen una semántica precisa y no son ejecutables. Por el contrario, los lenguajes de programación siempre se han visto con una semántica muy precisa (la que proporciona el compilador) y ejecutables, aunque de muy bajo nivel.

Sin embargo, con la llegada de MDE esta distinción se ha ido perdiendo y los dos tipos de lenguajes pueden verse como similares: ambos cuentan con los mismos componentes (sintaxis abstracta, sintaxis concreta y semántica), y la ejecutabilidad es algo que ya no es exclusivo de los lenguajes de programación, tal y como hemos mencionado anteriormente.

**Ejemplo**

Por ejemplo, en la página http://wiki.eclipse.org/MoDisco/Components/Java/Documentation se describe en detalle el metamodelo del lenguaje Java, es decir, su sintaxis abstracta.

Quizá la mayor diferencia se encuentre en la actualidad en el nivel de detalle que permiten expresar estos lenguajes, siendo por lo general de más alto nivel los lenguajes de modelado que los de programación. De hecho, los lenguajes de modelado suelen utilizarse sobre todo en labores de diseño, mientras que los lenguajes de programación suelen usarse para labores de implementación. Esta es probablemente la principal razón por la que los lenguajes de modelado suelen contar con notaciones gráficas o visuales, frente a las notaciones textuales de los lenguajes de programación. No hay que olvidar la facilidad con la que se representan en los lenguajes visuales las relaciones entre los elementos de un modelo, algo que resulta más complicado en los lenguajes de programación. Por el contrario, y como también hemos mencionado anteriormente, describir procesos e instrucciones de procesamiento secuenciales suele ser más sencillo y apropiado usando lenguajes de programación. De todas formas la situación va cambiando y ya vemos cómo los lenguajes de modelado como UML se van complementando con notaciones ejecutables, a la vez que lenguajes de programación como Java comienzan a contar con asertos y elementos de mayor nivel de abstracción para describir sistemas.

Además del nivel de abstracción, también es importante destacar una limitación actual de los lenguajes de programación: la **unificación**. Esta es la capacidad de conseguir que varios lenguajes de programación coexistan y se combinen de forma sencilla y coherente para formar la especificación integrada de un mismo sistema. En este sentido, la **arquitectura de metamodelado** que hemos presentado en este apartado permite un mecanismo para ayudar a solucionar este problema, al compartir los lenguajes de modelado un metamodelo común. El metamodelado, junto con el uso de transformaciones de modelos para convertir unos modelos en otros (tanto a nivel horizontal como vertical), son las claves en las que se basa MDE, y en particular MDA, para el modelado, análisis y desarrollo de sistemas de información de una forma sistemática, predecible, medible y rigurosa. Las transformaciones de modelos las estudiaremos en el siguiente apartado de este módulo.

# 4. Transformaciones de modelos

## 4.1. La necesidad de las transformaciones de modelos

Como se describió en el apartado 1, los modelos son piezas claves en el ámbito de MDE (*model-driven engineering*). Por ello, es necesario contar con mecanismos de manipulación de modelos. Las transformaciones de modelos proporcionan los mecanismos que permiten especificar el modo de producir modelos de salida a partir de modelos de entrada.

El concepto de transformación en el mundo de la informática no es nuevo. De hecho, las transformaciones son algo fundamental en la ingeniería del software, pudiendo verse la computación como un proceso de transformación de datos.

> Los procesos de transformación de datos se refieren a la transformación de información en general, cubriendo desde cambios en la representación de los datos a cambios en los datos mismos. Por ejemplo, un proceso de transformación de datos puede tomar un fichero con una codificación de caracteres concreta y convertirlo en otro fichero con otra codificación de caracteres diferente, normalmente para lograr la interoperabilidad de aplicaciones o sistemas diferentes. Otro proceso puede tomar los datos de una base de datos de clientes y obtener otra base de datos con la información de las ventas de la compañía, a partir de los datos de la primera.

**Consulta recomendada**

Los sistemas de transformaciones de datos (o programas) están fuera del ámbito de este apartado. No obstante, una buena referencia es: Helmuth Parstsch; Ralf Steinbrüggen (1983). "Program Transformation Systems". *ACM Comput. Surv.* (vol. 3, núm. 15, págs. 199-236).

Entre las aplicaciones de las transformaciones de modelo en el ámbito de MDE nos encontramos:

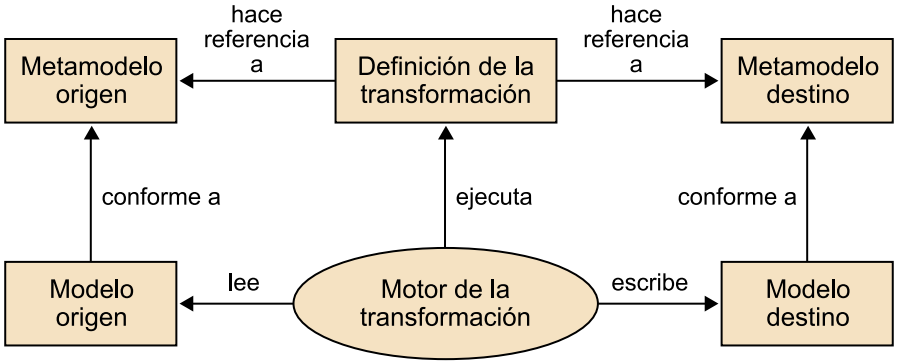
- Generación de modelos a más bajo nivel, y finalmente código, a partir de modelos a alto nivel.
- Sincronización y *mapping* entre modelos al mismo o diferentes niveles de abstracción.
- Creación de vistas basadas en *queries* sobre un sistema.
- Aplicación a la ingeniería inversa para obtener modelos a más alto nivel a partir de modelos a más bajo nivel.

## 4.2. Conceptos básicos

Las **transformaciones de modelos** pueden verse como programas que toman un modelo (o más de uno) como entrada y devuelven otro modelo (o más de uno) como salida.

En la figura 24 se pueden observar los participantes involucrados en una transformación de modelos. En ella se muestra un escenario de transformación con un modelo de entrada y uno de salida, los cuales deben ser conformes a sus respectivos metamodelos. Como se vio en los apartados previos, un metamodelo define una sintaxis abstracta, y los modelos que la respetan se dice que son conformes a ella. Cuando se define una transformación entre modelos, se hace respecto a sus metamodelos. Posteriormente se utilizará un motor de transformación para ejecutarla sobre modelos concretos. En general, una transformación puede tener varios modelos origen y destino (*source model* y *target model*, en inglés, respectivamente), siendo posible además que el metamodelo origen y destino sean el mismo. Un ejemplo de transformación donde coinciden el modelo origen y destino la encontramos, por ejemplo, cuando se proporcionan transformaciones que describen casos de estudio de un sistema.

Figura 24. Participantes de una transformación de modelos




Generalmente, una transformación de modelos está formada por un conjunto de **reglas de transformación**, que son consideradas como las unidades de transformación más pequeñas. Cada una de las reglas describe cómo uno o más elementos del modelo origen son transformados en uno o más elementos del modelo destino.

**Nota**

Al hablar de elementos nos referimos a clases, atributos, relaciones, etc.

### 4.3. Tipos de transformaciones

En MDE se describen diferentes tipos de transformaciones entre modelos, dependiendo de varios criterios:

**Consulta recomendada**

En el siguiente trabajo se hace una clasificación muy detallada y completa de las transformaciones de modelos: Krzysztof Czarnecki; Simon Helen. "Feature-Based Survey of Model Transformation Approaches". *IBM System Journal* (vol. 45, núm. 3 (206), págs. 621-645).

**a)** Nivel de abstracción de los modelos de entrada y salida:

- Transformaciones **verticales**. Relacionan modelos del sistema situados en distintos niveles de abstracción y pueden aplicarse tanto en sentido descendente[14] como en sentido ascendente[15], siendo estos últimos el resultado de aplicar un proceso de ingeniería inversa.

- Transformaciones **horizontales**. Relacionan los modelos que describen un sistema desde un nivel de abstracción similar. Se utilizan también para mantener la consistencia entre distintos modelos de un sistema, es decir, garantizan que la información modelada sobre una entidad en un modelo es consistente con lo que se dice sobre dicha entidad en cualquier otra especificación situada al mismo nivel de abstracción.

(14)Transformaciones de PIM a PSM, también denominados refinamientos.

(15)De PSM a PIM, llamados abstracciones.

**b)** Tipo de lenguaje que se utiliza para especificar las reglas. Se distingue entre lenguajes **declarativos**, **imperativos** o **híbridos** (si permiten ambos tipos de forma combinada).

**Consulta recomendada**

El siguiente artículo de Perdita Stevens estudia la especificación de transformaciones bidireccionales utilizando QVT: "Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions", *SoSyM* 9 (1): 7-20 (2010).

**c)** Atendiendo a la direccionalidad, nos encontramos con:

- Transformaciones **unidireccionales**, las reglas se ejecutan en una sola dirección.

- Transformaciones **bidireccionales**. En estas, las transformaciones se pueden aplicar en ambas direcciones[16].

(16)Del modelo origen al destino y del destino al origen.

**d)** Dependiendo de los modelos origen y destino:

- Transformaciones **exógenas**. Los metamodelos origen y destino son distintos. Es el caso más común: se tiene un modelo de entrada y se obtiene el de salida a partir de él aplicando las reglas de la transformación.

- Transformaciones **endógenas**. Aquí, los metamodelos origen y destino son el mismo, se trata de las llamadas transformaciones *in-place*: las reglas de transformación se van aplicando sobre el propio modelo de entrada, de modo que este se va transformando hasta que no se puede aplicar ninguna regla más y el modelo de entrada pasa a ser el de salida.

**e)** Atendiendo al tipo de modelo destino:

- Transformaciones **modelo a modelo** (M2M), las cuales generan modelos a partir de otros modelos. Se pueden clasificar a su vez en subgrupos, como se verá en el apartado 4.4.

- Transformaciones **modelo a texto** (M2T), que generan cadenas de texto a partir de modelos. Son usadas, por ejemplo, para generar código y documentos. Se clasifican a su vez en subgrupos, como se explica en el apartado 4.5.

## 4.4. Lenguajes de transformación modelo-a-modelo (M2M)

La OMG identifica, a través de MDA, cuatro tipos de transformaciones de modelo a modelo (M2M) dentro del ciclo de vida del software:

**1) Transformaciones PIM2PIM**. Estas transformaciones se usan en el tratamiento de modelos independientes de la plataforma y se aplican cuando se obtienen, filtran o especializan unos PIM a partir de otros.

**2) Transformaciones PIM2PSM**. Transforman un modelo independiente de la plataforma en su correspondiente modelo en una plataforma concreta.

**3) Transformaciones PSM2PSM**. Se utilizan para refinar un modelo desplegado en una plataforma concreta.

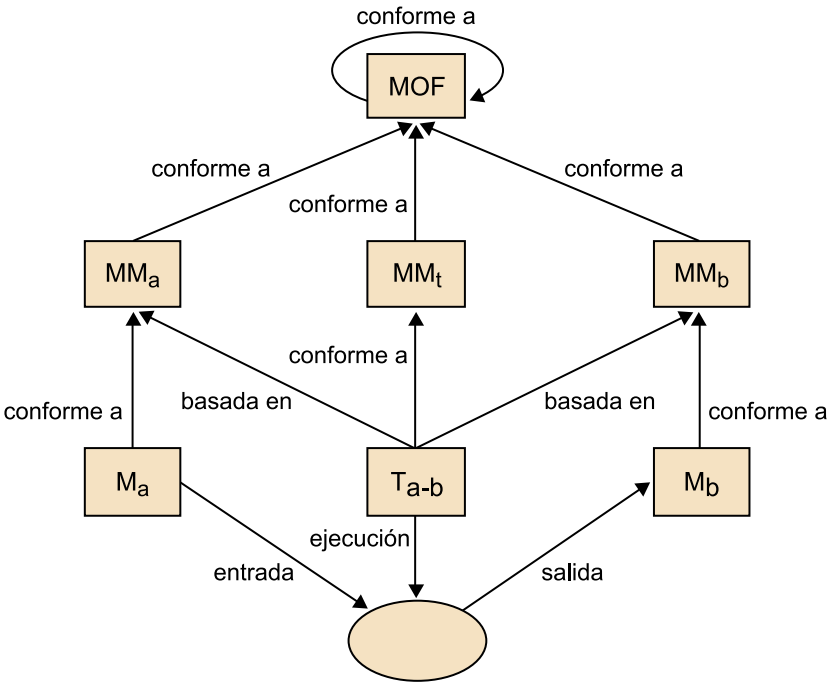
**4) Transformaciones PSM2PIM**. Abstraen modelos desplegados en plataformas concretas en modelos independientes de la plataforma.

