Facultad de Informática de la
Universidad de Murcia

PROYECTO INFORMÁTICO

Ingeniería de Modelos con MDA
Estudio comparativo de OptimalJ y ArcStyler

Alumno
Jesús Rodríguez Vicente
jrv1@alu.um.es

Director
Jesús J. García Molina
jmolina@um.es

Departamento de Informática y Sistemas

Junio de 2004

Indice

Índice de Figuras

1 Introducción

1.1 Contexto

El *Object Management Group* (OMG) es un consorcio de empresas de informática creado en el año 1990, con el objetivo de potenciar el desarrollo de aplicaciones orientadas a objetos distribuidas. Para ello, desde un principio se prestó especial atención al problema de la **iteroperabilidad e integración** de sistemas software, lo que ha llevado al OMG a definir numerosas especificaciones y estándares como CORBA, UML, MOF, XMI y CWM.

En el año 2001, el OMG estableció el **framework MDA** como arquitectura para el desarrollo de aplicaciones. Mientras que el anterior framework propuesto por OMG, formado por las especificaciones OMA y CORBA, estaba destinado al desarrollo de aplicaciones distribuidas, MDA representa un nuevo paradigma de desarrollo de software en el que los modelos guían todo el proceso de desarrollo. Este nuevo paradigma se ha denominado **Ingeniería de modelos** o **Desarrollo basado en modelos**.

En la actualidad, la construcción de software se enfrenta a continuos cambios en las tecnologías de implementación, lo que implica realizar esfuerzos importantes tanto en el diseño de la aplicación, para integrar las diferentes tecnologías que incorpora, como en el mantenimiento, para adaptar la aplicación a cambios en los requisitos y en las tecnologías de implementación. Por otra parte, las aplicaciones distribuidas *Business to Business* (B2B) y *Client to Business* (C2B) son cada vez más comunes, siendo difícil satisfacer los requisitos de escalabilidad, seguridad y eficiencia. La idea clave que subyace a MDA es que si el desarrollo está guiado por los modelos del software, se obtendrán beneficios importantes en aspectos fundamentales como son la **productividad**, la **portabilidad**, la **interoperabilidad** y el **mantenimiento**.

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

A lo largo de la historia del desarrollo de software, se ha evolucionado desde los lenguajes ensambladores de los años 50 hasta los lenguajes orientados a objetos y de cuarta generación (4GLs) de nuestros días. La ingeniería de modelos es considerada como un nuevo paso en el camino hacia lenguajes de programación más expresivos y hacia una mayor automatización.

En los dos últimos años se han creado diferentes entornos de desarrollo basados en MDA, así como herramientas de transformación de modelos. *Optimal*J de *Compuware* y *ArcStyler* de *Interactive Objects* son reconocidas como las dos herramientas MDA más

maduras. El mercado de herramientas MDA es actualmente muy activo, pero deben pasar todavía algunos años para explotar todas las posibilidades de este nuevo paradigma.

1.2 Objetivo del Proyecto

El objetivo de este proyecto es realizar un análisis comparativo de las dos herramientas MDA más extendidas: *OptimalJ* y *ArcStyler*.

Por otro lado, en la actualidad no hay disponibles muchos libros sobre MDA, y ninguno en español, por lo que otro objetivo es elaborar un documento que sirva como introducción práctica a MDA y a las dos herramientas mencionadas.

1.3 Método de Trabajo

En este apartado comentaremos la distribución de tiempo y el orden en que se ha elaborado cada parte del proyecto. Cabe destacar que la distribución de tiempo es aproximada, y que cada semana mencionada aquí equivale a unas 20 horas de dedicación.

Durante las primeras cuatro semanas del proyecto, recopilamos toda la información posible acerca de MDA, principalmente de la página de MDA del OMG (http://www.omg.org/mda/). En ese tiempo pudimos leer numerosos artículos relacionados [2, 22, 24, 26, 28, 31] y la guía oficial de MDA [21]. Todos estos documentos dan una visión muy teórica de MDA, encontrando escasos ejemplos prácticos.

Entonces tuvimos acceso al libro *MDA Explained* [19], donde se explica este framework de forma mucho más práctica y sencilla. Durante las tres semanas siguientes leímos en detalle dicho libro y comenzamos a documentar nuestra introducción a MDA, lo que nos llevó cerca de cuatro semanas. En ese tiempo también definimos la aplicación *Pet Store* que hemos usado como ejemplo de sistema software a desarrollar.

En el libro *MDA Explained* se utiliza la herramienta OptimalJ como herramienta MDA de ejemplo, así que la siguiente semana descargamos e instalamos la versión *OptimalJ Demo Edition 2.2*, observando por primera vez un entorno de desarrollo MDA.

Más tarde conseguimos la versión *OptimalJ Professional Edition 3.0* [3] a través de la firma de un convenio con la empresa *Compuware*, siendo la primera universidad española que lo hacía. Tardamos cerca de una semana en completar los tutoriales [5] y familiarizarnos con la herramienta. Tras esto, estuvimos aproximadamente tres semanas elaborando el caso práctico *Pet Store* [29] con OptimalJ, intentando "explotar" al máximo las posibilidades de la herramienta.

Mientras recopilábamos información para evaluar la herramienta tuvimos conocimiento del informe del *King's College London* [20] donde se evalúa la misma herramienta, sorprendentemente también con el mismo ejemplo del *Pet Store*. Dada la calidad y cantidad de criterios de evaluación para herramientas MDA allí expuestos, decidimos

usarlos también para nuestro estudio comparativo, realizando algunas modificaciones. A continuación elaboramos durante tres semanas el resumen de la arquitectura herramienta de OptimalJ y su evaluación.

La última parte la dedicamos a evaluar la herramienta ArcStyler [12]. Familiarizarnos con la herramienta nos llevó cerca de cuatro semanas, pues tuvimos que consultar numerosos manuales y tutoriales [13, 14, 15, 16, 17] y realizar muchas pruebas previas. Tras esto, en cerca de cinco semanas elaboramos la aplicación *Pet Store* con ArcStyler.

En las siguientes dos semanas realizamos la documentación y evaluación de la herramienta. Las últimas dos semanas las dedicamos a corregir algunos errores y a completar la documentación del proyecto.

1.4 Organización del Documento

En este **capítulo 1** presentamos el contexto y los objetivos del proyecto.

El **capítulo 2** realiza una introducción a MDA, presentando los aspectos más importantes de este nuevo paradigma y comparándolo con el desarrollo de software tradicional.

El **capítulo 3** profundiza en los fundamentos que hacen posible MDA, como *MOF*, *OCL* o los *UML Profiles*.

A continuación, en el **capítulo 4** hablaremos en detalle de las transformaciones de modelos, haciendo hincapié en las propiedades que deben tener dichas transformaciones y explicando la propuesta de *QVT-Partners* para un lenguaje de definición de transformaciones. También haremos un repaso de los distintos tipos de herramientas de transformación y de otras herramientas importantes para dar soporte completo a MDA.

En el **capítulo 5** se describen los criterios de evaluación y el ejemplo de aplicación usado para evaluar las herramientas OptimalJ y ArcStyler.

En el **capítulo 6** realizamos un repaso de la herramienta OptimalJ. Para ello, resumimos su arquitectura y evaluamos la herramienta según los criterios presentados en el capítulo 5. También explicaremos brevemente cómo construir una aplicación básica con OptimalJ y mostraremos la interfaz de usuario de nuestra aplicación de ejemplo creada con la herramienta.

En el **capítulo 7** haremos lo mismo que en el capítulo 6 para la herramienta ArcStyler, mostrando su arquitectura, los pasos principales para la construicción de nuestra aplicación, la interfaz generada y la evaluación de la herramienta.

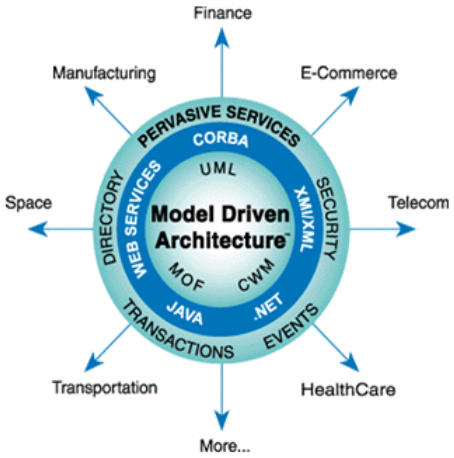
En el **capítulo 8** se lleva a cabo un completo estudio comparativo de ambas herramientas, donde se evalúan frente a frente los aspectos más relevantes estas dos herramientas MDA.

Por último, el **capítulo 9** presentará las conclusiones finales del proyecto y las vías futuras de investigación.

2 Presentación de MDA

2.1 Introducción

Según el OMG, MDA proporciona una solución para los cambios de negocio y de tecnología, permitiendo construir aplicaciones independientes de la plataforma e implementarlas en plataformas como CORBA, J2EE o Servicios Web. La Figura 1, obtenida de la página de MDA del OMG[1], pretende resumir lo más importante de este nuevo framework, mencionando los estándares relacionados y las principales ventajas de la aplicación de MDA.



How Systems Will Be Built

MDA® provides an open, vendor-neutral approach to the challenge of business and technology change. Based firmly upon OMG's established standards*, MDA aims to separate business or application logic from underlying platform technology. Platform-independent applications built using MDA and associated standards can be realized on a range of open and proprietary platforms, including CORBA®, J2EE, .NET, and Web Services or other Web-based platforms. Fully-specified platform-independent models (including behavior) can enable intellectual property to move away from technology-specific code, helping to insulate business applications from technology evolution, and further enable interoperability. In addition, business applications, freed from technology specifics, will be more able to evolve at the different pace of business evolution.

Figura 1. Logotipo de MDA y breve explicación

Pero la Figura 1 apenas explica nada acerca de en qué consiste esta nueva arquitectura. ¿Qué es en realidad *Model Driven Architecture*[2] o MDA? Se trata de un framework de desarrollo de software que define una nueva forma de construir software en la que se usan modelos del sistema, a distintos niveles de abstracción, para guiar todo el proceso de desarrollo, desde el análisis y diseño hasta el mantenimiento del sistema y su integración con futuros sistemas.

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

- definir un sistema independientemente de la plataforma sobre la que se construye,
- definir plataformas sobre las que construir los sistemas,
- elegir una plataforma particular para el sistema, y
- transformar la especificación inicial del sistema a la plataforma elegida.

Los principales objetivos de MDA son mejorar la **productividad**, la **portabilidad**, la **interoperabilidad** y la **reutilizacion** de los sistemas.

[1] http://www.omg.org/mda/
[2] En castellano, *Arquitectura Dirigida por Modelos*

A grandes rasgos, el proceso de desarrollo de software con MDA se puede dividir en tres fases:

- **Construcción de un *Modelo Independiente de la Plataforma*** (*Platform Independent Model* o *PIM*), un modelo de alto nivel del sistema independiente de cualquier tecnología o plataforma.

- **Transformación del modelo anterior a uno o varios *Modelos Específicos de Plataforma*** (*Platform Specific Model* o *PSM*). Un PSM es un modelo de más bajo nivel que el PIM que describe el sistema de acuerdo con una tecnología de implementación determinada.