**Nota**

Las transformaciones PSM2PIM se utilizan, sobre todo, para labores de ingeniería inversa y de modernización de sistemas.

De todos los lenguajes existentes para describir transformaciones modelo a modelo, los más usados hoy en día son QVT (*query/view/transformation*) y ATL (*Atlas transformation language*). Ambos presentan el mismo contexto operacional, que se muestra en la figura 25.

Figura 25. Contexto operacional de ATL y QVT




La transformación está representada por $T_{a\text{-}b}$, cuya ejecución resulta en la creación de un modelo $M_b$ a partir de un modelo $M_a$. Las tres entidades mencionadas ($T_{a\text{-}b}$, $M_b$ y $M_a$) son modelos conforme a los metamodelos MOF $MM_t$, $MM_b$ y $MM_a$, respectivamente. $MM_a$ y $MM_b$ representan las sintaxis abstractas de los lenguajes origen y destino de la transformación, mientras $MM_t$ representa la sintaxis abstracta del lenguaje de transformación usado, en este caso QVT o ATL.

**Nota**

Recordemos que, como se vio en el apartado "Conceptos básicos", MOF (*meta-object facility*) es el lenguaje de la OMG para describir metamodelos.

En los siguientes apartados se detallan QVT y ATL por separado, haciendo especial hincapié en el segundo, con el que realizaremos algunos ejemplos.

### 4.4.1. QVT: Query-View-Transformation

QVT (del inglés, *query-view-transformation*) es un estándar propuesto por la OMG y formado por un conjunto de lenguajes destinados a las transformaciones de modelos. Es capaz de expresar transformaciones, vistas y peticiones entre modelos dentro del contexto de la arquitectura de metamodelado MOF 2.0.

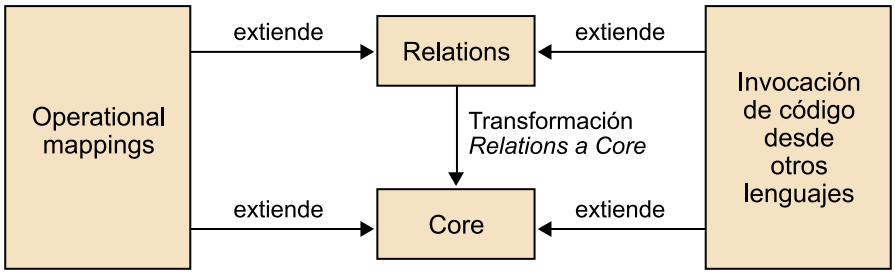
**Web recomendada**

La última especificación de MOF-QVT (enero de 2011) se encuentra en http://www.omg.org/spec/QVT/1.1/.

La sintaxis abstracta de QVT se define como un metamodelo MOF 2.0. Dicho metamodelo define tres sub-lenguajes para transformar modelos, los cuales componen un lenguaje de transformaciones **híbrido** con constructores tanto declarativos como imperativos. Estos lenguajes se llaman *Relations*, *Core* y *Operational Mappings*, y se organizan en una estructura por niveles que se muestra en la figura 26. La definición de estos tres lenguajes incorpora en su sintaxis el lenguaje estándar OCL para expresar consultas (*queries*) sobre los modelos. Los lenguajes *Relations* y *Core* son declarativos y se encuentran en dos niveles de

abstracción diferentes. El tercer lenguaje, *Operational Mappings*, es un lenguaje imperativo que extiende a los otros dos lenguajes. A continuación detallamos cada uno de estos lenguajes.

Figura 26. Arquitectura en niveles de los lenguajes de QVT




**1) *Relations***. Este lenguaje permite especificar transformaciones como un conjunto de relaciones entre modelos. Esto permite al desarrollador crear elementos en el modelo destino a partir de elementos del modelo origen, y también aplicar cambios sobre modelos existentes. Este lenguaje se encarga de la manipulación automática de las trazas (estructuras de datos donde se guardan las relaciones entre los conceptos de los modelos origen y los modelos destino) de modo transparente al desarrollador.

**Consulta recomendada**

El siguiente trabajo compara los lenguajes ATL y QVT y estudia la interoperabilidad entre ambos: Frédéric Jouault; Ivan Kurtev (2006). "On the Architectural Alignment of ATL and QVT ". *SAC* (págs. 1188-1195).

**2) *Core***. Este lenguaje declarativo es más simple que el anterior. Por tanto, las transformaciones escritas en *Core* suelen ser más largas que sus equivalentes especificadas en *Relations*. Las trazas en *Core* son tratadas como elementos del modelo, es decir, su manejo no es transparente en este caso, por lo que el desarrollador se responsabiliza de crear y usar las trazas. Una de las razones de ser de este lenguaje es proporcionar una base sobre la que especificar la semántica del lenguaje *Relations*. Dicha semántica se da como una transformación desde *Relations* a *Core*, la cual es escrita en el lenguaje *Relations*.

A veces es difícil desarrollar una solución puramente declarativa para una transformación determinada. Por ello, QVT propone dos mecanismos para "extender" los lenguajes declarativos *Relations* y *Core* con nuevos conceptos y mecanismos: el lenguaje *Operational Mappings* y un nuevo mecanismo para invocar la funcionalidad de transformaciones implementadas en otros lenguajes (representado en la figura 26 por el rectángulo en el extremo derecho).

**3) *Operational Mappings***. Como se ha mencionado, este lenguaje extiende al lenguaje *Relations* con constructores imperativos y constructores OCL. Así, este lenguaje es puramente imperativo y similar a los lenguajes procedurales de programación tradicional. Además, incorpora elementos constructivos diseñados para operar (crear, modificar y eliminar) sobre el contenido de los modelos. *Operational Mappings* se estructura básicamente en procedimientos u operaciones, denominados *mapping operations* y *helpers* o *queries*.

**4) Invocación de código desde otros lenguajes (*black box* en inglés)**. Este mecanismo permite introducir y ejecutar código externo en la transformación en tiempo de ejecución, lo cual permite implementar algoritmos complejos en cualquier lenguaje de programación y reutilizar librerías existentes. Sin embargo, esto hace que algunas partes de la transformación sean opacas, lo que crea un riesgo potencial, ya que la funcionalidad será arbitraria y no podrá ser controlada por el motor de ejecución.

**Nota**

En este módulo no entraremos en detalle sobre los mecanismos de QVT *Operational Mappings* y *Black Box*, por ser de un nivel más avanzado. El lector interesado puede encontrar ejemplos en el siguiente enlace: http://www.eclipse.org/m2m/qvto/doc/M2M-QVTO.pdf

### 4.4.2. ATL: Atlas Transformation Language

ATL es un lenguaje de transformación de modelos desarrollado sobre EMF (*eclipse modeling framework*) como parte de la plataforma AMMA (*ATLAS model management architecture*). Los desarrolladores de ATL se inspiraron en QVT para su creación, y, como en este, OCL forma parte del lenguaje.

**Web recomendada**

El manual y numerosos ejemplos de ATL se pueden encontrar en http://www.eclipse.org/m2m/atl/doc/

A pesar de que ATL es un lenguaje híbrido, que proporciona tanto constructores declarativos como imperativos, es recomendado usarlo de forma declarativa a no ser que sea estrictamente necesario, es decir, que la transformación no pueda especificarse de otro modo.

**Web recomendada**

La descripción de AMMA se encuentra detallada en http://wiki.eclipse.org/AMMA

Las transformaciones escritas con ATL son unidireccionales, las cuales operan sobre modelos origen de solo lectura y generan modelos destino de solo escritura. Esto quiere decir que las transformaciones no pueden navegar (consultar) el modelo de salida que se va creando, ni tampoco modificar el modelo de entrada (aunque sí navegarlo). Si se quiere implementar una transformación bidireccional con ATL, es necesario definir una transformación en cada sentido.

El código de cualquier transformación ATL se organiza en tres partes bien definidas: la **cabecera**, un conjunto de ***helpers*** (opcional) y un conjunto de **reglas**.

**Ved también**

En el apartado 4.4.4. se muestra un caso de estudio donde se ejemplifica el uso de las reglas ATL.

En la **cabecera** se declara información general como, por ejemplo, el nombre del módulo (es decir, el nombre de la transformación), los metamodelos de entrada y salida y la importación de librerías. Los ***helpers*** son subrutinas que se usan para evitar código redundante: se pueden ver como los equivalentes a los métodos en Java. Las **reglas** son la parte más importante de las transformaciones ATL, puesto que describen cómo se transforman los elementos del modelo origen en los elementos del modelo destino. Están formadas por vínculos (*bindings*), de modo que cada una expresa una correspondencia (*mapping*) entre un elemento de entrada y uno de salida. A continuación se muestran tres tipos de reglas existentes en ATL y sus propiedades: *ATL Matched rules*, *ATL Lazy rules* y *ATL Unique Lazy rules*.

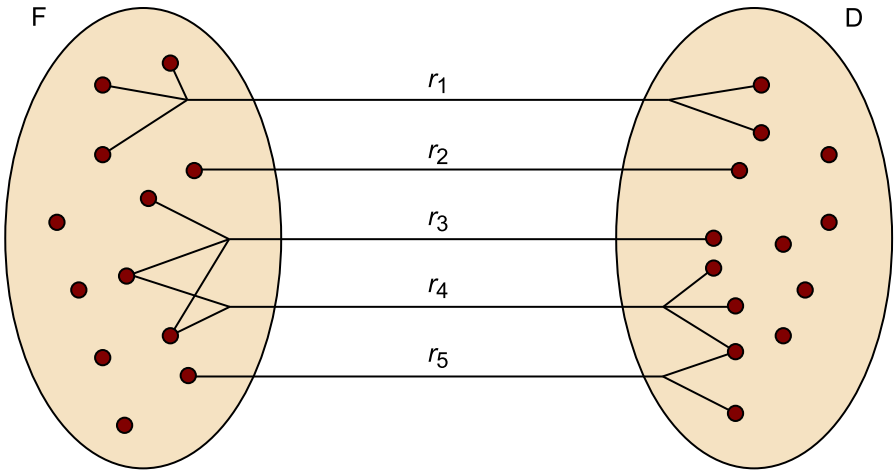
### *ATL matched rules*

En una transformación ATL vamos a tener reglas que se aplicarán sobre los elementos del modelo origen para generar elementos en el modelo destino. A las reglas declarativas que realizan esta acción se les llama *matched rules*. Estas reglas constituyen las piezas clave de las transformaciones ATL.

Una *matched rule* se identifica por el nombre que se le da, tiene como entrada uno o más tipos de objetos del modelo origen y genera uno o más tipos de objetos del modelo destino especificando los valores que reciben los atributos y referencias de los objetos creados en el modelo destino.

Estas reglas son declarativas, aunque también admiten una parte imperativa opcional, la cual se ejecuta una vez que ha terminado la parte declarativa.

Figura 27. Correspondencias entre los modelos origen y destino definidas en *matched rules*




### *ATL Lazy rules*

ATL también ofrece la posibilidad de usar otro tipo de reglas declarativas, llamadas *lazy rules* y *unique lazy rules*.

Las *lazy rules* son como las *matched rules* con la diferencia de que son ejecutadas solo cuando son llamadas desde otra regla (a las *matched rules* no se les invoca desde ningún sitio).

### *ATL Unique Lazy rules*

ATL ofrece un tercer tipo de regla declarativa llamada *unique lazy rule*. Estas reglas son un tipo especial de las *lazy rules* que siempre devuelven el mismo elemento destino para el mismo elemento origen. Las *lazy rules*, en cambio, siempre crean nuevos elementos del modelo destino en sus llamadas.