- **Generación de código a partir de cada PSM**. Debido a que cada PSM está muy ligado a una tecnología concreta, la transformación de cada PSM a código puede automatizarse.

El paso de PIM a PSM y de PSM a código no se realiza "a mano", sino que se usan herramientas de transformación para automatizar estas tareas. Todos estos aspectos se explicarán más detalladamente a lo largo de este documento. La Figura 2 muestra de manera resumida el proceso de desarrollo con MDA, suponiendo que tenemos un único PSM.

Figura 2. Pasos en el desarrollo con MDA

2.2 Modelos en MDA

Introducción a los modelos

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

UML es un lenguaje de modelado bien definido que se ha adoptado como el principal lenguaje de modelado en MDA. Decimos el "principal lenguaje de modelado" porque **MDA no está restringido a UML**, sino que se puede usar cualquier lenguaje bien definido. No obstante, la realidad nos muestra que UML se ha convertido en el lenguaje de modelado "oficial" de MDA, así que de aquí en adelante se asumirá que todos los modelos se construyen usando este lenguaje.

De entre los distintos tipos de modelos definidos en UML, los ***Modelos de Clases***, que muestran la vista estática del sistema, son los más importantes dentro de MDA, ya que el PIM y la mayoría de PSMs son modelos de este tipo. Estos modelos se representan mediante ***Diagramas de Clases*** de UML.

Ejemplo:

La Figura 3 muestra un diagrama de clases, que representa el modelo de clases de un sistema de venta de animales.

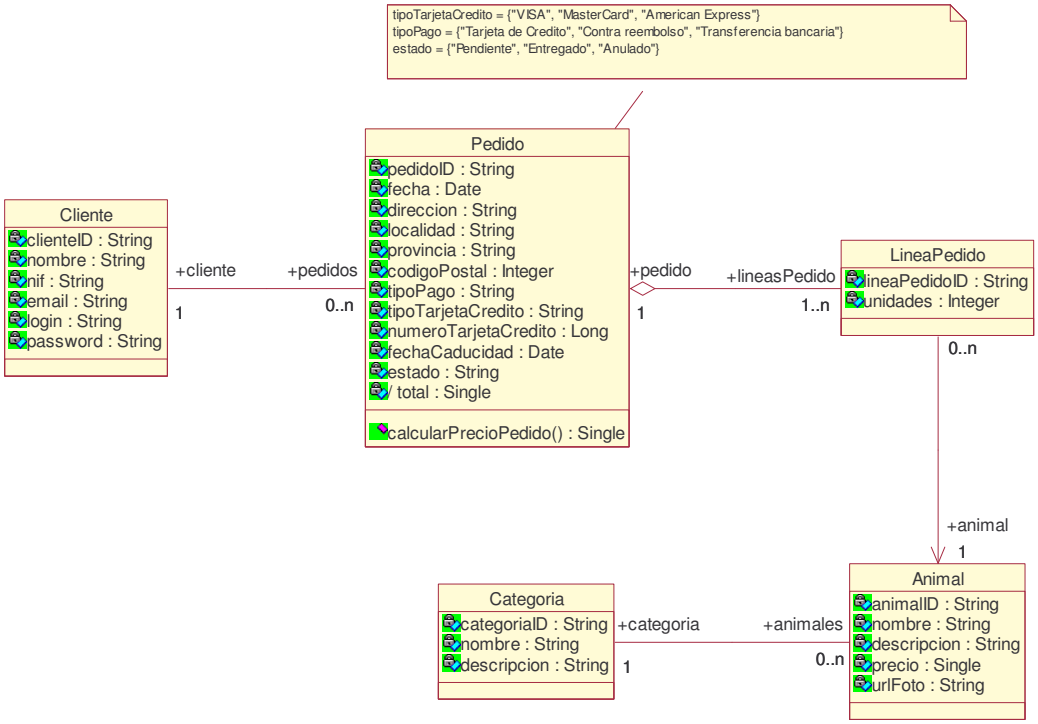


Figura 3. Un diagrama de clases UML

PIM (Platform Independent Model)[3]

Un ***Modelo Independiente de Plataforma*** o ***PIM*** es un modelo del sistema de alto nivel que representa la estructura, funcionalidad y restricciones del sistema sin aludir a una plataforma determinada. Este modelo servirá de base para todo el proceso de desarrollo, y es el único que debe ser creado íntegramente por el desarrollador.

Al no incluir detalles específicos de una tecnología determinada, este modelo es útil en dos aspectos:

- Es fácilmente comprensible por los usuarios del sistema, y por lo tanto, les resultará más sencillo validar la corrección del sistema.

[3] En la guía oficial de MDA [21] se habla también de un *Modelo Independiente de Computación* (CIM), como modelo previo al PIM que describe el sistema del negocio. No obstante, este modelo es opcional y a nuestro entender no tiene mucho interés en el proceso de desarrollo con MDA.

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

La Figura 4 muestra un ejemplo de un PIM sencillo con tres clases interrelacionadas. El diagrama de clases mostrado en la Figura 3 también es un ejemplo de PIM.

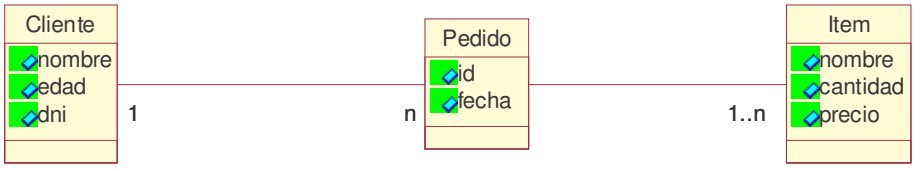


Figura 4. Ejemplo de PIM

Como vemos, el PIM se modela mediante el **diagrama de clases de UML**.

PSM (Platform Specific Model)

Un ***Modelo Específico de Plataforma*** o ***PSM*** es un modelo del sistema con detalles específicos de la plataforma en la que será implementado. Se genera a partir del PIM, así que representa el mismo sistema pero a distinto nivel de abstracción. Podemos decir que un PSM es un PIM al que se le añaden detalles específicos para ser implementado en una plataforma determinada.

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

La Figura 5 muestra un PSM construido a partir del PIM de la Figura 4, representado también mediante un diagrama de clases UML. Este sencillo ejemplo muestra un posible PSM para la plataforma EJB[4]. Como vemos, en el paso del PIM al PSM se han producido varias transformaciones:

- Se ha añadido el estereotipo `<<EJBEntity>>` a cada clase, para indicar que la clase representa un EJB de tipo *Entity*.
- Se ha modificado la visibilidad de los atributos del PIM de público a privado.
- Se han generado métodos públicos de lectura y modificación (*get* y *set*) para cada atributo.

Para la construcción de PSMs se usan los ***Perfiles UML*** (*UML Profiles*), que son extensiones de UML que permiten añadir información semántica a los modelos para expresar detalles específicos de la plataforma. Por ejemplo, el estereotipo `<<EJBEntity>>` de las clases de la Figura 5 forma parte del *Perfil UML* para la plataforma *EJB*. De los *Perfiles UML* hablaremos con más detalle en la sección Perfiles UML (pág. 20).

[4] *Enterprise Java Beans*. Para más información sobre esta plataforma, consúltese [7]

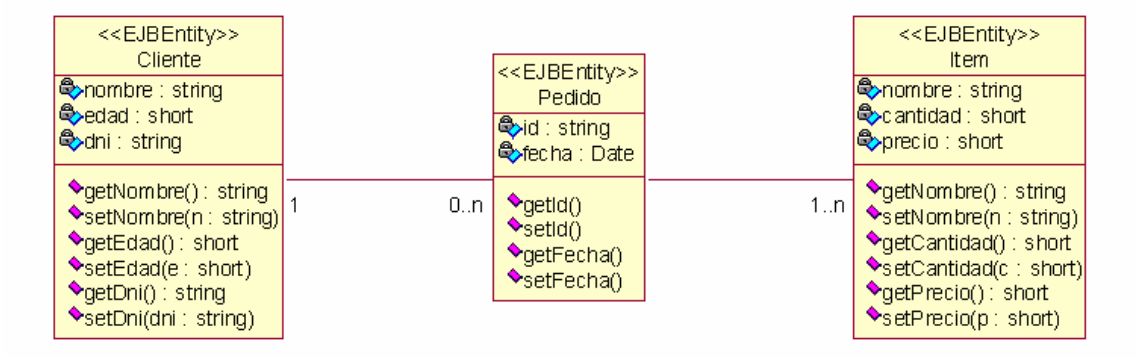


Figura 5. Ejemplo de PSM

Hay que destacar que a partir de un mismo PIM pueden generarse **varios PSMs**, cada uno describiendo el sistema desde una perspectiva diferente. Podemos ver un ejemplo de esto más adelante en el apartado Generación de puentes de comunicación. Ejemplo de desarrollo con varios PSMs.

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

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

Para las transformaciones automáticas se usan herramientas especializadas. Estas herramientas tienen implementados distintos algoritmos de transformación para pasar de un tipo de modelo a otro, y suponen uno de los pilares de MDA. Las herramientas de transformación de modelos se verán con detalle en el apartado Herramientas de Transformación (pág. 26).

Un PSM también puede refinarse, transformándose sucesivamente en PSMs de más bajo nivel[5], hasta llegar al punto en que pueda ser transformado a código de manera directa.

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

En el apartado Transformaciones de Modelos (pág. 23), explicaremos con más detalle las transformaciones de MDA.

[5] El PSM inicial asumiría el papel de PIM del PSM "refinado".

Generación de puentes de comunicación. Ejemplo de desarrollo con varios PSMs

En la Figura 2 vimos un esquema sencillo mostrando las fases de desarrollo con MDA. Un esquema más general sería aquel en el que tenemos varios PSMs derivados del mismo PIM, representando cada PSM una parte del sistema. En ese caso, además de generar los PSMs a partir del PIM y el código a partir de cada PSM, también deberían generarse los **puentes de comunicación** entre las distintas partes, tanto a nivel de PSM como de código. El siguiente ejemplo puede aclarar esta idea.

Ejemplo:

Un esquema típico de desarrollo con MDA usando varios PSMs es el que aparece en la Figura 6*. A partir del PIM se generan tres PSMs:*

- *Un modelo relacional del sistema, que describe el esquema de base de datos del sistema mediante un diagrama de Entidad-Relación.*
- *Un modelo EJB, mostrando los aspectos relativos a la plataforma EJB.*
- *Un modelo Web para describir las interfaces Web del sistema.*

Los puentes de comunicación entre PSMs y código permitirían a la capa Web comunicarse con los componentes EJB, y a estos comunicarse con la base de datos del sistema[6]*.*

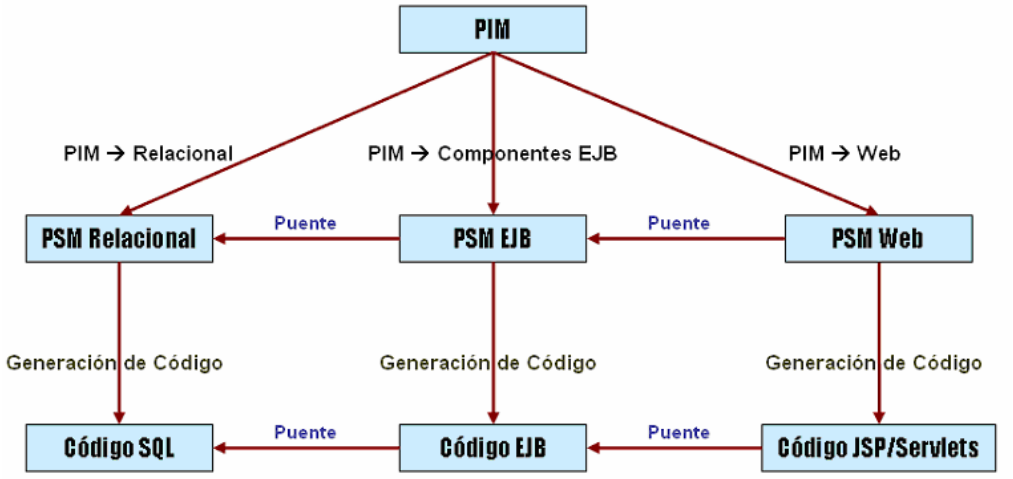


Figura 6. Ejemplo de desarrollo en MDA con varios PSMs

2.3 Desarrollo tradicional vs. Desarrollo con MDA

En el capítulo 1 de *MDA Explained* [19] se realiza una interesante comparativa entre el desarrollo tradicional y el desarrollo con MDA, ilustrando de este modo los beneficios

[6] El esquema de la Figura 6 (modelo de base de datos + modelo EJB + modelo Web) es el que seguiremos para probar las herramientas MDA que se evalúan en este documento, como veremos más adelante.

de este nuevo proceso de desarrollo. La mayor parte de esta sección está extraída de dicho capítulo.

Problemas del desarrollo tradicional

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

1. Recogida de requisitos
2. Análisis
3. Diseño
4. Codificación
5. Prueba
6. Despliegue

La Figura 7 esquematiza el proceso de desarrollo de software tradicional.

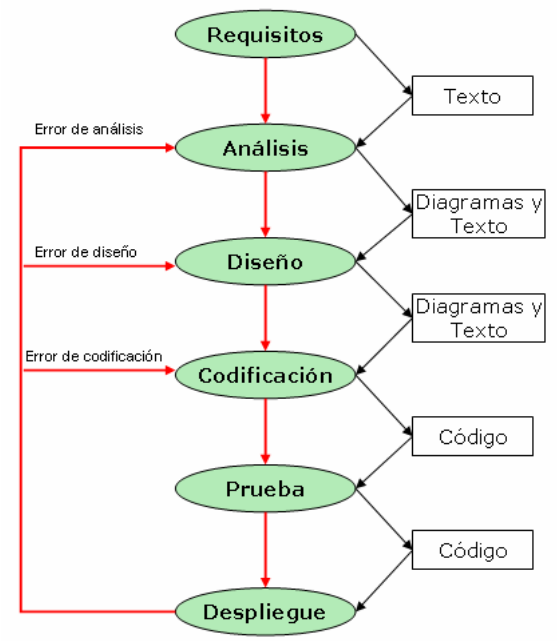


Figura 7. Proceso de desarrollo de software tradicional

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

- **Productividad**. El proceso tradicional produce una gran cantidad de documentos y diagramas para especificar requisitos, clases, colaboraciones, etc. La mayoría de este material pierde su valor en cuanto comienza la fase de codificación, y gradualmente se va perdiendo la relación entre los diagramas. Y más aún cuando el sistema cambia a lo largo del tiempo: realizar los cambios en todas las fases (requisitos, análisis, diseño…) se hace inmanejable, así que generalmente se realizan las modificaciones sólo en el código. ¿Entonces para qué perder el tiempo en construir los diagramas y la documentación de alto nivel? Lo cierto es que para sistemas complejos sigue siendo necesario. Lo que necesitamos

entonces es un soporte para que un cambio en cualquiera de las fases se traslade fácilmente al resto.

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

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

- **Mantenimiento y documentación**. Documentar un proyecto software es una tarea lenta que consume mucho tiempo, y que en realidad no interesa tanto a los que desarrollan el software, sino a aquellos que lo modificarán o lo usarán más adelante. Esto hace que se ponga poco empeño en la documentación y que generalmente no tenga una buena calidad. La solución a este problema a nivel de código es que la documentación se genere directamente del código fuente, asegurándonos que esté siempre actualizada[7]. No obstante, la documentación de alto nivel (diagramas y texto) todavía debe ser mantenida a mano.

Estos problemas se solucionan en MDA, como veremos en el siguiente apartado.

Beneficios del MDA

A continuación explicaremos cómo MDA resuelve cada uno de los problemas del desarrollo tradicional expuestos en el apartado anterior:

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

[7] Un ejemplo de esto es la herramienta *Javadoc* para Java.

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

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

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

El nuevo proceso de desarrollo

Si comparamos el proceso MDA con el proceso tradicional de desarrollo de software, vemos las fases de **recogida de requisitos**, de **prueba** y de **despliegue** seguirán igual.

Lo que cambia son las fases de **análisis**, **diseño** y **codificación**, de la siguiente manera:

- **Análisis**: un grupo especial de personas desarrollarán el PIM, guiados por las necesidades del negocio y la funcionalidad que debe incorporar el sistema.

- **Diseño**: otro grupo diferente de personas se encargarán de la transformación del PIM a uno o más PSMs. Estas personas tendrán conocimientos sobre distintas plataformas y arquitecturas, y conocerán también las transformaciones disponibles en las herramientas que usan. De este modo podrán elegir la plataforma o arquitectura que mejor se adapte a los requisitos del sistema y establecer los parámetros de las distintas transformaciones. Los creadores de PSMs tendrán comunicación constante con los diseñadores del PIM para tener más información sobre el sistema (p.e., para conocer los requisitos no funcionales).

- **Codificación**: esta fase se reduce a generar el código del sistema mediante herramientas especializadas. Los programadores únicamente tendrán que

añadir la funcionalidad que no puede reflejarse en los modelos y, si es necesario, "retocar" el código generado. Según los expertos, en un futuro desaparecerá esta fase de codificación dentro de MDA, y se pasará directamente del PSM a la fase de prueba.

Pero en este nuevo proceso de desarrollo aún falta un tercer grupo de personas, aquellos que escriben **definiciones de transformaciones** para empresas de construcción de software o, principalmente, para los vendedores de herramientas de MDA. Estas transformaciones son fundamentales para construir software con MDA, pues permiten a las herramientas pasar de PIM a PSM y de PSM a código.

La Figura 8 muestra un esquema sencillo que ilustra este nuevo proceso de desarrollo con MDA. Vemos como los cambios se centran en el PIM, base de todo el desarrollo, y gracias a las herramientas de transformación estos cambios se trasladan rápidamente al resto de las fases.

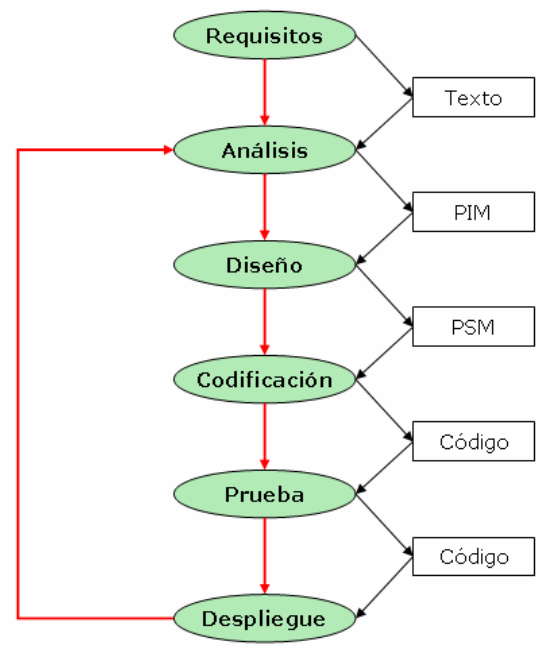


Figura 8. Proceso de desarrollo con MDA

2.4 Visión Alternativa de MDA

Una visión alternativa y práctica de MDA la encontramos en en un manifiesto [2] elaborado por los principales investigadores de IBM trabajando en MDA, en donde se resume MDA como la combinación de tres ideas complementarias. La Figura 9 muestra estas tres ideas, que resumiremos a continuación:

1. **Representación directa**. Se desplaza el foco del desarrollo del software del dominio de la tecnología hacia las ideas y conceptos del dominio del problema. Reduciendo la distancia semántica entre el dominio del problema y su representación permitimos un mayor

acoplamiento de las soluciones a los problemas, logrando diseños más acertados e incrementando la productividad.

2. **Automatización**. Usar herramientas para mecanizar aquellas facetas del desarrollo del software que no dependen del ingenio humano. MDA incrementa la velocidad de desarrollo y reduce errores usando herramientas automatizadas para transformar modelos específicos de plataforma en código de implementación. Es lo mismo que hacen los compiladores para los lenguajes de programación tradicionales.

3. **Estándares abiertos**. Los estándares han sido uno de los mejores impulsores del progreso en la historia de la tecnología. Los estándares de la industria no solo ayudan a eliminar la diversidad gratuita, sino que también alentan a los vendedores a producir herramientas tanto para propósitos generales como para propósitos especializados, incrementando así las posibilidades del usuario. El desarrollo de código abierto asegura que los estándares se implementan consistentemente y anima la adopción de estándares por parte de los vendedores.

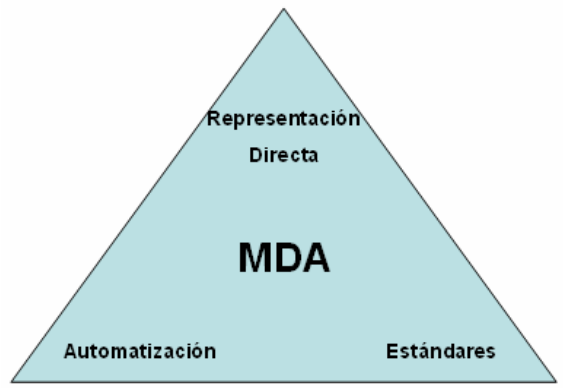


Figura 9. Visión alternativa de MDA

3 Fundamentos de MDA

3.1 Metamodelado y MOF

En este apartado explicaremos qué es el metamodelado y porqué es relevante en el contexto de MDA.

Introducción

El ***metamodelado*** es un mecanismo que permite definir formalmente lenguajes de modelado. Así, un *metamodelo* de un lenguaje es una definición precisa de sus elementos mediante conceptos y reglas de cierto metalenguaje, necesaria para crear modelos en ese lenguaje. Básicamente se trata de usar modelos para describir otros modelos. Por ejemplo, el metamodelo de UML define los conceptos y reglas que se necesitan para crear modelos UML.

Este concepto se entenderá mejor cuando analicemos en el siguiente apartado la arquitectura de cuatro capas de modelado que define el OMG.

Las cuatro capas de modelado del OMG

El OMG usa una arquitectura de cuatro niveles o capas de modelado para sus estándares. En la terminología del OMG estas capas se llaman **M0**, **M1**, **M2** y **M3**, y las veremos en detalle en las próximas secciones.

Capa M0. Instancias

En el nivel M0 están todas las instancias "reales" del sistema, es decir, los objetos de la aplicación. Aquí no se habla de *clases* ni *atributos*, sino de entidades físicas que existen en el sistema. El siguiente ejemplo nos puede ayudar a entender este concepto.