### Bloques imperativos

Como se mencionó anteriormente, ATL proporciona **constructores imperativos** que se deben usar cuando la transformación es demasiado compleja como para ser expresada declarativamente. La parte imperativa la componen las *called rules* y los *action blocks*. Los *action blocks* son secuencias de sentencias imperativas que constituyen la parte imperativa de las reglas declarativas. Se definen y ejecutan después de la parte declarativa y se pueden usar en vez de o en combinación con un elemento destino creado en la parte declarativa. En esta parte imperativa se permite hacer asignaciones, utilizar condiciones (con *if*), definir bucles (con *for*) e invocar *called rules*. Las *called rules* son reglas que se llaman desde la parte imperativa de otras reglas, por lo que hacen las veces de procedimientos. Así, estas reglas pueden verse como un tipo especial de *helpers*, ya que tienen que ser llamadas explícitamente para ser ejecutadas y aceptan parámetros de entrada. Sin embargo, al contrario de los *helpers*, estas reglas pueden generar elementos en el modelo destino. Estas reglas deben llamarse desde la parte imperativa de otras reglas.

## 4.4.3. Distintos enfoques para las transformaciones M2M

Según la forma en que se ejecutan las transformaciones de modelo a modelo, hay diversos mecanismos para especificarlas. Debido a la gran oferta que hay, es importante que los desarrolladores sean capaces de comparar y elegir las herramientas y lenguajes más apropiados para su problema concreto. De hecho, para la especificación de algunas transformaciones complejas entre modelos puede ser conveniente utilizar varios lenguajes. Anteriormente hemos descrito los dos lenguajes de transformación de modelos más importantes hoy día: QVT y ATL. A continuación mencionamos otros enfoques existentes en la actualidad:

**Web recomendada**

Más información sobre JMI en http://java.sun.com/products/jmi/

- **Enfoques de manipulación directa**. Son los de más bajo nivel. Este mecanismo proporciona una infraestructura mínima para organizar las transformaciones, por lo que los usuarios normalmente tienen que implementar sus reglas de transformación desde cero y con lenguajes de programación orientados a objetos como Java.

  Un ejemplo de estos lenguajes es JMI (Java *metadata interface*).

- **Enfoques dirigidos a estructuras**. Al realizar una transformación de modelos con estos lenguajes, se siguen dos fases: en la primera se crea la estructura jerárquica del modelo destino, en la segunda se establecen los atributos y referencias del modelo.

  Un ejemplo de este enfoque es el *framework* para realizar transformaciones M2M ofrecido por OptimalJ.

- **Enfoques operacionales**. Esta categoría engloba lenguajes que ofrecen un enfoque similar a la manipulación directa pero que ofrecen un soporte más

refinado al escribir las transformaciones. Así, por ejemplo, pueden ofrecer facilidades especiales como el *tracing* a través de librerías.

Ejemplos de lenguajes de este tipo son QVT *Operational Mappings*, XMF-Mosaic's executable MOF, MTL y Kermeta.

- **Enfoques basados en plantillas**. Estas plantillas son modelos que alojan metacódigo escrito en la sintaxis concreta del lenguaje destino. Esto ayuda al desarrollador a predecir el resultado de una instanciación de la plantilla. El desarrollador puede guiarse por la plantilla y completar lo que falta.

- **Enfoques relacionales**. En esta categoría están aquellos lenguajes declarativos donde el concepto principal son las relaciones matemáticas. La idea es especificar las relaciones entre los elementos origen y destino usando restricciones.

Ejempos de lenguajes de este tipo son QVT *Relations*, MTF, Kent Model Transformation Language, Tefkat y AMW.

- **Lenguajes basados en transformaciones de grafos**. Estos lenguajes operan sobre grafos tipados, etiquetados y con atributos, que no son más que representaciones en forma de grafo de modelos de clases simplificados. En estos lenguajes, un modelo se representa como un grafo donde los objetos son nodos y los links entre ellos son aristas. Cada regla describe un patrón en forma de subgrafo (parte izquierda de la regla) que, cada vez que aparece en el modelo (*matching*), debe sustituirse por otro patrón (parte derecha de la regla). A este tipo de transformaciones se les llama *in-place*, ya que es el grafo de entrada el que se va modificado a medida que se aplican las reglas de transformación hasta que no haya ningún *matching* de ninguna regla con el grafo. En ese momento, el resultado de la transformación es el propio grafo.

Entre los lenguajes de este tipo podemos encontrar AGG, AToM$^3$, VIATRA, GReAT, UMLX, BOTL, MOLA y Fujaba.

**Ved también**

Los lenguajes de transformación de grafos se mencionaron en el apartado "Semántica", porque también permiten definir la semántica de los lenguajes específicos de dominio.

- **Lenguajes híbridos**. Estos lenguajes combinan diferentes técnicas de las mostradas anteriormente, bien como componentes separados o bien de un modo más fino al nivel de las reglas.

Un ejemplo de este tipo de lenguajes es QVT, que se compone de los tres lenguajes explicados en el apartado 4.4.1: *Relations*, *Operational Mappings* y *Core*. ATL es otro ejemplo de lenguaje híbrido (admite mezclar bloques declarativos e imperativos en sus reglas).

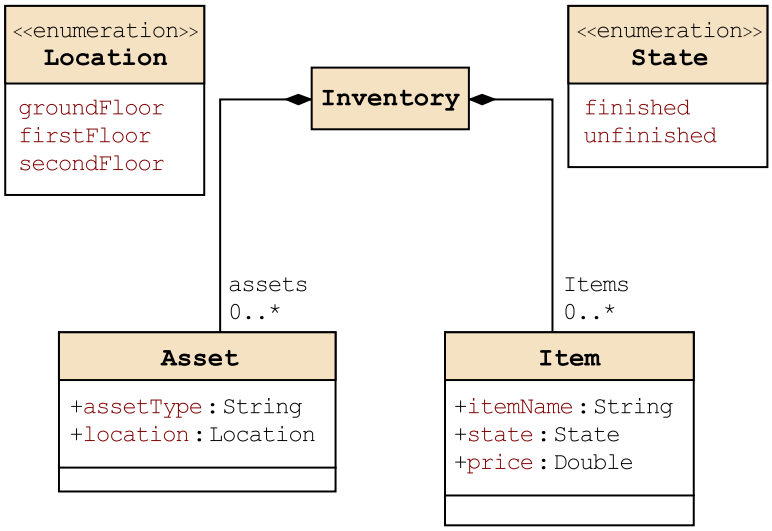
**Nota**

Algunos términos, como *matching* o *in-place*, no suelen traducirse del inglés porque han sido adoptados tal cual. Así no suele utilizarse "emparejamiento" para referirse a *matching*, o *in-situ* para transformaciones *in-place*.

### 4.4.4. Caso de estudio de transformación M2M (ATL)

En el apartado "Introducción a MDA y MDE" se estudió el modelado de una cadena de montaje cuyo metamodelo es el mostrado en la figura 7. Ahora nuestro objetivo es obtener vistas de los modelos de la cadena de montaje. Estas nuevas vistas van a ser modelos de inventario utilizados en la empresa, y cuyo metamodelo se muestra en la figura 28.

Figura 28. Metamodelo del inventario




Con este metamodelo se pueden definir modelos que ofrezcan información sobre el inventario de una cadena de montaje en un momento determinado. Así, un inventario se compone de los recursos inventariables de la cadena de montaje (clase `Asset`) y las piezas que se tengan en la fábrica en el momento que se realiza el inventario, es decir, los elementos fungibles (clase `Item`). Los objetos de tipo *Asset* son de un cierto tipo según sean un tipo de máquina (subclase de `Machine`) o una bandeja (`Tray`) en el modelo origen, y registran información sobre su ubicación (valores planta baja, primera o segunda) de acuerdo a un tipo enumerado `Location`. Los objetos de tipo `Item` tienen un nombre, según su clase (subclase de `Part` a la que pertenecen), un estado (terminado o no) especificado por un enumerado `State` y su precio.

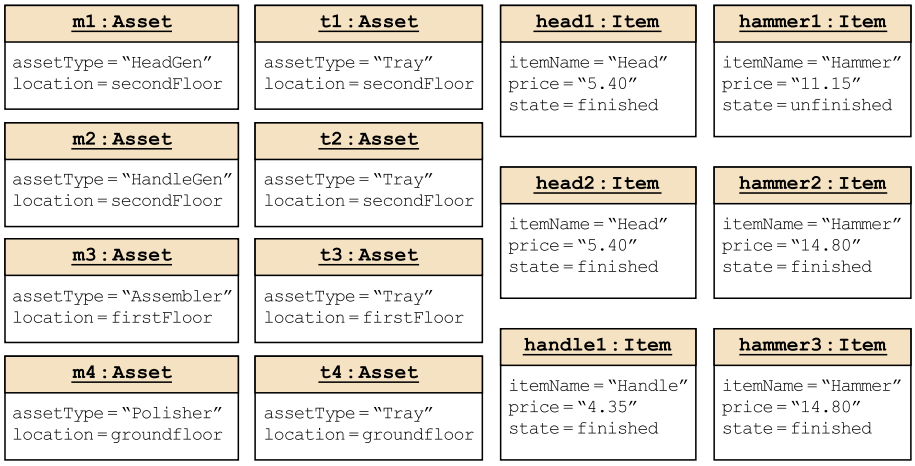
Centrémonos en el modelo de la cadena de montaje de la figura 9. Dicho modelo consta de cuatro máquinas (dos generadores, un ensamblador y una pulidora), cuatro bandejas y seis piezas (dos cabezas de martillo, un mango de martillo, un martillo no pulido y dos martillos pulidos). En el modelo no se especificó nada sobre la posición de los elementos, pero para la transformación vamos a suponer que sí contamos con esta información sobre el modelo y que dicha información es la mostrada en la tabla 1. La tabla refleja la posición (coordenadas $x$ e $y$) que suponemos para cada una de las máquinas y bandejas del modelo.

Tabla 1. Posiciones para los elementos del modelo de la figura 8

| pos | *m1* | *m2* | *t1* | *t2* | *m3* | *t3* | *m4* | *t4* |
|---|---|---|---|---|---|---|---|---|
| *x* | 6 | 6 | 6 | 6 | 4 | 4 | 1 | 1 |
| *y* | 1 | 4 | 2 | 5 | 3 | 4 | 5 | 7 |

Considerando esta información, y tomando como modelo de entrada de la transformación el modelo de la figura 9, el modelo de salida que deseamos obtener con nuestra transformación *Plant2Inventory* es el mostrado en la figura 29. Explicaremos en detalle cuáles son los criterios para asociar un piso (*location*), precio (*price*) y estado (*state*) a cada elemento del modelo al detallar cómo funciona la transformación.

Figura 29. Modelo de inventario resultado de la transformación *Plant2Inventory*




Vamos a describir ahora la transformación ATL que se encarga de llevar a cabo esa transformación.

En la **cabecera** de la definición de nuestra transformación declaramos el nombre del módulo y los metamodelos de entrada y salida:

```
module Plant2Inventory; -- Module Template
create OUT : Inventory  from IN : Plant;
```

**Nota**

Recordemos que toda transformación ATL está formada por una **cabecera** y una serie de **reglas**.

Por otro lado, las **reglas** que transformarán los elementos del modelo origen en los elementos del modelo destino en nuestro ejemplo son las siguientes:

**1)** Regla **Machine2Asset**: para cada instancia de la clase `Machine`, crea una instancia de la clase `Asset`.

- El `assetType` contendrá el nombre de la subclase concreta de `Machine`. Por ejemplo `'Headgen'`, `'Handlegen'`, `'Assembler'`, `'Polisher'`.

- El atributo `location` indica si la máquina está en la planta baja, primera o segunda. Estarán en la planta baja aquellas máquinas cuya coordenada `y` sea 0 o 1, en la primera planta aquellas cuya coordenada `y` esté entre 2 y 4 y en la segunda planta aquellas que tengan un valor entre 5 y 7 para dicha coordenada.

**2)** Regla **Tray2Asset**: para cada instancia de la clase `Tray`, crea una instancia de la clase `Asset`.

- El `assetType` contendrá el nombre de la clase, es decir, `Tray`.

- El atributo `location` indica si la máquina está en la planta baja, primera o segunda. Se calcula como se ha descrito en la regla anterior.

**3)** Regla **Part2Item**: para cada instancia de la clase `Part` crea una instancia de la clase `Item`.

- El atributo `itemName` contendrá el nombre de la subclase concreta de `Part`, es decir, `'Head'`, `'Handle'` o `'Hammer'`.