Ejemplo:

Supongamos un sistema que debe manejar clientes de un videoclub. En la Figura 10 *vemos las entidades que almacenan los datos (nombre, dirección y teléfono) de los clientes David Aguilar y Miguel Martínez. Estas entidades son instancias pertenecientes a la capa M0.*

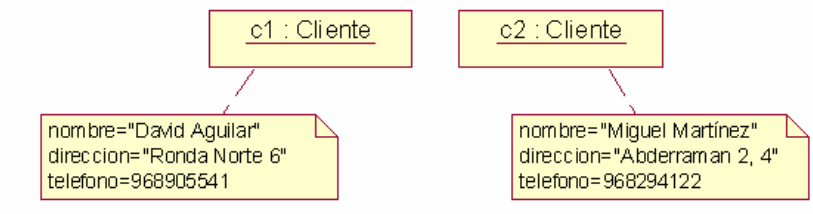


Figura 10. Instancias del sistema (capa M0)

Capa M1. Modelo del Sistema

Por encima de la capa M0 se sitúa la capa M1, que representa el modelo de un sistema software. Los conceptos del nivel M1 representan categorías de las instancias de M0. Es decir, cada elemento de M0 es una instancia de un elemento de M1.

Ejemplo:

Como vemos en la Figura 11, e*n el nivel M1 aparecería la entidad Cliente con los atributos nombre, dirección y teléfono, que representa a todos los clientes de la aplicación. El cliente de nombre David Aguilar es una instancia de esa entidad. La entidad Artículo, que representa a todos los artículos que pueden alquilarse en el videoclub, también pertenece al nivel M1.*

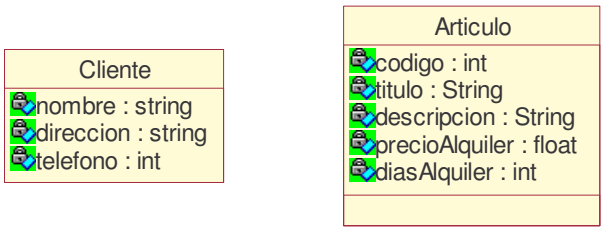


Figura 11. Entidades del modelo del sistema (capa M1)

Los modelos creados usando UML pertenecen a este nivel M1.

Capa M2. Metamodelo

Análogamente a como ocurría con M0 y M1, los elementos del nivel M1 son a su vez instancias del nivel M2. Esta capa recibe el nombre de capa de *metamodelo*. En este nivel aparecerán conceptos como *Clase*, *Atributo* o *Relación*.

Ejemplo:

La entidad Cliente mostrada en el apartado anterior sería una instancia de la metaclase UML Class del metamodelo de UML que aparece en la Figura 12*, junto a la metaclase UML Attribute.*

Figura 12. Entidades del metamodelo de UML (capa M2)

El metamodelo de UML estaría situado en este nivel M2, y gracias a él podemos construir modelos de nivel M1 en UML.

Capa M3. Meta-metamodelo

En la misma línea, podemos ver los elementos de M2 como instancias de otra capa superior M3, la capa de *meta-metamodelo*.

Dentro del OMG, MOF es el lenguaje estándar de la capa M3. Esto supone que todos los metamodelos de la capa M2, por ejemplo, el metamodelo de UML, son instancias de MOF. O lo que es lo mismo, **UML se define usando MOF**. Esto quedará más claro cuando se explique más detalladamente el estándar MOF en la página 18.

Ejemplo:

La Figura 13 *muestra un ejemplo de entidades de la capa M3. La clase de M2 UML Class es una instancia de la clase de M3 MOF Class.*

Figura 13. Entidades de MOF (capa M3)

Una visión completa de las capas

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

La Figura 14 muestra un ejemplo completo donde se aprecia la relación existente entre las capas de modelado definidas por el OMG.

Para terminar, mostramos una tabla resumiendo las distintas capas de modelado:

Capa	Contenido	Ejemplo
M0 – Instancias	Instancias reales del sistema	Cliente con nombre="David Aguilar" y DNI=48491673
M1 – Modelo	Entidades del modelo del sistema	Clase "Cliente" con atributos "nombre" y "DNI"
M2 – Metamodelo	Entidades de un lenguaje de modelado	Entidad "UML Class" del metamodelo de UML
M3 – Meta-metamodelo	Entidades para definir lenguajes de modelado	Entidad "MOF Class" de MOF

Importancia del Metamodelado en MDA

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

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

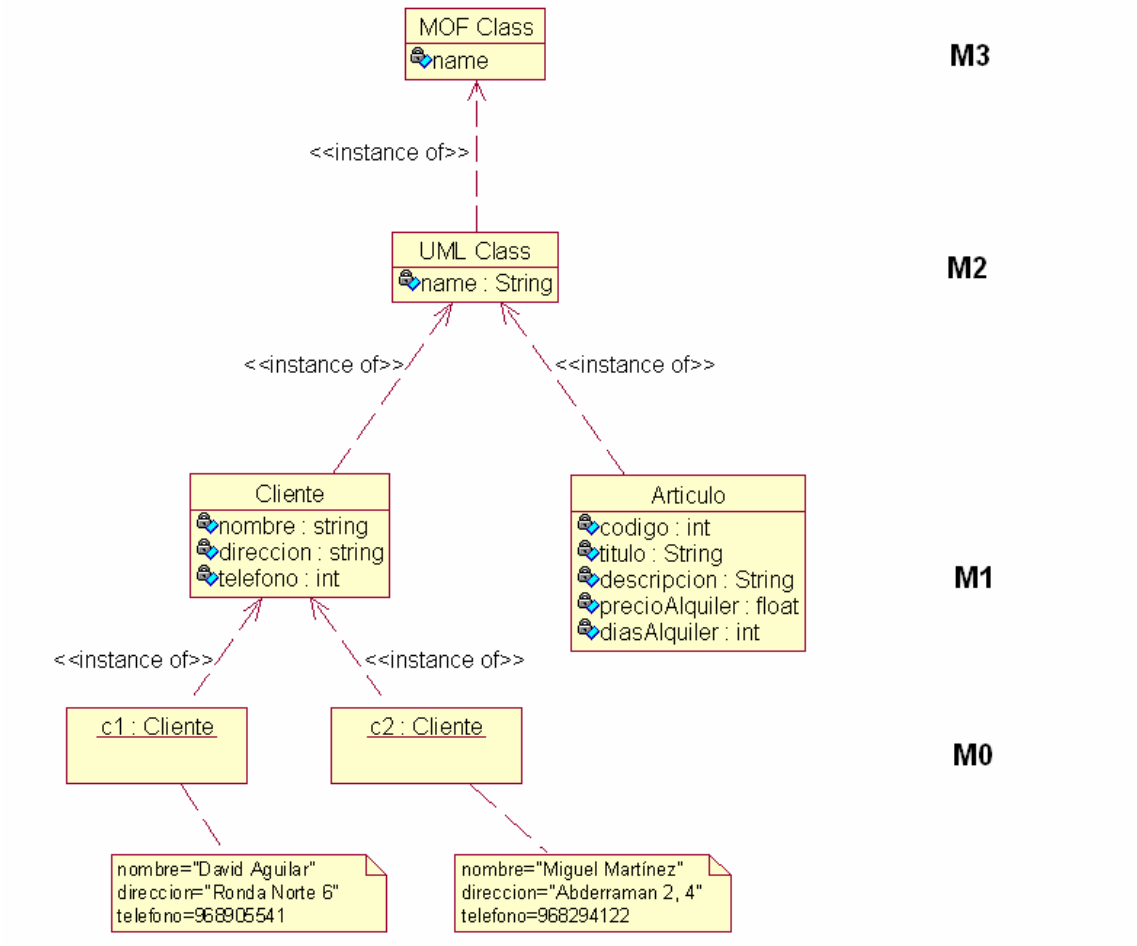


Figura 14. Relación entre las distintas capas de modelado

MOF

Uno de los estándares más importantes usados en MDA es ***Meta Object Facility*** o **MOF**. Se trata de un estándar del OMG que define un **lenguaje común y abstracto para definir lenguajes de modelado**, y cómo acceder e intercambiar modelos expresados en dichos lenguajes. MOF usa cinco construcciones básicas para definir un lenguaje de modelado:

- **Clases**: usadas para definir tipos de elementos en un lenguaje de modelado. Por ejemplo, la relación de dependencia de UML (la llamaremos *UML::Dependencia*) es una clase definida en MOF. Esta clase representa el tipo de todas las dependencias que pueden crearse en un modelo UML.

- **Generalización**: define herencia entre clases. Por ejemplo, *UML::Clasificador* es una generalización de *UML::Clase*. La subclase hereda todas las características de la clase padre.

- **Atributos**: usados para definir propiedades de elementos del modelo. Los atributos tienen un tipo y una multiplicidad. Por ejemplo,

UML::Elemento::nombre es un atributo de tipo *string* que define que cada elemento de un modelo UML tiene un nombre.

- **Asociaciones**: definen relaciones entre clases. Una asociación tiene dos extremos, cada uno de los cuales puede tener definido un nombre de rol, navegabilidad y multiplicidad. Por ejemplo, *UML::Atributo* tiene una asociación con *UML::Clasificador* con nombre *tipo*. Esta asociación se usa para definir el tipo de un atributo definido dentro de una clase UML.

- **Operaciones**: definen operaciones dentro del ámbito de una clase, junto con una lista de parámetros.

Atendiendo a las 4 capas de modelado del OMG, MOF estaría situado en el nivel M3 (nivel de meta-metamodelado)[8]. Mediante MOF puede definirse cualquier lenguaje de modelado, incluido UML. En la Figura 15 puede verse una versión simplificada del metamodelo de MOF.

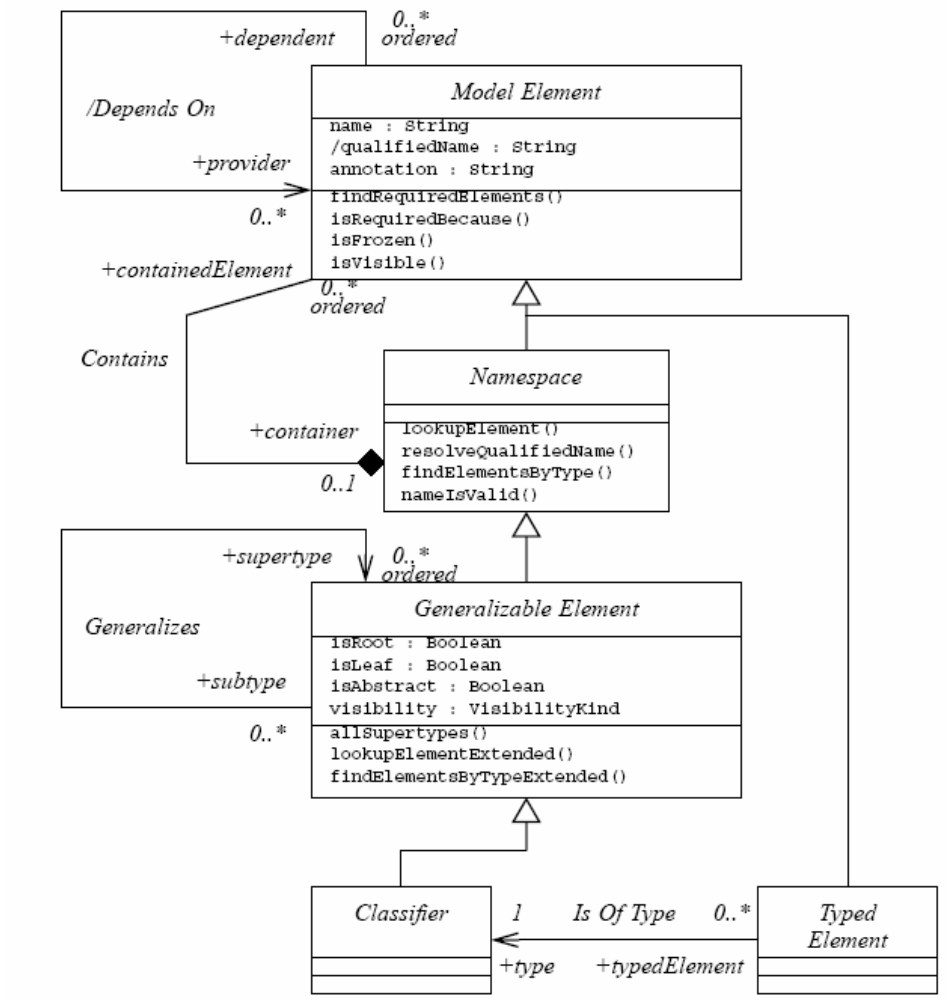


Figura 15. Abstracciones principales de MOF, extraído de [23]

[8] No hay ninguna capa superior a M3, ya que MOF se define usándose a sí mismo.

La definición de MOF también permite la construcción de herramientas para definir lenguajes de modelado, además de diversa funcionalidad adicional:

- ***MOF Repository Interface***. Se trata de un interface que permite obtener información sobre modelos de nivel M1 de un repositorio basado en MOF.

- **Intercambio de Modelos**. MOF también define un formato de intercambio basado en XML para modelos M1, llamado *XMI* (*XML Metadata Interchange*). Ya que MOF se define usándose a sí mismo, puede usarse también XMI para generar formatos de intercambio de metamodelos (nivel M2).

El papel principal de MOF dentro del MDA es proporcionar los conceptos y herramientas para razonar sobre lenguajes de modelado. Gracias a ello podemos definir transformaciones sobre los metamodelos de una manera estándar.

3.2 Perfiles UML

Un ***perfil UML*** se define como un conjunto de *estereotipos*, *restricciones* y *valores etiquetados*. Estos conceptos forman parte del mecanismo de extensión de UML. Explicaremos brevemente cada uno de ellos:

- Mediante ***estereotipos*** se pueden crear nuevos tipos de elementos de modelado basados en elementos ya existentes en el metamodelo de UML. Por lo tanto, un estereotipo será un nuevo tipo de elemento de modelado que extiende la semántica del metamodelo. Los estereotipos están definidos por un nombre y por una serie de elementos del metamodelo sobre los que puede asociarse. Gráficamente un estereotipo se representa con su nombre entre comillas `<<nombre-estereotipo>>`. Opcionalmente el elemento estereotipado se puede dibujar con un nuevo icono asociado al estereotipo.

- Las ***restricciones*** imponen condiciones que deben cumplir determinados elementos del modelo para que éste esté "bien formado", según un dominio de aplicación específico. Gráficamente, una restricción se representa como una cadena de caracteres entre llaves colocada junto al elemento al que está asociada o conectada a él por una relación de dependencia. Una restricción generalmente se define mediante una expresión en OCL.

- Un ***valor etiquetado*** es una extensión de las propiedades de un elemento de UML, permitiendo añadir nueva información en la especificación del elemento. Gráficamente un valor etiquetado se representa como una cadena de caracteres entre llaves asociada al nombre del elemento. La cadena incluye un nombre (etiqueta), un separador (=), y un valor (el de la etiqueta).

En la Figura 16 podemos ver un ejemplo sencillo de la representación gráfica de estos tres elementos en UML.

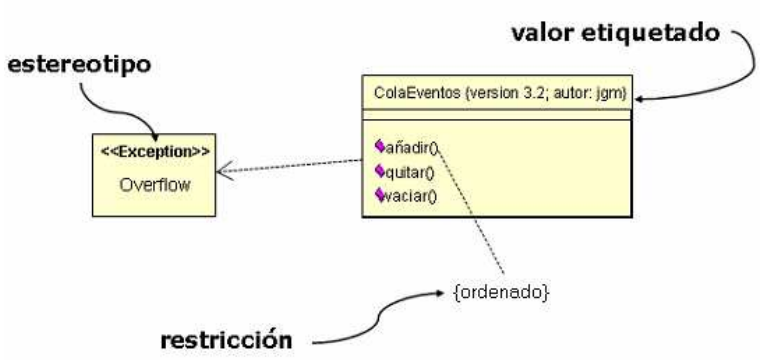


Figura 16. Ejemplo de *estereotipo*, *restricción* y *valor etiquetado* en UML

Para obtener un perfil tendremos entonces que especializar un subconjunto de UML a través de estereotipos, restricciones y valores etiquetados. Como resultado de crear un *perfil UML* obtenemos una variante de UML para un propósito específico, que podemos usar para completar un modelo con detalles específicos de una tecnología o plataforma determinada, o lo que es lo mismo, podemos usarlo como **lenguaje para definir PSMs**.

Los perfiles UML nos permiten:

- Disponer de una terminología y vocabulario propio de un dominio de aplicación o de una plataforma de implementación concreta.
- Definir una nueva notación para símbolos ya existentes, más acorde con el dominio de la aplicación final.
- Añadir cierta semántica que no existe o no aparece determinada de forma precisa en el metamodelo.
- Añadir restricciones a las existentes en el metamodelo, restringiendo su forma de utilización.
- Añadir información que puede ser útil a la hora de transformar el modelo a otros modelos o a código.

Actualmente existen perfiles para CORBA, Java, EJB o C++, lo que permitirá construir PSMs específicos para estas tecnologías.

3.3 OCL

Object Constraint Languaje (OCL) es un lenguaje de especificación con el que podemos escribir expresiones sobre modelos, por ejemplo, el cuerpo de una operación de consulta, invariantes y pre y postcondiciones. De esta manera, podemos definir modelos más precisos y completos. Algunos de los usos de OCL son:

- Especificar valores iniciales de atributos.
- Definir el cuerpo de operaciones de consulta.
- Establecer condiciones de guardia en *Statecharts.*
- Especificar las reglas de derivación para atributos o asociaciones.
- Expresar restricciones sobre clases o atributos.

A continuación mostraremos algunos ejemplos de expresiones en OCL, extraídos de [25].

La primera expresión define un invariante para la clase *Compañia*, estableciendo que el número de empleados debe ser mayor que 50.

```
context Compañia inv:
self.numeroDeEmpleados > 50
```

El siguiente ejemplo muestra la postcondición de una operación, que establece las ganancias de una persona en una fecha determinada a 5000. La palabra reservada *result* representa el resultado de la operación.

```
context Persona::ganancias(d : Date) : Integer
post: result = 5000
```

El último ejemplo muestra el cuerpo (*body*) de una operación de consulta que recupera el cónyuge actual de una persona. La operación establece como precondición (*pre*) que la persona esté casada.

```
context Persona::getConyugeActual() : Persona
pre: self.estaCasado = true
body: self.matrimonios->select( m | m.finalizado = false ).conyuje
```

Además de dar más precisión a los modelos, OCL puede usarse de manera muy efectiva en la definición de transformaciones: una primera expresión en OCL especifica los elementos del modelo origen, y una segunda expresión describe los elementos en el modelo destino de la transformación. Las condiciones necesarias para aplicar una transformación también pueden expresarse usando OCL. No obstante, el lenguaje QVT, que veremos más adelante, es el lenguaje estándar propuesto por el OMG para definir transformaciones.

4 Transformaciones de Modelos

4.1 Definiciones de Transformación

Una **definición de transformación** o ***mapping MDA*** proporciona la especificación de la transformación de un PIM en un PSM para una plataforma determinada. Según [21], podemos distinguir dos tipos de definiciones de transformación:

- **Transformaciones de tipos (*Model Type Mapping*)**. Según la especificación, "*un mapping de tipos especifica un mapping para transformar cualquier modelo construido con tipos del PIM a otro modelo expresado con tipos del PSM*". Es decir, a cada tipo de elemento del PIM se le aplica una regla determinada para transformarlo en uno o varios elementos del PSM. En el caso de UML, estas reglas pueden estar asociadas a tipos del metamodelo (clase, atributo, relación, etc.) o a nuevos tipos definidos mediante *estereotipos*, como puede verse en el primer ejemplo. También pueden definirse reglas en función de valores de instancias en el PIM, como se ilustra en el segundo ejemplo.

  Ejemplos:

  1. *Para cada instancia de clase con el estereotipo* `<<ComponentSegment>>` *en el PIM, transformarlas a un EJB Entity en el PSM para la tecnología EJB.*

  2. *Para cada atributo del PIM con valor "public" en la visibilidad, transformarlo en el PSM a un atributo con visibilidad "private" y añadir dos métodos de consulta y actualización (get y set) a la clase a la que pertenezca.*

- **Transformaciones de instancias (*Model Instante Mapping*).** Identifica elementos específicos del PIM que deben ser transformados de una manera particular, dada una plataforma determinada. Esto se puede conseguir mediante **marcas**.

  Una marca representa un concepto del PSM, y se aplica a un elemento del PIM para indicar cómo debe ser transformado. Las marcas, al ser específicas de la plataforma, **no son parte del PIM**. El desarrollador marca el PIM para dirigir o controlar la transformación a una plataforma determinada.

  Ejemplo:

  Para la plataforma EJB, podríamos tener la marca Entity aplicable a clases de un PIM; si está activa, esta marca indicará que la clase será transformada a un EJB Entity en el PSM.

La mayoría de las definiciones de transformación consistirán en alguna **combinación de los dos enfoques**. Un transformación de tipos solo es capaz de expresar transformaciones en términos de reglas sobre elementos de un tipo en el PIM que se

transforman en elementos de uno o más tipos en el PSM. Sin embargo, si no se puede marcar el modelo con información adicional para dirigir la transformación, la transformación siempre será determinística, y se basará íntegramente en información independiente de la plataforma para generar el PSM.

Asímismo, toda transformación de instancias del modelo tiene restricciones implícitas de tipo que deben cumplirse al marcar el modelo para que la transformación tenga sentido. Implícitamente a cada tipo de elemento del PIM solo pueden aplicarse determinadas marcas, que indican qué tipo de elemento se generará en el PSM. Las transformaciones basadas en marcas deben establecer qué marcas son aplicables a qué tipos del PIM.

4.2 Características Deseables de las Transformaciones

Para poder llevar a cabo el enfoque propuesto por MDA, conviene que el proceso de transformación disponga de una serie de propiedades o características. En el capítulo 7 de *MDA Explained* [19] se exponen las características deseables que deben tener las transformaciones en MDA. Estas propiedades son, por orden de importancia, la posibilidad de **ajustar las transformaciones,** la **trazabilidad**, la **consistencia incremental** y la **bidireccionalidad**. Las explicaremos brevemente en los siguientes apartados.