- El atributo `State` indica si la pieza está terminada o no. Se considera que los martillos que no han sido pulidos no están terminados. Por el contrario, los martillos pulidos, las cabezas de martillo y los mangos de martillo sí están terminados, pues pueden ser reutilizados para cualquier otro fin.

- `Price` indica el precio de la pieza. Las cabezas de martillo tienen un precio de 5,40 euros, los mangos cuestan 4,35 euros, un martillo sin pulir 11,15 euros y uno pulido 14,80 euros.

### 4.4.5. *ATL Matched rules*

Para mostrar el uso de las *matched rules*, el siguiente extracto de código corresponde a la primera de las reglas descritas en el apartado anterior:

```
rule Machine2Asset {
   from  m : Plant!Machine
   to    a : Inventory!Asset(
               assetType <- m.getMachineType(),
               location <- m.y.getFloor()
           )
}
```

La regla se llama `Machine2Asset`, y va a tener de entrada una máquina que va a generar un activo. Los elementos de entrada se especifican tras la palabra reservada `from`, y los de salida tras la palabra reservada `to`. Cuando se quiere hacer referencia a una clase se escribe el nombre del metamodelo (dicho nom-

bre se asigna en la cabecera de la transformación) seguido de "! " y del nombre de la clase. Así, para referirse a la clase `Machine` hay que escribir `Plant!Machine`, y para referirse a la clase `Asset` hay que escribir `Inventory!Asset`. En la generación del elemento de salida vemos cómo se inicializan sus atributos. En este caso, los dos atributos de `Asset`, `assetType` y `location`, se han inicializado haciendo uso de sendos *helpers*:

```
helper context Plant!Machine def : getMachineType() : String =
   if (self.oclIsTypeOf(Plant!HeadGen))
      then 'HeadGen'
      else if (self.oclIsTypeOf(Plant!HandleGen))
         then 'HandleGen'
         else if (self.oclIsTypeOf(Plant!Assembler))
            then 'Assembler'
            else 'Polisher'
         endif
      endif
   endif ;

helper context Integer def : getFloor() : Inventory!Location =
   if (self >= 0 and self <= 1)
     then #groundFloor
     else if (self >= 2 and self <= 4)
       then #firstFloor
       else #secondFloor
    endif
   endif ;
```

Como se mencionó anteriormente, los *helpers* nos permiten definir procedimientos y funciones. Permiten definir código ATL al que se puede llamar desde distintas partes de la transformación. Un *helper* se define con los siguientes elementos:

- Su nombre, utilizado para invocarlo.
- Un tipo de contexto (opcional). Especifica el contexto en el que se define el atributo. Un `self` que se utiliza dentro del *helper* se refiere al contexto.
- Valor a devolver. En ATL, todo *helper* debe devolver un valor.
- Una expresión ATL que representa el código del *helper*.
- Un conjunto de parámetros de entrada (opcional). Al igual que en Java, un parámetro de entrada es identificado por una pareja (nombre del parámetro, tipo del parámetro).

El primero de los *helpers* anteriores, `getMachineType`, recibe un objeto de tipo máquina como entrada y devuelve una cadena de texto que se corresponde con el nombre de la subclase concreta de la máquina. Como ATL utiliza OCL, en este *helper* se ha hecho uso del método `oclIsTypeOf` que se explicó en

el apartado 2. El *helper* comprueba a qué subclase no abstracta de `Machine` pertenece el objeto desde el que se llama al *helper* y devuelve el nombre de la clase como cadena de texto.

El segundo *helper*, `getFloor`, recibe un entero que se corresponde con la coordenada `y` de un objeto y devuelve la ubicación a la que corresponde dicha coordenada `y`. El valor a devolver es del tipo enumerado `Location`. Podemos observar en el *helper* que para especificar los valores de un tipo enumerado se utiliza el caracter '#' seguido del nombre del literal.

La segunda de las reglas especificadas anteriormente (`Tray2Asset`) se corresponde con la siguiente *matched rule*:

```
rule Tray2Asset {
   from  t : Plant!Tray
   to    a : Inventory!Asset(
            assetType <- 'Tray',
            location <- t.y.getFloor()
         )
}
```

En esta regla también se utiliza el *helper* `getFloor` para obtener la ubicación de la bandeja. Esta regla es fácilmente entendible tras la explicación de la anterior. La tercera regla (`Part2Item`) se muestra a continuación:

```
rule Part2Item {
   from  p : Plant!Part
   to    i : Inventory!Item(
            itemName <- p.getItemName(),
            state <- p.getPartState(),
            price <- p.getPrice()
         )
}
```

Esta regla convierte los objetos de tipo `Part` del modelo origen en objetos de tipo `Item` en el modelo destino. Observamos que los valores de los atributos del elemento destino se han inicializado utilizando tres *helpers*. El primero de ellos, `getItemName`, es muy similar al *helper* `getMachineType` mostrado anteriormente. El *helper* `getPartState` es:

### Ejercicio propuesto

Definir el *helper* `getItemName`.

```
helper context Plant!Part def : getPartState() : Inventory!State =
   if self.oclIsTypeOf(Plant!Head) or self.oclIsTypeOf(Plant!Handle)
     then #finished
     else if  (self.oclIsTypeOf(Plant!Hammer) and self.isPolished)
      then #finished else #unfinished
     endif
   endif ;
```

Este *helper* asigna el estado `finished` a aquellas piezas que sean cabezas o mangos. Para los martillos, comprueba si está pulido (para asignarle `finished`) o no (para asignarle `unfinished`). El tercer *helper*, `getPrice`, asigna el precio a cada pieza como se explicó anteriormente.

**Ejercicio propuesto**

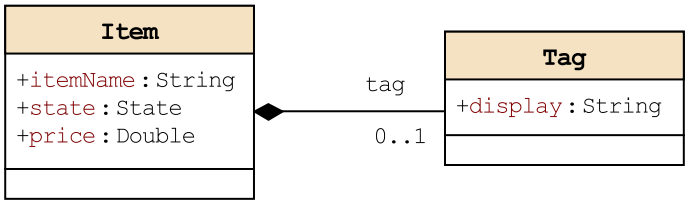
Definir el *helper* `getPrice` cuya cabecera es:

```
helper context Plant!Part def : getPrice() : Real.
```

### 4.4.6. *ATL Lazy rules*

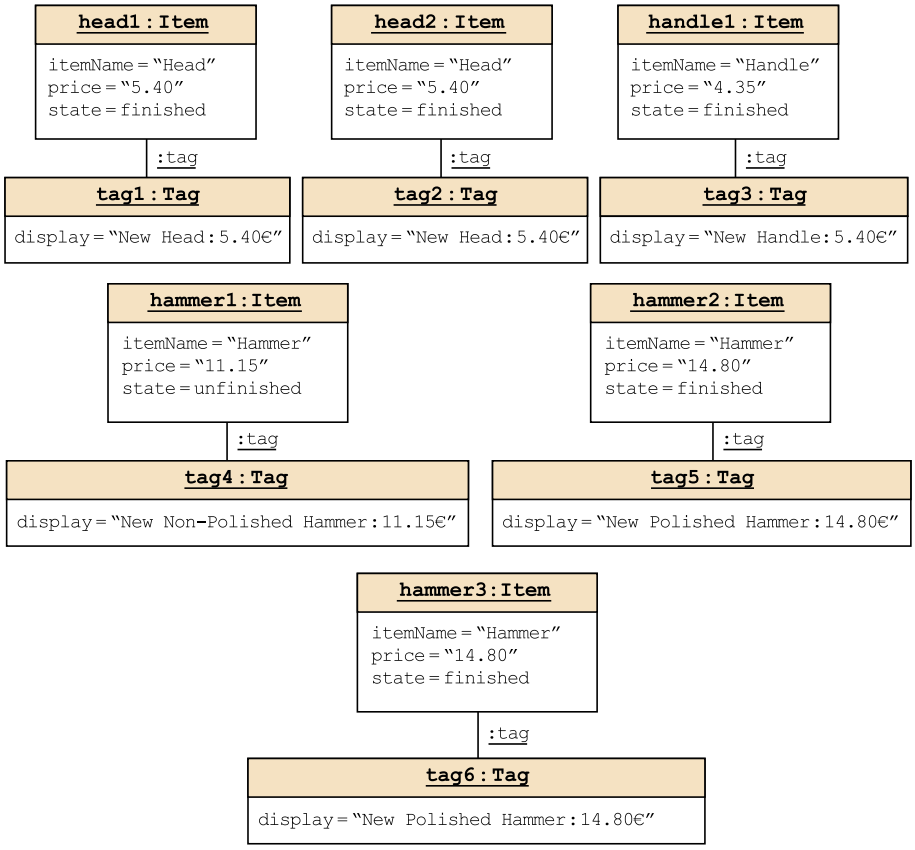
Para mostrar el uso de estas reglas, supongamos que añadimos una clase adicional `Tag` al metamodelo de inventarios, tal y como muestra la figura 30.

Figura 30. Extensión del metamodelo de inventarios




El objetivo de la nueva clase `Tag` es etiquetar los artículos para poder venderlos. Así, siempre que se cree una instancia de tipo `Item` mediante la transformación, ahora también se querrá tener una etiqueta del mismo.

Figura 31. Modelo de inventario resultado de la transformación `Plant2InventoryWithTags` (los objetos de tipo `Asset` no se muestran)




Llamemos a la nueva transformación `Plant2InventoryWithTags`. El modelo de salida que debe quedar ahora (figura 31) es una extensión del modelo de la figura 28 en el que aparecen objetos de tipo `Tag`: Para crear los objetos etiqueta podemos utilizar una *lazy rule*.

```
rule Part2Item2 {
   from  p : Plant!Part
   to    i : Inventory!Item(
               itemName <- p.getItemName(),
               state <- p.getPartState(),
               price <- p.getPrice(),
               tag <- thisModule.CreateTag(p)
             )
}
lazy rule CreateTag {
   from  p : Plant!Part
   to    t : Inventory!Tag(
               display <- p.getDisplay()
             )
}
```

La regla `Part2Item2` es la misma *matched rule* de antes, aunque ahora hay que darle un valor a la nueva referencia, `Tag`, cuando se crea una instancia de la clase `Item`. Para crear la nueva instancia de tipo `Tag` llamamos a una *lazy rule* llamada `CreateTag` y le pasamos el objeto de tipo `Part` que disparó la *matched rule*. Para invocar una *lazy rule* hay que escribir "`this Module`" delante de

su nombre. Los objetos de entrada de la *lazy rule*, que aparecen seguidos de la palabra reservada `from`, son los que se pasan como parámetro. Esta *lazy rule* crea un objeto de tipo `Tag` e inicializa su único atributo mediante un *helper*.

**Ejercicio propuesto**

Definir el *helper* `getDisplay()` deduciendo su comportamiento observando el modelo de la figura 28.