Ajustar las Transformaciones

El requisito principal que debería cumplir toda herramienta de transformación es permitir que el desarrollador tenga cierto control sobre el proceso de transformación. Esto puede lograrse de varias maneras:

- **Control manual**: implica un control directo del usuario sobre la transformación. El usuario puede definir manualmente qué elementos del modelo van a ser transformados por qué reglas de transformación. Esta es la solución más flexible, pero es propensa a errores y complica la tarea del desarrollador, por lo que es el método menos utilizado.

- **Condiciones en las transformaciones**: el desarrollador asigna una condición a cada regla de transformación, la cual describe cuándo debe aplicarse la regla. Este enfoque puede combinarse con el control manual. Un ejemplo podría ser: "*Todas las clases con el estereotipo* `<<persistent>>` *se transforman en …"*.

- **Parámetros de transformación**: las definiciones de transformaciones pueden parametrizarse para permitir al desarrollador cambiar el "estilo" de las transformaciones. Por ejemplo, cuando se transforma un atributo público en uno privado con métodos *get* y *set*, los prefijos exactos (normalmente *get* y *set*) podrían definirse como parámetros de la transformación. Otro parámetro típico de transformación podría ser la longitud de tipos de datos de tamaño indefinido. Por ejemplo, en lugar de transformar un *String* en el PIM en un *VARCHAR(40)* en un PSM relacional, podríamos definir la

longitud del dato como un parámetro de transformación. Los parámetros de transformación suponen el principal método de control sobre el proceso de transformación. Las marcas vistas en el apartado anterior pueden verse como parámetros de transformación para instancias del modelo.

Trazabilidad

La *trazabilidad* implica que pueda conocerse el elemento origen a partir del cual se ha generado cualquier elemento del modelo destino.

Dentro del marco de MDA, la trazabilidad es una característica muy útil en muchas situaciones. Imaginemos que cambiamos el nombre de una operación del PSM que ha sido generada directamente a partir del PIM. Sería deseable que ese cambio se reflejase también en el PIM. Esto no es posible si la herramienta no dispone de un mecanismo para conocer el origen en el PIM de la operación modificada.

La trazabilidad también es útil en la búsqueda y corrección de errores. Las partes de código erróneas pueden encontrarse buscando los elementos del PIM que presentan la funcionalidad defectuosa y siguiendo su traza hasta el código.

Consistencia Incremental

Normalmente, cuando se genera un modelo destino, éste necesita algún trabajo extra, como rellenar el código de una operación o refinar la interfaz de usuario. Si regeneramos de nuevo el modelo destino, debido a cambios en el modelo origen, querremos que ese trabajo extra se conserve. Esto es lo que llamamos *consistencia incremental*.

Cuando se produce un cambio en el modelo origen, el proceso de transformación sabe qué elementos en el modelo destino necesitan cambiarse también. Un proceso de transformación incremental puede reemplazar los elementos viejos con los nuevos, mientras mantiene la información extra del modelo destino en su sitio. Esto significa que cambios en el modelos origen tienen el mínimo impacto en el modelo destino.

Bidireccionalidad

La *bidireccionalidad* implica que las transformaciones puedan operar en ambas direcciones. Esta propiedad, aunque interesante, tiene menor prioridad que las anteriores. Las transformaciones bidireccionales pueden lograrse de dos formas:

- Ambas transformaciones se ejecutan de acuerdo con una única definición de transformación.
- Una transformación y su inversa se especifican mediante dos definiciones de transformación diferentes.

Es muy complicado definir transformaciones bidireccionales, principalmente porque es muy difícil asegurar que una definición de transformación es la inversa de otra. De hecho, en muchas ocasiones la bidireccionalidad es imposible de conseguir. Por

ejemplo, cuando transformamos un modelo de negocio en un modelo relacional, sólo transformamos la información estructural del modelo origen, ignorando toda la información dinámica. Esto hace imposible regenerar el modelo de negocio completo a partir del modelo relacional.

4.3 Herramientas de Transformación

Ninguna de las promesas de MDA sería posible sin herramientas que den soporte a este nuevo framework: transformación de modelos, verificación de modelos, generación de código, etc. La mayor parte del peso recae sobre estas nuevas herramientas de construcción de software.

El soporte para MDA puede darse en diferentes variantes. La generación de código a partir de modelos ya se lleva haciendo desde hace más de una década, y encaja bien dentro del marco de MDA. Además de la generación de código, necesitamos otros tipos de transformaciones, fundamentalmente transformaciones de modelos, para dar soporte completo a MDA. Estas transformaciones pueden ser implementadas por distintas herramientas, que resumiremos a continuación.

Herramientas de transformación de PIM a PSM

Permiten transformar un PIM de alto nivel en uno o varios PSMs. En la actualidad no hay disponibles muchas herramientas de este tipo, y las que existen ofrecen escasa funcionalidad. La herramienta *OptimalJ* es una de las pocas herramientas que ofrecen esta funcionalidad (Véase el apartado La herramienta OptimalJ, pág. 35).

Herramientas de transformación de PSM a código

El soporte más popular para MDA lo encontramos en las herramientas que transforman PSM a código. Poseen una definición de transformación integrada que toman un tipo predefinido de modelo de entrada (PSM) y producen otro tipo predefinido como salida (modelo de código). Las herramientas CASE tradicionales siguen este patrón para la generación de código.

Muchas de estas herramientas mantienen la relación entre el PSM y el código, permitiendo que los cambios realizados en cualquiera de los modelos (PSM o código) se refleje inmediatamente en el otro. Esto es posible porque PSM y código están muy cerca el uno del otro, y poseen casi el mismo nivel de abstracción.

Herramientas de transformación de PIM a código

Otro tipo de herramienta es aquella que soporta las dos transformaciones anteriores, de PIM a PSM y de PSM a código. En ocasiones el usuario simplemente verá una transformación directa de PIM a código, quedando implícito el PSM. La herramienta *ArcStyler* podría enmarcarse en este grupo, ya que no existe PSM explícito, sino que se

usa un PIM con *marcas* para generar directamente el código (Véase el apartado La Herramienta ArcStyler, pág. 57).

Herramientas de transformación ajustables

Las herramientas de MDA deberían permitir *refinar* o *ajustar* las transformaciones. Normalmente no se permite acceder a las definiciones de las transformaciones para ajustarlas a tus propios requisitos. En el mejor de los casos, las herramientas proporcionan sus definiciones de transformaciones en lenguajes de *script* específicos de la herramienta, y resulta muy engorroso hacer cambios en dichos lenguajes. Mientras no se estandarice un lenguaje de definición de transformaciones (Véase QVT en la página 29), éstas seguirán siendo específicas de cada herramienta.

Incluso la mayoría de herramientas trabajan con un lenguaje para PIMs predefinido, que suele ser una variante restrictiva de UML, e internamente usan también su propia definición de UML. Esto obliga a los usuarios de la herramienta a aprender la definición específica de UML de la herramienta para poder escribir definiciones de transformaciones.

Herramientas de definición de transformaciones

Un último tipo de herramientas lo constituirían aquellas herramientas que nos permiten crear y modificar transformaciones sobre modelos, imprescindibles si necesitamos usar transformaciones distintas a las disponibles en las herramientas. Las únicas herramientas disponibles de este tipo usan lenguajes de *script* específicos. La complejidad de la definición de transformaciones nos lleva de nuevo a la necesidad de un lenguaje estándar de definición de transformaciones (QTV) y a disponer de herramientas mejor preparadas para esta tarea. Por desgracia, todavía queda algo de tiempo para que esto se convierta en realidad.

Otras herramientas

Las herramientas de transformación son el verdadero corazón de un desarrollo con MDA, pero no son las únicas que necesitamos. En *MDA-Explained* [19] se detalla la funcionalidad más relevante para un lograr un entorno completo de desarrollo en MDA, que resumiremos a continuación:

- **Editor de código**. Realiza las tareas comunes proporcionadas por un Entorno de Desarrollo Interactivo (*Interactive Development Enviroment* o *IDE*), como depuración, compilación y edición de código.

- **Ficheros de código**. Aunque podemos considerar el código como un modelo, usualmente se almacena en forma de ficheros de texto. Los ficheros de texto no son un formato que otras herramientas puedan entender. Por consiguiente, necesitamos los siguientes dos elementos:

  - Analizador gramatical de ficheros de texto que lea un fichero de texto y almacene el código en forma de modelo en el repositorio de modelos, de manera que otras herramientas lo puedan usar.
  - Generador de ficheros de texto que lea el código del repositorio de modelos y produzca un fichero de código basado en texto.

- **Repositorio de modelos**. Es la base de datos para los modelos, donde los modelos se almacenan y se recuperan usando XMI, JMI o IDL.

- **Editor de modelos**. Herramienta CASE para construir y modificar modelos.

- **Verificador de modelos**. Los modelos usados para la generación de otros modelos deben estar extremadamente bien definidos. Estas herramientas pueden chequear los modelos para comprobar que cumplen unas determinadas reglas y asegurarse así que puedan ser transformados.

- **Editor de definiciones de transformaciones**, para permitir crear y modificar definiciones de transformaciones.

- **Repositorio de definiciones de transformaciones**, para almacenar las definiciones de transformaciones.

La Figura 17 muestra un diagrama que resume la funcionalidad deseable en un entorno completo de desarrollo con MDA.

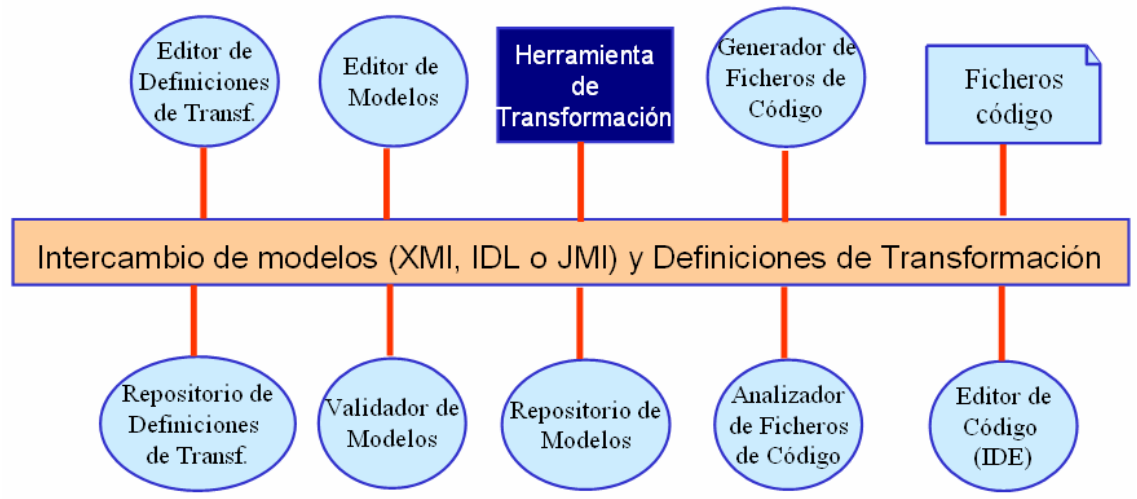


Figura 17. Funcionalidad en un entorno de desarrollo MDA

La mayoría de las herramientas actuales combinan, con mayor o menor medida, un número de estas funciones. Las herramientas CASE tradicionales proporcionan un editor de modelos y un repositorio de modelos. Un generador de código adherido a una herramienta CASE provee de la herramienta de transformación y del editor de definiciones de transformaciones.

4.4 QVT

Query, Views and Transformations (QVT) es un estándar actualmente en desarrollo, que define el modo en que se llevan a cabo las transformaciones entre modelos cuyos lenguajes han sido definidos usando MOF. Este estándar se incluirá en MOF, y constará de tres partes:

- Un lenguaje para crear vistas de un modelo.
- Un lenguaje para realizar consultas sobre modelos.
- Un lenguaje para escribir definiciones de transformaciones.

La última parte es la más relevante para MDA, pues actualmente no existe un modo estándar de definir transformaciones entre modelos.

Los próximos apartados explicarán brevemente los aspectos principales de este estándar todavía en desarrollo, obtenidos de la propuesta de QVT elaborada por *QVT-Partners* [27].

Consultas

Como lenguaje de consulta, el grupo *QVT-Partners* proponen una versión extendida de OCL 2.0 [25]. Con esta elección se obtendrían varios beneficios:

- La comunidad de usuarios ya está familiarizado con el lenguaje.
- No se necesita gastar esfuerzo en definir un nuevo lenguaje.
- Existe actualmente un soporte sustancial para OCL en muchas herramientas.

Vistas

QVT-Partners proponen que una vista sea una proyección de un modelo padre, creada mediante una transformación. No profundizaremos en este apartado, sólo destacaremos que la propuesta contempla el uso de consultas para crear las vistas de un modelo.

Transformaciones

La parte principal de la propuesta descrita en [27] se centra en las transformaciones, lo más relevante para MDA. La Figura 18 muestra los elementos más importantes de la sintaxis usada para representar las transformaciones.

La propuesta distingue entre **dos tipos de transformaciones**:

- **Relaciones**: especificaciones de transformaciones multidireccionales. No son ejecutables, en el sentido de que son incapaces de crear o modificar un modelo. Sin embargo pueden chequear la consistencia entre dos o más modelos relacionados. Típicamente se usarán en la especificación del desarrollo de un sistema o para chequear la validez de un *mapping*.

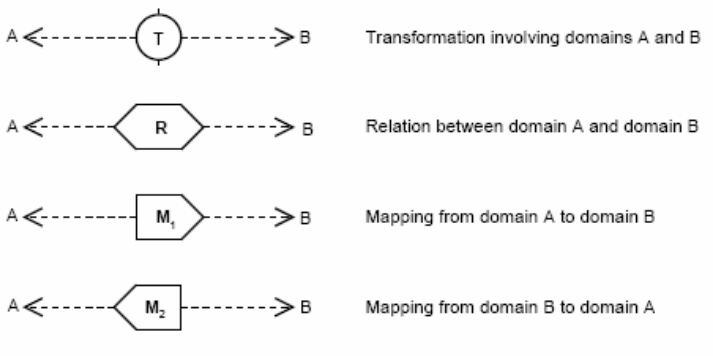


Figura 18. Sintaxis de las transformaciones

- ***Mappings***: implementaciones de transformaciones. A diferencia de las relaciones, los *mappings* son unidireccionales y pueden devolver valores. Un *mapping* puede *refinar* una o varias relaciones, en cuyo caso el *mapping* debe ser consistente con las relaciones que refina.

Ejemplo:

La Figura 19 *muestra una relación R entre dos dominios. También hay un mapping M que refina la relación R. Como el mapping M es dirigido (la flecha apunta hacia la izquierda), significa que transforma los elementos del modelo de la derecha en los elementos del modelo de la izquierda.*

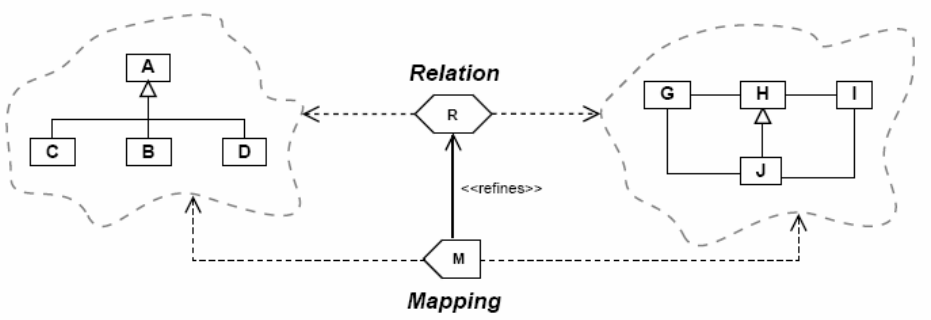


Figura 19. Una relación refinada por un *mapping*

La propuesta de *QVT-Partners* también incluye un lenguaje estándar para definir relaciones y *mapping*, llamado ***Model Transformation Language*** o **MLT**. MLT usa el ***pattern matching*** como uno de sus factores clave para permitir definir transformaciones potentes. La idea esencial detrás del *pattern matching* es permitir expresar brevemente restricciones complejas sobre un tipo de dato de entrada; los datos que cumplen el patrón se seleccionan y se retornan al invocador.

Ejemplo:

Un ejemplo de definición de relación en forma textual es el siguiente:

```
relation IncrementoSabiduria {
    domain { (Persona)[nombre = n, edad = a, sabiduria = w1]
                when a + 1 < 13 or a + 1 > 19 }
    domain { (Persona)[nombre = n, edad = a + 1, sabiduria = w2] }
    when { w2 > w1 }
}
```

Intuitivamente, este ejemplo comprueba que un cumpleaños va acompañado de un incremento de la sabiduría, excepto durante la adolescencia (entre los 13 y los 19 años), en la que no siempre se cumple ese caso. La expresión `(Persona)[nombre = n, edad = a + 1, sabiduria = w2]` *es un ejemplo de patrón.*

5 Criterios para Evaluar Herramientas MDA

En este apartado presentamos los criterios elegidos para evaluar herramientas MDA y se proporciona una breve descripción de la aplicación *Pet Store* usada como ejemplo para probar las herramientas.

Durante la etapa de este proyecto en la que se trabajaba en la identificación de las dimensiones y criterios de evaluación de herramientas MDA, tuvimos conocimiento de la publicación de los resultados de un interesante trabajo [20] en el que se había evaluado la herramienta OptimalJ y se habían identificado unos criterios completos y bien definidos que debe cumplir una herramienta con soporte completo para MDA. Dada la calidad de los criterios presentados en dicho estudio, hemos decidido usar la misma metodología para nuestra evaluación, adaptando ligeramente algunos de los criterios y añadiendo otros nuevos.

A continuación se muestra una tabla con las propiedades evaluadas en este estudio:

Id	Propiedad	Descripción
P01	Soporte para PIMs	La herramienta permite que se especifique un sistema mediante un modelo independiente de cualquier plataforma (PIM).
P02	Soporte para PSMs	La herramienta permite construir modelos del sistema que capturan los aspectos esenciales de una tecnología de implementación determinada (PSMs).
P03	Permite varias implementaciones	La herramienta posibilita la generación de varias implementaciones diferentes a partir del mismo PIM, utilizando PSMs, marcas u otros mecanismos. También se tendrá en cuenta que puedan añadirse nuevas implementaciones a las disponibles en la herramienta.
P04	Integración de Modelos	Permite integrar diferentes modelos para producir una única aplicación, principalmente mediante la generación de los "puentes" apropiados para comunicar las distintas partes entre sí.
P05	Interoperabilidad	La herramienta puede importar y exportar información a otras herramientas.
P06	Acceso a la definición de las transformaciones	La herramienta provee de un mecanismo de definición de transformaciones entre modelos y permite al usuario crear nuevas transformaciones o modificar las existentes para satisfacer sus requisitos específicos.
P07	Verificador de modelos	Incluye algún mecanismo para chequear la corrección de los modelos, incluidos PIM y PSM.
P08	Expresividad de los modelos	La herramienta tiene un lenguaje para representar PIMs y PSMs lo suficientemente expresivo como para capturar de forma precisa la estructura y funcionalidad en los distintos niveles de abstracción.
P09	Uso de patrones	La herramienta aplica o permite aplicar patrones de diseño en la construcción de PIMs, PSMs y código, y pueden definirse otros nuevos o modificar los existentes.
P10	Soporte para la regeneración de modelos	La herramienta proporciona soporte para rehacer modelos, por ejemplo, regenerar el PIM a partir de los PSMs y viceversa. También debe permitir conservar los cambios efectuados "manualmente" tanto a nivel de modelo como de código.

P11	Transformaciones intra-modelo	Provee soporte para transformar un PSM a otros PSMs, o un PIM a otros PIMs.
P12	Trazabilidad	Incorpora un mecanismo para seguir el rastro de determinadas transformaciones desde su origen hasta su destino.
P13	Ciclo de vida	Incluye la mayor parte posible del ciclo de vida de un desarrollo con MDA, esto es, el análisis, el diseño, la implementación, el ensamblado, el despliegue y el mantenimiento.
P14	Estandarización	La herramienta utiliza los estándares básicos de MDA. Por ejemplo, expresa sus modelos en UML, es capaz de importar y exportar modelos en XMI y de guardarlos en un repositorio MOF.
P15	Control y refinamiento de las transformaciones	La aplicación permite dirigir o controlar las transformaciones entre modelos, entre PIM y PSMs o entre PSM y código. Por ejemplo, dispone de parámetros en las transformaciones, permite seleccionar los elementos a ser transformados o establecer condiciones para las transformaciones.
P16	Calidad del código generado	La aplicación genera código de calidad, bien documentado, legible y que puede ser adaptado o extendido fácilmente por el desarrollador.
P17	Herramientas de soporte	Además de las herramientas de transformación, la aplicación incluye otras herramientas para dar soporte completo a MDA: editor de código, editor de modelos, herramientas para prueba y despliegue, etc.

Para evaluar con detalle cada herramienta, puntuaremos **del 0 al 4** cada propiedad de la tabla anterior, según el siguiente criterio:

Puntuación	Descripción
0	Soporte nulo de la propiedad
1	Soporte mínimo para la propiedad
2	Soporte medio para la propiedad
3	Buen soporte de la propiedad
4	Excelente soporte de la propiedad

5.1 Caso de Estudio. Tienda de animales (Pet Store)

Para evaluar las herramientas hemos utilizado un ejemplo típico de aplicación de venta electrónica, el de la *Tienda de Animales* o *Pet Store*. Este ejemplo es usado por Sun como ejemplo típico de una aplicación J2EE [29], y se emplea con mucha frecuencia para probar nuevas herramientas y tecnologías.

Se trata de una tienda dedicada a la venta de animales que ha decidido extender su mercado mediante el uso de Internet. Para ello han pensado que podrían vender sus animales a través de la Web. Se trata de un sistema de información típico que permite a un cliente consultar el catálogo de animales, colocarlos en un carro de la compra y realizar un pedido a través de una interface Web.

El sistema debería permitir la siguiente funcionalidad:

- Autenticación de usuarios mediante login/password.

- Alta y modificación de usuarios.
- Consulta de los animales disponibles por categorías.
- Realización de pedidos de animales por parte de un usuario.
- Consulta de los pedidos realizados por el usuario.