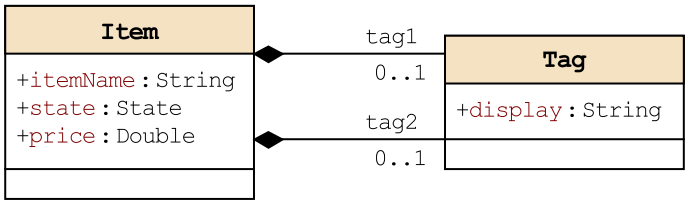
El código anterior en el que se ha utilizado una *lazy rule* se podría haber escrito, alternativamente, solo con la *matched rule* y extendiéndola para que ella misma crease los dos elementos destino (el `Item` y el `Tag`):

```
rule Part2Item {
   from  p : Plant!Part
   to    i : Inventory!Item(
            itemName <- p.getItemName(),
            state <- p.getPartState(),
            price <- p.getPrice(),
            tag <- t
          ),
         t : Inventory!Tag(
            display <- p.getDisplay()
          )
}
```

### 4.4.7. *ATL Unique Lazy rules*

Vamos a explicar este tipo de reglas con un ejemplo. Imaginemos que en el metamodelo de inventarios los artículos pueden tener ahora dos etiquetas (iguales), y que las modelamos con relaciones `tag1` y `tag2`:

Figura 32. Extensión del metamodelo de inventarios

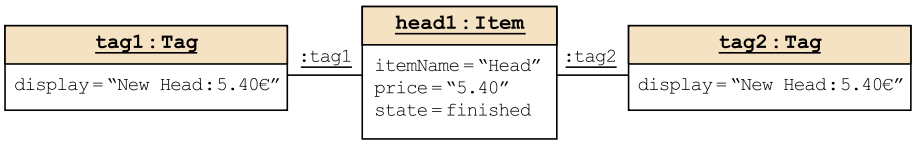



Ahora, en la regla `Part2Item` debemos tener en cuenta las dos relaciones. El código utilizando la *lazy rule* anterior puede ser el siguiente:

```
rule Part2Item3 {
   from  p : Plant!Part
   to    i : Inventory!Item(
               itemName <- p.getItemName(),
               state <- p.getPartState(),
               price <- p.getPrice(),
               tag1 <- thisModule.CreateTag(p),
               tag2 <- thisModule.CreateTag(p)
             )
}
lazy rule CreateTag {
   from  p : Plant!Part
   to    t : Inventory!Tag(
               display <- p.getDisplay()
             )
}
```

Observamos que se ha llamado a la misma *lazy rule* dos veces con el mismo elemento de entrada. Si nos fijamos solo en el objeto `head1` de la figura 33, ahora tiene dos etiquetas asociadas. Ambas tienen el mismo contenido pero son **dos instancias diferentes** de la clase `Tag`:

Figura 33. Creación de dos instancias de la clase `Tag` invocando dos veces a la misma *lazy rule*

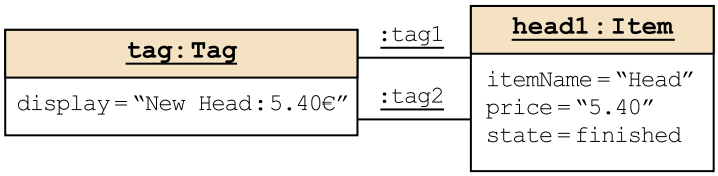



Si la regla `CreateTag`, que ahora mismo es una *lazy rule*, la reemplazamos por una *unique lazy rule*, se obtiene un resultado diferente. Sintácticamente, una *unique lazy rule* se diferencia de una *lazy rule* solo en la definición de la misma, que incluye la palabra `unique`:

```
unique lazy rule CreateTag {
   from  p : Plant!Part
   to    t : Inventory!Tag(
         display <- p.getDisplay()
       )
}
```

Ahora, la primera invocación de la regla `CreateTag` desde la regla `Part2Item` crea un objeto de tipo `Tag` y este es asociado con el objeto de tipo `Item` mediante la relación `tag1`. En la segunda llamada a `CreateTag`, que ahora es una *unique lazy rule*, el motor de ejecución de ATL recupera la instancia que se creó en la anterior llamada, pues el parámetro de entrada es el mismo, en vez de crear un nuevo objeto. De este modo, la relación `tag2` asocia el mismo item con el mismo tag que la relación `tag1`.

Figura 34. Modelo correspondiente al de la figura 33 pero usando una *unique lazy rule* en vez de *lazy rule*




### 4.4.8. Bloques imperativos

Vamos a ver un ejemplo muy simple sobre cómo se escribe la parte imperativa de una regla. El uso que le damos a la parte imperativa en este ejemplo es el de modificar un objeto que se ha creado en la parte declarativa de la misma regla. Recordemos la regla `Part2Item` vista anteriormente. En la misma, se crea un objeto `Item` y otro `Tag` a partir de un objeto de tipo `Part`. En la parte declarativa se le asigna un nombre al objeto de tipo `Item` (en su atributo `itemName`) mediante la llamada al *helper* `getItemName`. Imaginemos que queremos añadir algo más a ese nombre, y para ello vamos a usar la parte imperativa de la regla.

Un ejemplo es el siguiente:

```
rule Part2Item {
   from  p : Plant!Part
   to    i : Inventory!Item(
                 itemName <- p.getItemName(),
              …
           )
   do {
            i.itemName <- i.itemName + ' imperative code';
   }
}
```

Al ejecutar esta regla, el campo `itemName` de un objeto de tipo martillo será "`Hammer imperative code`" en vez de simplemente "`Hammer`". Hay que destacar que este sencillo ejemplo simplemente muestra la forma de cómo se escribe código en la parte imperativa de ATL y cómo se accede a los objetos creados en la parte declarativa y a sus atributos. En este ejemplo la cadena de caracteres que se añade se podría haber añadido en la parte declarativa.

## 4.5. Lenguajes de transformación modelo-a-texto (M2T)

Las transformaciones de modelo-a-texto (M2T) son aquellas que toman un modelo como entrada y devuelven una cadena de texto como salida. Existen dos enfoques a la hora de especificar estas transformaciones:

- Enfoques basados en **visitas**. Consistente en proporcionar un mecanismo para recorrer la representación interna de un modelo y escribir texto en un flujo de texto.

  Un ejemplo es Jamda, un *framework* orientado a objetos que proporciona un conjunto de clases para representar modelos UML, una API para manipular modelos y un mecanismo de visita para generar código.

**Web recomendada**

Más información sobre Jamda en The Jamda Project: http://jamda.sourceforge.net

- Enfoques basados en **plantillas**. La mayoría de las herramientas MDA disponibles actualmente soportan la generación de código con este mecanismo: openArchitectureWare, JET, FUUT-je, Codagen Architect, AndroMDA, ArcStyler, MetaEdit+, OptimalJ, TCS, MOFScript. Una plantilla consiste en que se da el texto destino y contiene fragmentos de metacódigo para acceder a información del modelo origen y permite la expansión iterativa del texto.

Entre las herramientas existentes destacamos **MOFScript**, una herramienta y lenguaje para declarar transformaciones M2T. Proporciona un lenguaje independiente del metamodelo que permite usar cualquier tipo de metamodelo y sus instancias para la generación de código. También puede ser utilizado para definir transformaciones de modelo a modelo cuando se quiere construir el modelo desde cero o para aumentar modelos existentes. Esta herramienta está basada en EMF y Ecore como plataforma de metamodelado. El segundo lenguaje que resaltamos es **TCS**, el cual se describe en el siguiente apartado.

**Web recomendada**

Más información sobre MOFScript en http://eclipse.org/gmt/mofscript/doc/

### 4.5.1. TCS: Textual Concret Syntax

TCS es un lenguaje para especificar transformaciones de modelo a texto que proporciona maneras de asociar elementos sintácticos (es decir, palabras clave como `if`, símbolos especiales como "+") con elementos del metamodelo con muy poca redundancia. Tanto las transformaciones M2T como T2M pueden ser desarrolladas utilizando una especificación sencilla. Para realizar la transformación se necesita su especificación TCS y el metamodelo y modelo origen.

### 4.5.2. Caso de estudio de M2T

Vamos a mostrar la sencillez de TCS a través de un caso de estudio en el que queremos convertir en texto modelos conforme al metamodelo de la figura 27. Por ejemplo, si tenemos el modelo del inventario mostrado en la figura 28, nuestro objetivo es definir una transformación `Inventory` que produzca exactamente el texto de la figura 35, que nos da información textual sobre el inventario.

Figura 35. Información textual sobre el inventario de la figura 28

```
New Inventory:
  Asset of type HeadGen in second floor
  Asset of type HandleGen in second floor
  Asset of type Assembler in first floor
  Asset of type Polisher in ground floor
  Asset of type Tray in second floor
  Asset of type Tray in second floor
  Asset of type Tray in first floor
  Asset of type Tray in ground floor
  Item: finished Head with cost 5.40 €
  Item: finished Head with cost 5.40 €
  Item: finished Handle with cost 4.35 €
  Item: unfinished Hammer with cost 11.15 €
  Item: finished Hammer with cost 14.80 €
  Item: finished Hammer with cost 14.80 €
End of Inventory
```

Para generar automáticamente esta información podemos utilizar la siguiente transformación modelo a texto escrita en TCS:

```
syntax Inventory {
  enumerationTemplate Location auto :
       #groundFloor = "ground floor",
       #firstFloor     = "first floor",
       #secondFloor = "second floor" ;
  enumerationTemplate State auto :
       #finished      = "finished",
       #unfinished  = "unfinished" ;
  template Inventory main :
     "New Inventory:"
     [assets items]{nbNL = 1, indentIncr = 1}
     "End of Inventory" ;
  template Asset :
    "Asset of type " assetType " in " location ;
  template Item :
    "Item: " state " " itemName " with cost " price "€" ;
}
```

Las dos primeras plantillas son introducidas para especificar lo que hay que escribir cuando aparezca un dato de tipo enumerado. A la izquierda de la asignación se escriben los valores del tipo enumerado y a la derecha lo que se quiere que aparezca en el texto.

**Consulta recomendada**

En el artículo "TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering", de Frédéric Jouault, Jean Bézivin e Ivan Kurtev, se explica en profundidad TCS. También hay más información en: http://wiki.eclipse.org/index.php/TCS

La plantilla `Inventory` es la que inicia la transformación. Se pone entre comillas lo que se quiere que aparezca tal cual siempre que nos encontremos con un objeto de tipo `Inventory`. Así, al principio se escribe "`New Inventory`:". Lo que aparece en la línea justo después, `[assets items]`, se refiere a la relación de `Inventory` con `Asset` y con `Item`, respectivamente. Así, tras la escritura de "`New Inventory`:" en el fichero de salida, se volcará el texto proporcionado por las plantillas para los objetos de tipo `Asset` y a continuación el de las plantillas asociadas a los objetos de tipo `Item`. Lo que aparece después, `{nbNL = 1, identIncr = 1}`, hace que la información de cada objeto se escriba en una línea nueva y se aplique sangría. Una vez procesados todos los objetos de tipos `Asset` e `Item` se escribe la cadena "`End of Inventory`".

En las plantillas para los objetos `Asset` e `Item` se puede observar cómo se accede a la información de los atributos de los objetos: simplemente escribiendo el nombre del atributo. Con ello se obtiene una cadena de texto que representa el valor del atributo. Así, en las dos plantillas se utilizan cadenas de texto fijas que se concatenan con los valores recuperados de los atributos.

## 4.6. Conclusiones

Junto con los modelos, las transformaciones forman parte esencial del desarrollo de software dirigido por modelos y MDA. En este apartado hemos presentado los conceptos principales de las transformaciones de modelos, sus mecanismos básicos, y hemos ilustrado ejemplos usando dos de los principales lenguajes de transformación de modelos como son QVT y ATL.

Aunque estos lenguajes ya están bastante maduros y son ampliamente utilizados, aún precisan mejoras en algunas áreas concretas. Por ejemplo, aún no existen depuradores suficientemente potentes, ni herramientas de especificación y pruebas de transformaciones de modelos. Aunque al principio las transformaciones que se necesitaban eran razonablemente pequeñas, su tamaño y complejidad ha crecido últimamente de manera notable: actualmente muchos proyectos utilizan transformaciones ATL con cientos de reglas y miles de líneas de código. También se precisan avances en las teorías que soportan los lenguajes de transformación de modelos para poder razonar sobre las transformaciones y sus reglas, y poder definir mecanismos de modularidad, reutilización, extensión, refactorización, herencia, etc. En este sentido, la comunidad científica trabaja intensamente en estos aspectos, que suelen debatirse en el seno de conferencias especializadas como ICMT (International Conference on Model Transformations).

# Resumen

En este módulo hemos presentado los conceptos y mecanismos que se utilizan en el desarrollo de software dirigido por modelos, una propuesta para el desarrollo de software en la que los modelos, las transformaciones de modelos y los lenguajes específicos de dominio juegan los papeles principales, frente a las propuestas tradicionales basadas en lenguajes de programación, plataformas de objetos y componentes software.

El objetivo de MDD es reducir los costes y tiempos de desarrollo de las aplicaciones software y mejorar la calidad de los sistemas que se construyen, con independencia de las plataformas de implementación y garantizando las inversiones empresariales frente a la rápida evolución de la tecnología.

Aunque aún en sus comienzos, MDD ya ha cobrado cuerpo como disciplina reconocida y cada vez más aceptada por la comunidad de ingeniería del software para el desarrollo de aplicaciones y grandes sistemas. Por supuesto, MDD no es la panacea que es aplicable a todo tipo de situaciones y proyectos, y quizá un aspecto determinante para su implementación con éxito es saber determinar en qué proyectos, empresas y circunstancias interesa realmente aplicarlo y en cuáles no. Y para eso es imprescindible conocer a fondo sus principales conceptos, fundamentos y herramientas; ese ha sido precisamente el objeto del presente módulo.

## Actividades

**1.** Definid el metamodelo de los comandos en línea de GNU/Linux, que constan de un nombre, cero, uno o más argumentos y cero, uno o más *flags* (argumentos precedidos de guión). Indica el modelo correspondiente al comando: "rm –rf uoc.bak uoc.tmp", compuesto por el nombre, dos *flags* y dos argumentos.

**2.** Modificad el modelo de la cadena de montaje de martillos para tener en cuenta que las máquinas (generadores, ensamblador y pulidora) pueden producir piezas defectuosas. ¿Cómo se refleja esto en el resto de los modelos?

**3.** Modificad la transformación de modelos Plant2Inventory teniendo en cuenta los cambios realizados en los diferentes metamodelos para considerar piezas defectuosas, según la actividad anterior.

**4.** Especificad en el modelo de la cadena de montaje una operación auxiliar OCL que permita conocer cuál es la bandeja de entrada más cercana a una máquina. La distancia se mide con la operación "`distanceTo()`" especificada en el apartado "Creación de variables y operaciones adicionales". Señalad otra que devuelva la bandeja más cercana, ya sea de entrada o de salida.

**5.** Definid un metamodelo para Twitter, con sus principales conceptos (tweet, usuario, mensaje directo, lista, etc.) y las relaciones entre ellos (por ejemplo, los seguidores de un usuario, los *tags* de un tweet, etc.).

**6.** Añadid operaciones al metamodelo de Twitter desarrollado en la actividad anterior (p. ej., publicar un tweet, seguir a un usuario o responder a un mensaje), y especificad su comportamiento utilizando OCL.

**7.** Considerad el modelo de la figura 13, y definid cuatro metamodelos, cada uno representando la sintaxis abstracta de las cuatro interpretaciones diferentes del modelo mencionadas en el texto, y comparadlos entre sí. ¿Cuáles son las diferencias entre ellos?

**8.** Definid el *helper* `getPrice` utilizado en la transformación Plant2Inventory (apartado "ATL *matched rules*") para asignar precio a las piezas.

**9.** Definid un metamodelo con la sintaxis abstracta de un subconjunto mínimo del lenguaje Java, que solo contenga la definición de clases, sus atributos y métodos.

**10.** Implementad una transformación de modelos en ATL que tome como entrada un modelo conforme al metamodelo de Java definido en el ejercicio anterior y devuelva otro modelo conforme a ese mismo metamodelo, pero donde todos los atributos públicos se conviertan en privados y se generen los correspondientes métodos *getter* y *setter* para ellos.

## Ejercicios de autoevaluación

**1.** Indicad las principales diferencias entre MDA, MDD y MDE.

**2.** Describid la diferencia entre un modelo y un metamodelo. Indicad dos lenguajes de modelado y dos de metamodelado.

**3.** Describid los componentes principales del patrón MDA.

**4.** Describid el significado y uso de la palabra reservada `context` en OCL.

**5.** Señalad la diferencia entre los operadores "^" y "^^" de OCL.

**6.** Indicad los principales componentes de un lenguaje específico de dominio, y explicad cómo se especifica cada uno de ellos.

**7.** Describid las tres formas que ofrece OMG para definir lenguajes específicos de dominio.

**8.** Describid la diferencia entre *tag definition* y *tag value* en la definición de un perfil UML.

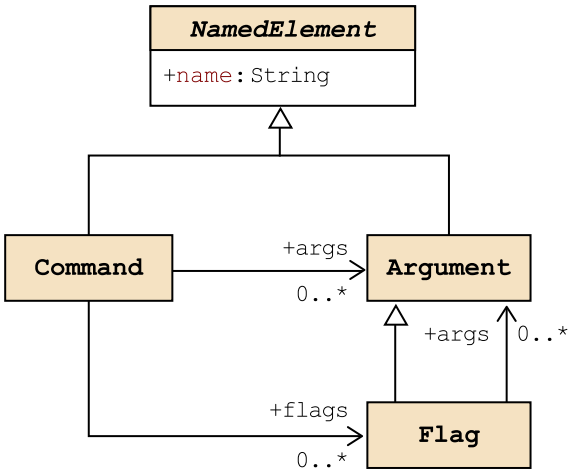
**9.** Describid la diferencia entre una transformación de modelos *in-place* y una que no lo sea.

**10.** Describid las diferencias entre los tres tipos de reglas declarativas de ATL: *matched rules*, *lazy rules* y *unique lazy rules*.

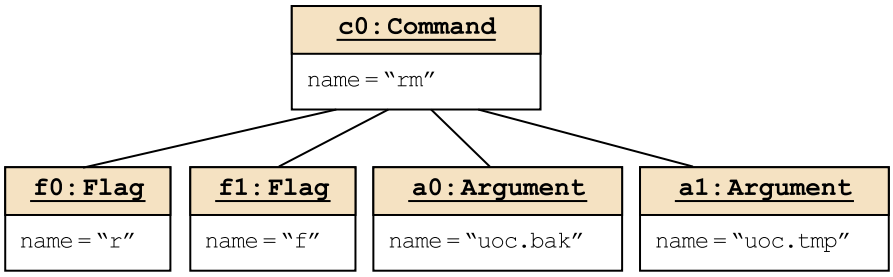
# Solucionario

**Actividades**

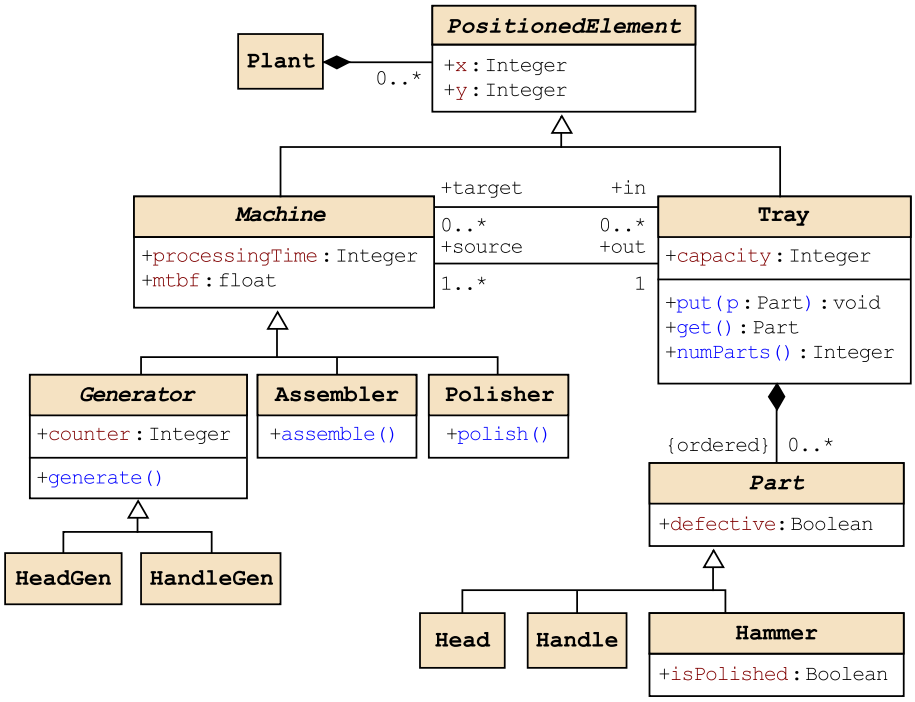
**1.** El metamodelo pedido puede representarse de la siguiente manera:



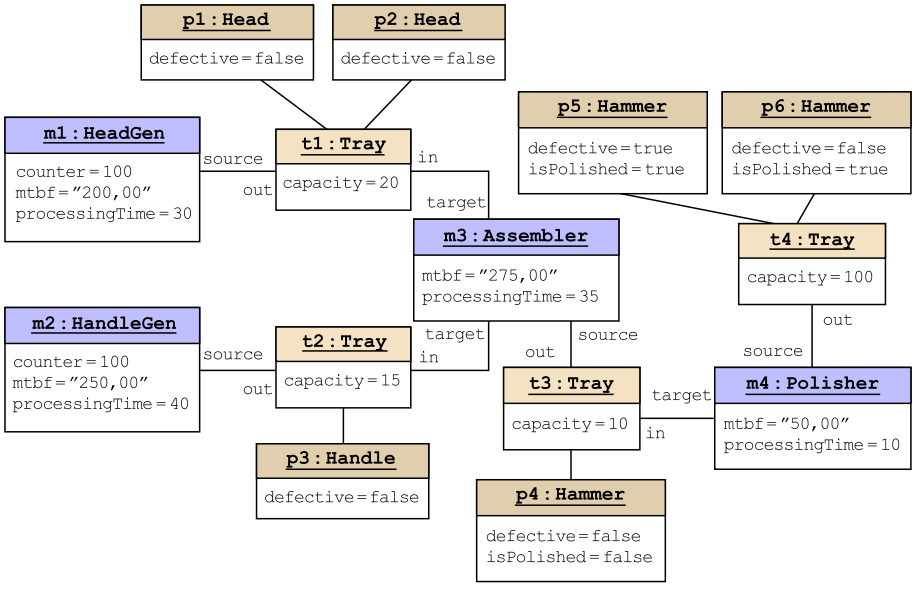

Dado este metamodelo, el modelo correspondiente al comando "rm –rf uoc.bak uoc.tmp" sería el siguiente:




**2.** Los nuevos requisitos podrían ser implementados mediante la inclusión de un atributo (*defective: boolean*) en la clase `Part` y un atributo (*mtbf: float*) en la clase `Machine`, tal y como puede verse en el metamodelo modificado:




El atributo *defective* almacena la información sobre si la pieza es defectuosa o no, y el atributo *mtbf* almacena un valor numérico que indica el tiempo medio entre fallos de la máquina (medido en segundos). Con estos cambios, un posible modelo conforme a este metamodelo es el mostrado a continuación, en el que la pulidora es rápida pero tiene una elevada tasa de fallos, y de hecho ha producido una pieza defectuosa (p5):




**3.** Para tener en cuenta piezas defectuosas, según la actividad anterior, consideramos que el nuevo metamodelo origen es PlantModified, y que el modelo destino (Inventory) no cambia. Solamente no se tienen en cuenta las piezas defectuosas, que son excluidas.

```
module PlantModified2Inventory;
create OUT : Inventory  from IN : PlantModified;
rule Machine2Asset {
   from  m : PlantModified!Machine
   to    a : Inventory!Asset(
              assetType <- m.getMachineType(),
              location <- m.y.getFloor()    )
}
rule Tray2Asset { -- does not change
   from  t : PlantModified!Tray
   to    a : Inventory!Asset(
               assetType <- 'Tray',
               location <- t.y.getFloor() )
}
rule Part2Item {
   from  p : PlantModified!Part (not p.defective)
   to    i : Inventory!Item(
               itemName <- p.getItemName(), ),
               state <- p.getPartState(),
               price <- p.getPrice() )
}
```

El resto de la transformación (por ejemplo, los *helpers*) no cambia.

**4.** La operación auxiliar que devuelve la bandeja de entrada más cercana a una máquina puede especificarse como sigue:

```
context Machine
def closestInTray() : Tray
pre:  self.in->notEmpty()
post: result = self.in->any(c : Tray |
        forAll(t : Tray | c.distanceTo(self) <= t.distanceTo(self)))
```

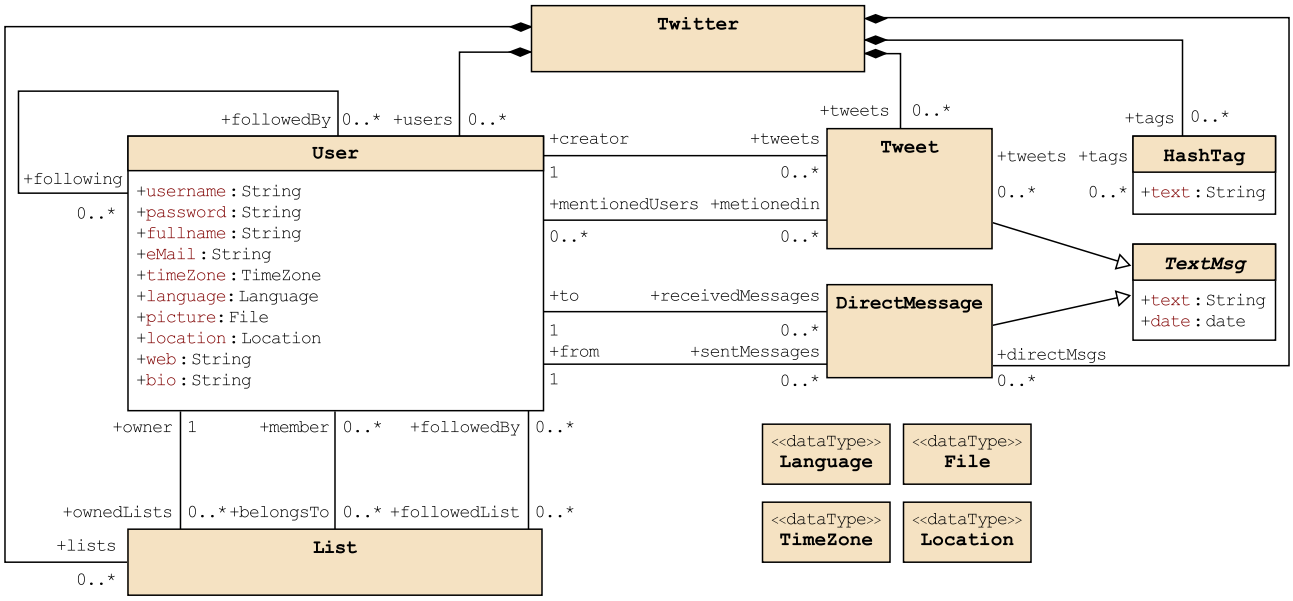
La operación auxiliar que devuelve la bandeja más cercana a una máquina, independientemente de si es de entrada o de salida puede especificarse de dos formas distintas. En primer lugar como:

```
context Machine
def closestTray() : Tray
pre:  self.in->union(self.out)->notEmpty()
post: result = self.in->union(self.out)->any(c : Tray |
        forAll(t : Tray | c.distanceTo(self) <= t.distanceTo(self)))
```

o, alternativamente:

```
context Machine
def closestTray() : Tray
pre:  self.in->notEmpty() or self.out->notEmpty()
post: result = Tray.allInstances()->any(c : Tray |
        forAll(t : Tray | c.distanceTo(self) <= t.distanceTo(self)))
```

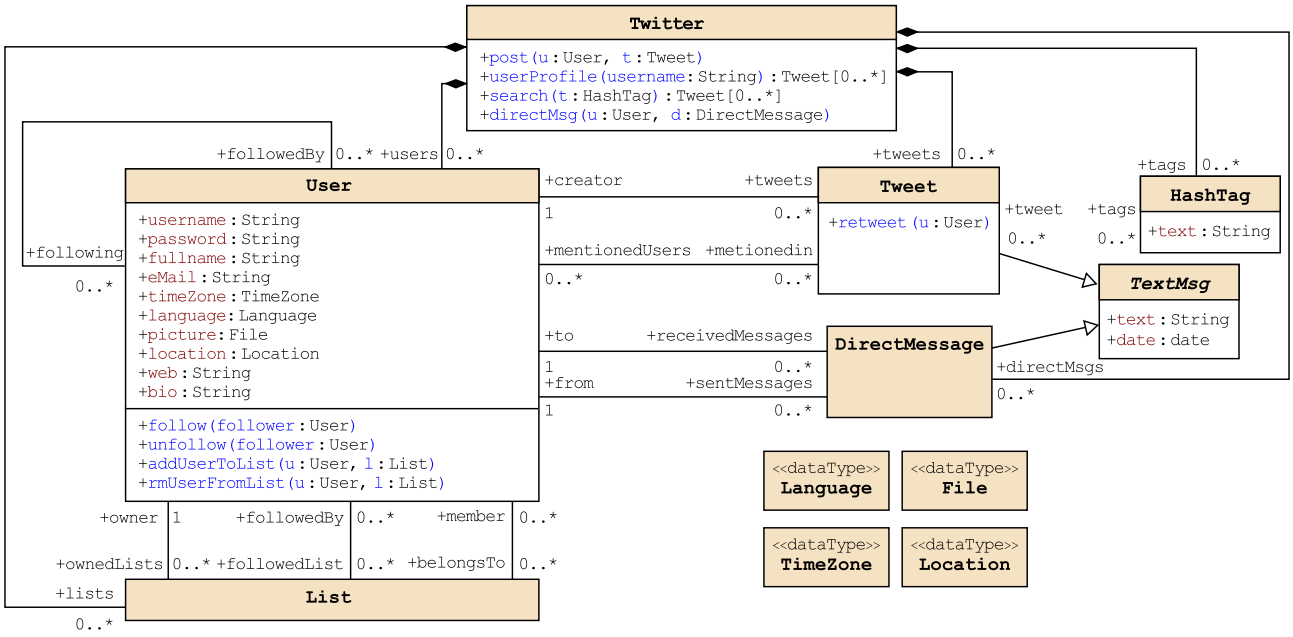
**5.** Un posible metamodelo para Twitter, descrito por un diagrama UML con sus principales conceptos y las relaciones entre ellos es el siguiente:




En cuanto a las restricciones de integridad, podemos considerar:

```
context Twitter inv usernameIsKey:
   self.users->isUnique(username)

context User inv noAutoFollow:
   self.following->excludes(self) and self.followedBy->excludes(self)

context TextMsg inv hundredFortyCharsIsEnough:
   self.text.size()<=140

context Tweet inv validMention:
   self.mentionedUser->forAll(u |
            self.text.indexOf("@"+u.username)<>0))

context Tweet inv mentionedTag:
   self.tags->forAll(t | (self.text.indexOf(t.text)<>0))

context HashTag inv validTag:
   self.text.indexOf("#")=1  -- startsWith "#"
   and     -- and does not contain another "#"
   self.text.substring(2,self.text.size()).indexOf("#") = 0
   and     -- and does not contain spaces
   self.text.indexOf(" ") = 0

context DirectMessage inv validMsg:
   -- starts with "D" and then a username
   self.text.indexOf("D @"+self.to.userName)=1
```

**6.** El metamodelo de Twitter puede ser enriquecido con operaciones como muestra el siguiente diagrama:




Dichas operaciones pueden ser especificadas en OCL como sigue:

```
context Twitter::post(u:User,t:Tweet)
pre:  t.indexOf("D @") <> 1  -- it is not a direct message
post:    self.tweets->includes(t) and
      u.tweets->includes(t) and
         t.creator = u and
      t.tags->includesAll(t.mentionedTags()) and
      t.mentionedUsers->includesAll(t.mentionedUsers())

context Twitter::directMsg(u:User,d:DirectMsg)
pre: -- user u can send direct messages only to users that follow u
    let uname : String =
         d.substring(4,d.substring(4,d.size()).indexOf(" ")) in
      u.followedBy->any(u | u.username = uname)
post: let uname : String =
         d.substring(4,d.substring(4,d.size()).indexOf(" ")) in
      self.directMsgs->includes(d) and
      d.to = self.users->one(username = uname) and
      d.from = u

-- AUXILIARY OPERATIONS
context Tweet::mentionedTags() : Set{HashTag}
-- returns the set of tags mentioned in a tweet
body: ...
context Tweet::mentionedUsers() : Set{User}
-- returns the set of users mentioned in a tweet
body: ...
```

**7.** Las posibles interpretaciones son las siguientes:

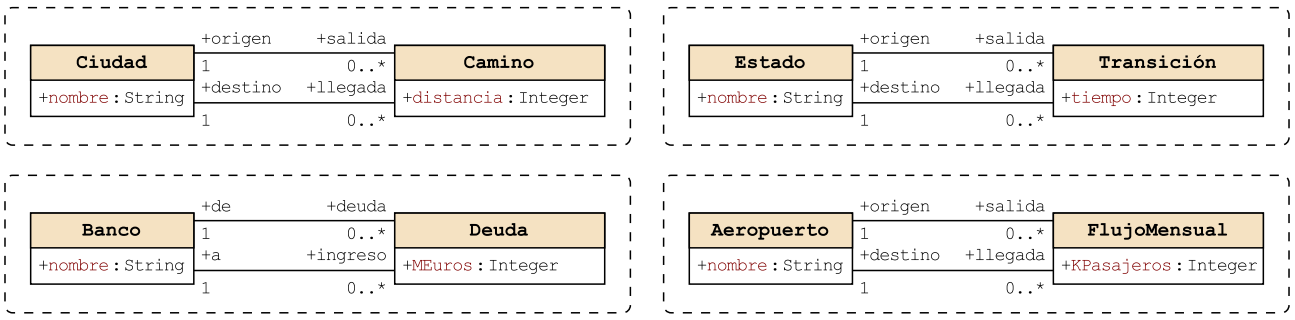
**1)** Distancias entre ciudades, expresadas en kilómetros.

**2)** Transiciones entre estados de un diagrama de estados, en donde los números indican el tiempo en milisegundos que tarda en llevarse a cabo cada transición.

**3)** Flujos de migración mensuales entre aeropuertos, expresados en millares de personas.

**4)** Deudas entre entidades bancarias, expresadas en millones de euros.

Y los correspondientes metamodelos son:

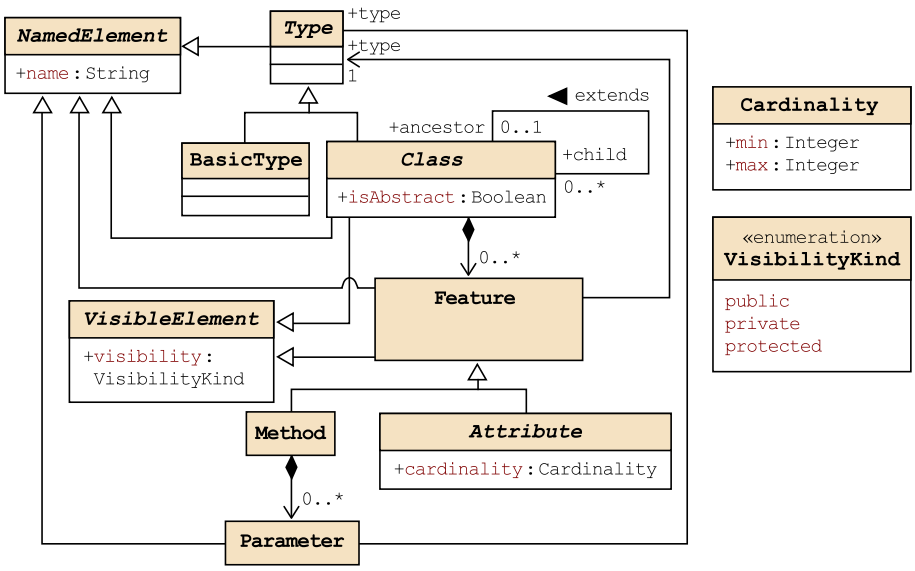



Salvo los nombres de las clases y los atributos, los cuatro metamodelos son los mismos. Lo que va a distinguir a unos de otros es su semántica.

**8.** Según se detalla en el apartado "ATL *matched rules*", el atributo *price* indica el precio de la pieza. Las cabezas de martillo tienen un precio de 5,40 euros, los mangos cuestan 4,35 euros un martillo sin pulir 11,15 euros y uno pulido 14,80 euros.

```
helper context Plant!Part def : getPrice() : Real =
   if (self.oclIsTypeOf(Plant!Head)) then 5.40
   else if (self.oclIsTypeOf(Plant!Handle)) then 4.35
   else if (self.oclIsTypeOf(Plant!Hammer) and not self.isPolished)
        then 11.15 else 14.80 endif
   endif endif;
```

**9.** Un metamodelo para el sublenguaje de Java de la actividad podría ser el siguiente:




**10.** La transformación ATL pedida es una transformación *in place*, que solo se tiene que encargar de definir su comportamiento para los elementos que hay que transformar, en este caso los atributos públicos. El resto no es preciso modificarlos y por tanto no hace falta incluirlos en la especificación de la transformación.

```
module Public2Private;
create OUT : Java refining IN : Java;
helper context String def: firstToUpper() : String =
   self.substring(1,1).toUpper() + self.substring(2, self.size());
rule PublicAttribute {
   from  s : Java!Attribute ( s.visibility = #public )
   to    t : Java!Attribute ( -- change visibility to #private
            visibility <- #private ),
         tg : Java!Method ( -- define public "getter"
            name <- 'get' + s.name.firstToUpper(),
            visibility <- #public,  type <- s.type ),
         ts : Java!Method (-- define public "setter"
            name <- 'set' + s.name.firstToUpper(),
            visibility <- #public,  parameter <- tsp ),
         tsp : Java!Parameter ( -- parameter of "setter".
            name <- 'newValue', type <- s.type )
}
```

**Ejercicios de autoevaluación**

**1.** MDD (*model driven development*) es más específico que MDE (*model-driven engineering*), pues MDD se refiere solo a las actividades de desarrollo de aplicaciones software usando modelos, mientras que MDE engloba también al resto de las actividades de ingeniería del software (mantenimiento, evolución, análisis, etc.). MDA es la propuesta concreta de la OMG para implementar MDD, usando los estándares y lenguajes definidos por esta organización. (Véase el apartado "Terminología")

**2.** Un modelo es una descripción o representación de un sistema. Un metamodelo es un modelo de un lenguaje de modelado, que sirve para describir modelos. MOF y Ecore son ejemplos de lenguajes de metamodelado. UML y BPMN son ejemplos de lenguajes de modelado. (Véase el apartado "Conceptos básicos")

**3.** El patrón MDA está compuesto por un modelo PIM del sistema, que se transforma en un modelo PSM del mismo sistema, usando una transformación de modelos. Dicha transformación puede estar parametrizada por otro modelo adicional, que puede ser por ejemplo el modelo de la plataforma destino, o bien parámetros de configuración del usuario. (Véase el apartado "Los modelos como piezas claves de ingeniería")

**4.** En OCL, la palabra reservada `context` se utiliza para identificar el contexto en el que se define la expresión, es decir, el elemento del modelo que sirve como referencia a la expresión OCL, y desde donde se realizan las navegaciones a otros elementos. La palabra reservada `self` se usa precisamente para referirse a instancias de ese elemento. (Véase el apartado "Restricciones de integridad en OCL")

**5.** En OCL, el operador "^" (hasSent) se utiliza para evaluar si un mensaje o señal ha sido enviado durante la ejecución de un método. El operador "^^" (SentMessages) devuelve el conjunto de mensajes enviados por un objeto durante la ejecución de un método. Ambos operadores solo pueden utilizarse en la especificación de poscondiciones de operaciones (Véase el apartado "Invocación de operaciones y salidas")

**6.** Un DSL consta de sintaxis abstracta, sintaxis concreta y semántica. La primera describe el vocabulario del lenguaje y sus reglas gramaticales; se describe mediante un metamodelo. La sintaxis concreta define la notación con la que se representan los modelos válidos que pueden expresarse con el DSL; se describe mediante una asociación entre los conceptos del DSL y un conjunto de símbolos o palabras (dependiendo de si el DSL es visual o textual). La semántica define el significado de los modelos válidos. Puede describirse de diferentes formas: denotacional, operacional o axiomáticamente. (Véase el apartado "Componentes de un DSL")

**7.** La OMG define tres formas para definir DSLs: 1)directamente a partir de MOF; 2) extendiendo el metamodelo de UML, pero respetando su semántica (extensión "ligera", usando perfiles UML); y 3) modificando el metamodelo de UML sin respetar necesariamente su semántica (extensión "pesada"). (Véase el apartado "Formas de extender UML")

**8.** Un *tag definition* es un atributo de un estereotipo definido en un perfil UML. Dicho atributo puede tomar valores que han de definirse en el momento de aplicar el estereotipo sobre un elemento UML. A dichos valores se les denomina *tag values*. (Véase la sección "Los perfiles UML")

**9.** Una transformación de modelos *in-place* es un tipo particular de transformación en la que los metamodelos origen y destino son el mismo. (Véase el apartado "Tipos de transformaciones")

**10.** Una *matched rule* es una regla ATL que define un patrón y cómo han de transformarse los elementos del modelo origen que sean conformes a ese patrón en elementos del modelo destino. Se ejecutan cada vez que haya una coincidencia de elementos en el modelo origen con el patrón que especifica la regla. Una *lazy rule* es un tipo particular de *matched rule* que solo se ejecuta cuando es invocada desde otra regla ATL. Finalmente una *unique lazy rule* es un tipo particular de *lazy rule* que no genera nuevos elementos en el modelo destino cada vez que se ejecuta al ser invocada, sino que a partir de su segunda invocación siempre devuelven los mismos elementos del modelo destino para los mismos parámetros de entrada. (Véase el apartado "ATL: Atlas Transformation Language")

# Glosario

**ADM (*architecture-driven modernization*)** *m* Propuesta de la OMG para implementar prácticas de ingeniería inversa, usando modelos.

**ATL (*atlas transformation language*)** *m* Lenguaje de transformación de modelos definido por el grupo Atlanmod del INRIA, de amplia adopción.

**BPM (*business process modeling*)** *m* Rama del Model-Based Engineering que se centra en el modelado de los procesos de negocio de una empresa u organización, de forma independiente de las plataformas y las tecnologías utilizadas.

**BPMN (*business process modelling notation*)** *m* Lenguaje de modelado de procesos de negocio de la OMG.

**DSL (*domain specific language*)** *m* Lenguaje de modelado que proporciona los conceptos, notaciones y mecanismos propios de un dominio en cuestión, semejantes a los que manejan los expertos de ese dominio, y que permite expresar los modelos del sistema a un nivel de abstracción adecuado.

**LED (lenguaje específico de dominio)** Véase **DSL**.

**M2M (*model-to-model*)** *m* Tipo particular de transformaciones en las que el origen y el destino son modelos.

**M2T (*model-to-text*)** *m* Tipo particular de transformaciones en las que el origen es un modelo y el destino es un texto (por ejemplo, un trozo de código).

**MBE (*model-based engineering*)** *m* Término general que engloba los enfoques dentro de la ingeniería del software que usan modelos en alguno de sus procesos o actividades: MDE, MDA, MDD, etc. (Véase la figura 7).

**MDA (*model-driven architecture*)** *m* Propuesta concreta de la OMG para implementar MDD, usando las notaciones, mecanismos y herramientas estándares definidos por esa organización (MOF, UML, OCL, QVT, XMI, etc.).

**MDD (*model-driven development*)** *m* Paradigma de desarrollo de software que utiliza modelos para diseñar los sistemas a distintos niveles de abstracción, y secuencias de transformaciones de modelos para generar unos modelos a partir de otros hasta generar el código final de las aplicaciones en las plataformas destino.

**MDE (*model-driven engineering*)** *m* Paradigma dentro de la ingeniería del software que aboga por el uso de los modelos y las transformaciones entre ellas como piezas claves para dirigir todas las actividades relacionadas con la ingeniería del software.

**MDI (*model-driven interoperability*)** *m* Iniciativa para implementar mecanismos de interoperabilidad entre servicios, aplicaciones y sistemas usando modelos y técnicas de MBE.

**metamodelo** *m* Modelo que especifica los conceptos de un lenguaje, las relaciones entre ellos y las reglas estructurales que restringen los posibles elementos de los modelos válidos, así como aquellas combinaciones entre elementos que respetan las reglas semánticas del dominio.

**modelo (de un <x>)** *m* Especificación o descripción de un <x> desde un determinado punto de vista, expresado en un lenguaje bien definido, y con un propósito determinado. En esta definición, <x> puede referirse a un sistema existente o imaginario, un lenguaje, un artefacto software, etc.

**MOF (*meta-object facility*)** *m* Lenguaje de metamodelado definido por la OMG.

**OCL (*object constraint language*)** *m* Lenguaje textual de especificación de restricciones sobre modelos, definido por la OMG.

**OMG (*object management group*)** *m* Consorcio para la estandarización de lenguages, modelos y sistemas para el desarrollo de aplicaciones distribuidas y su interoperabilidad.

**perfil UML** *m* Extensión de un subconjunto de UML orientado a un dominio, utilizando estereotipos, valores etiquetados y restricciones.

**PIM (*platform independent model*)** *m* Modelo de un sistema que concreta sus requisitos funcionales en términos de conceptos del dominio y que es independiente de cualquier plataforma.

**plataforma** *f* Conjunto de subsistemas y tecnologías que describen la funcionalidad de una aplicación sobre la que se van a construir sistemas, a través de interfaces y patrones específicos.

**PSM (*platform specific model*)** *m* Modelo de un sistema resultado de refinar un modelo PIM para adaptarlo a los servicios y mecanismos ofrecidos por una plataforma concreta.

**QVT (*query-view-transformation*)** *m* Lenguaje de transformación de modelos definido por la OMG.

**semántica (de un DSL)** *f* Especificación del significado de los modelos válidos que pueden representarse con un DSL.

**sintaxis abstracta (de un DSL)** *f* Describe el vocabulario con los conceptos del lenguaje, las relaciones entre ellos, y las reglas que permiten construir las sentencias (programas, instrucciones, expresiones o modelos) válidas del lenguaje.

**sintaxis concreta (de un DSL)** *f* Define la notación que se usa para representar los modelos que pueden describirse con ese lenguaje.

**SPEM (*systems process engineering metamodel*)** *m* Lenguaje de modelado de procesos de negocio de la OMG.

**transformación de modelos** *f* Proceso de convertir un modelo de un sistema en un modelo del mismo sistema. Asimismo, dícese de la especificación de dicho proceso.

**UML (*unified modeling language*)** *m* Lenguaje de modelado de propósito general definido por la OMG.

**XMI (XML *metadata interchange*)** *m* Estándar de la OMG para el intercambio de metadatos.

# Bibliografía


### Bibliografía básica

**Clark, T.; Sammut, P.; Willans, J.** (2004). *Applied Metamodeling: A Foundation for Language Driven Development* (2.ª ed.). Ceteva. http://itcentre.tvu.ac.uk/~clark/docs/Applied%20Metamodelling%20%28Second%20Edition%29.pdf.

**Warmer, J.; Kleppe, A.** (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*. (2.ª ed.). Addison-Wesley.

### Bibliografía adicional

**Cook, S.; Jones, S.; Kent, S.; Wills, A. C.** (2007). *Domain-Specific Development with Visual Studio DSL Tools*. Willey.

**Fowler, M.** (2011). *Domain Specific Languages*. Addison-Wesley.

**Kelly, S.; Tolvanen, J. P.** (2008). *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press.

**Kleppe, A.** (2008). *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley.

### Referencias bibliográficas

**Bézivin, J.** (2005). "The Unification Power of Models". *SoSym* (vol. 2, núm. 4, págs. 171-188).

**Jouault, F.; Allilaire, J.; Bézivin, I.; Kurtev** (2008). "ATL: a model transformation tool". *Science of Computer Programming* (vol. 1-2, núm. 72, págs. 31-39).

**Mernik, M.; Heering, J.; Sloane, A.** (2005). "When and How to Develop Domain-Specific Languages". *ACM Computing Surveys* (vol. 4, núm. 37, págs. 316-344).

**Sendall, S.; Kozaczynski, W.** (2003). "Model Transformations: The heart and soul of Model-driven Software Development". *Proceedings of IEEE Software* (vol. 5, núm. 20, págs. 42-50). Boston.

**Strembeck, M.; Zdun, U.** (2009). "An Approach for the Systematic Development of Domain-Specific Languages". *Software: Practice and Experience* (vol. 15, núm. 39). SP&E.

### Enlaces recomendados

**OMG** (2001). Model Driven Architecture -- A technical perspective. OMG document: ormsc/2001-07-0.

**OMG** (2003). MDA Guide V1.0.1. OMG document: omg/03-06-01.

**OMG** (2004). Human-Usable Textual Notation (HUTN) Specification. OMG document: formal/04-08-01.

**OMG** (2008). MOF QVT Final Adopted Specification. OMG document: formal/08-04-03.

**OMG** (2010). UML 2.3.1 Superstructure specification. OMG document: formal/2010-05-05.

**OMG** (2012). OCL 2.3.1. OMG document: formal/2012-01-01.