Para los pedidos, el usuario indicará la dirección a la que desea que se le envíe el pedido. Además, se permitirán distintas formas de pago: tarjeta de crédito, contra reembolso y transferencia bancaria.

Los productos de la tienda son animales clasificados en categorías (perros, gatos, peces, etc.). De cada animal nos interesa su nombre y su precio. Sería también interesante la inclusión de una breve descripción del animal.

Como vemos, es un sencillo ejemplo que puede representarse fácilmente mediante el diagrama de clases de la Figura 20.

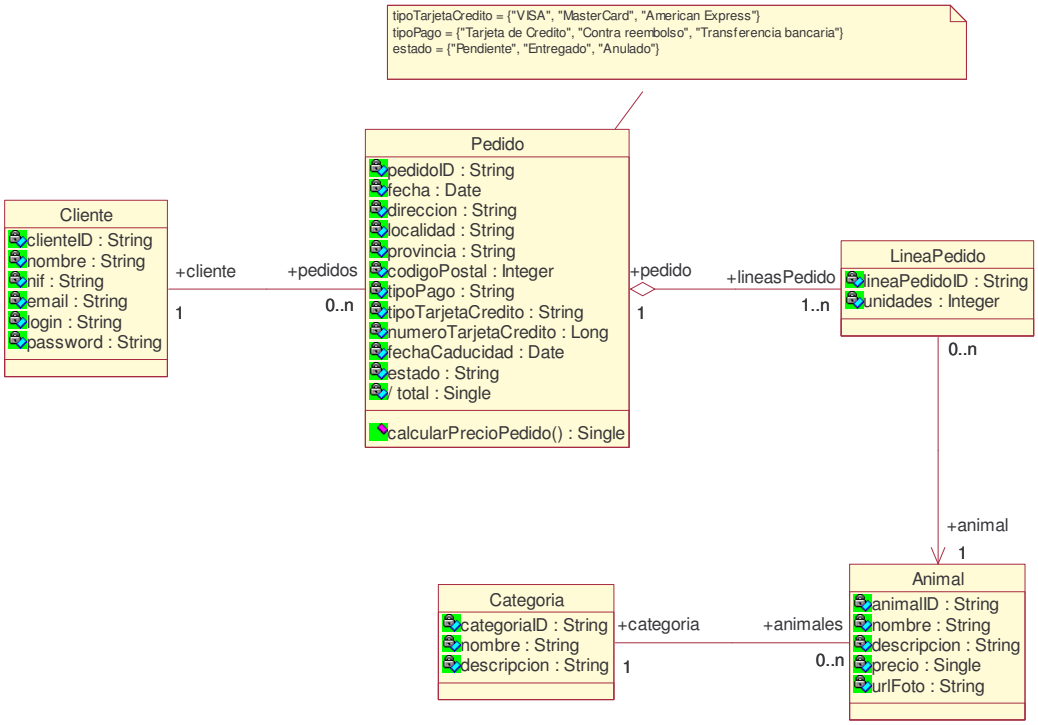


Figura 20. Diagrama de clases del sistema (PIM)

Este modelo sería el PIM de la aplicación que queremos construir, y como tal constituye la base de todo el desarrollo de la aplicación.

Del sistema propuesto no se implementará necesariamente toda la funcionalidad. Nuestro objetivo es comprobar hasta qué grado se puede generar una aplicación completa utilizando una herramienta MDA, cambiando y añadiendo el menor código fuente posible.

6 La herramienta OptimalJ

La primera herramienta MDA que evaluaremos es ***OptimalJ*** de *Compuware*. Se trata de un entorno de desarrollo de aplicaciones empresariales que permite generar con rapidez **aplicaciones J2EE** completas directamente a partir de un modelo de alto nivel (PIM), utilizando patrones para codificar adecuadamente las especificaciones de J2EE y haciendo uso de los estándares del MDA.

La versión evaluada en este estudio ha sido ***OptimalJ Professional Edition 3.0***. A fecha de finalización de este estudio ya está disponible la versión 3.1 de *OptimalJ Professional Edition*. Hemos probado brevemente esta nueva versión y no hemos apreciado cambios sustanciales entre ambas versiones, así que la evaluación efectuada para la versión 3.0 puede aplicarse también a la versión 3.1.

6.1 Arquitectura de OptimalJ

Introducción

OptimalJ implementa el framework MDA haciendo uso de tecnologías estándar como MOF, UML, XMI, XML, WSDL, y J2EE.

Básicamente, distingue entre tres tipos de modelos:

- **Modelo del Dominio** (*Domain Model*): modelo de alto nivel mostrando la arquitectura general del sistema, sin detalles de implementación. Equivale al PIM de la aplicación.
- **Modelo de la Aplicación** (*Application Model*): modelo del sistema desde el punto de vista de una tecnología determinada (J2EE). Contiene los PSMs de la aplicación.
- **Modelo de Código (***Code Model*): código de la aplicación, generado a partir del modelo de la aplicación.

Asimismo, OptimalJ usa dos tipos de patrones:

- Patrones de transformación **entre** modelos:
  - **Patrones de tecnología** (*Technology patterns*): transforman el modelo del dominio (PIM) en el modelo de aplicación (PSMs).
  - **Patrones de implementación** (*Implementation patterns*): transforman el modelo de aplicación (PSMs) en código.

- Patrones funcionales para hacer transformaciones **dentro** de un modelo. Estos patrones promueven *best practices*[9], reducen los errores y aceleran el desarrollo de la aplicación. OptimalJ ofrece:

    - **Patrones de dominio** (*Domain patterns*): patrones definidos por el usuario que permiten a los desarrolladores capturar, reutilizar y distribuir modelos del dominio.

    - **Patrones de aplicación** (*Application patterns*): usan el mismo principio que los patrones del dominio, pero aplicados a los modelos de aplicación.

    - **Patrones de código** (*Code patterns*): patrones de bajo nivel aplicados al código, como los patrones GoF [10] .

En la Figura 21 podemos ver un esquema de los tipos de modelos y de patrones disponibles en OptimalJ

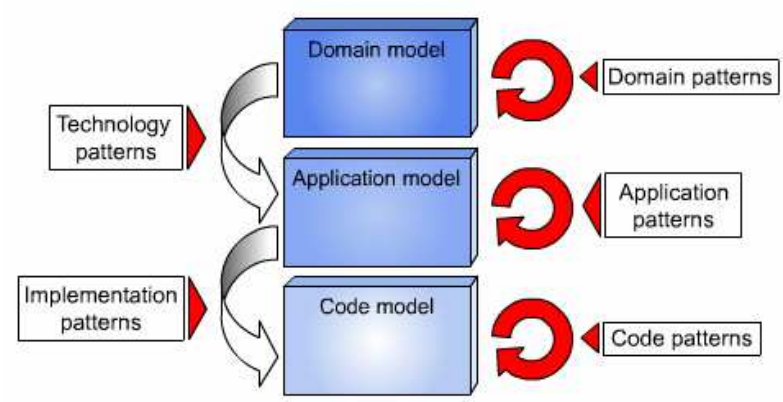


Figura 21. Tipos de modelos y patrones en OptimalJ, extraído de [6]

Explicaremos cada tipo de modelo y de patrón con más detalle en los próximos apartados.

Modelo del Dominio (*Domain Model*)

En OptimalJ el PIM está contenido dentro del **Modelo del Dominio** (*Domain Model*). Este modelo del dominio está formado a su vez por otros dos modelos: el **Modelo de Clases** y el **Modelo de Servicios.**

Dentro del modelo del dominio podemos definir **Patrones de Dominio** para reutilizar un modelo del dominio en futuros desarrollos.

Modelo de Clases (*Domain Class Model*)

En el **Modelo de Clases** se define la estructura de la información con la que trabaja la aplicación mediante un diagrama de clases UML. Este modelo es el **PIM** de la aplicación.

[9] Mejor solución encontrada para un problema determinado

Aquí se definirán, por ejemplo, las clases *Cliente*, *Pedido* o *Animal*, con sus respectivos atributos, operaciones y relaciones. En la Figura 22 se muestra el modelo de clases de OptimalJ para el ejemplo del *Pet Store*.

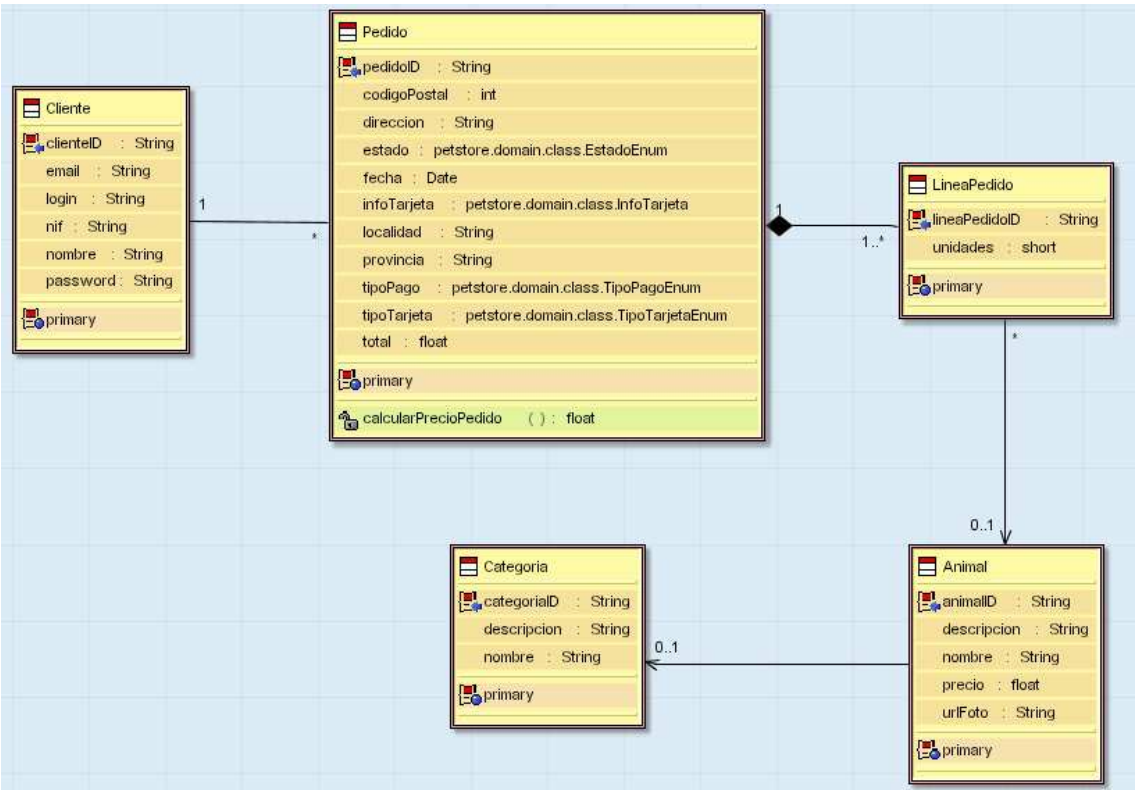


Figura 22. Modelo de Clases

Modelo de Servicios (*Domain Service Model*)

El **Modelo de Servicios** permite definir vistas sobre las clases definidas en el modelo de clases. Básicamente, la funcionalidad disponible en este modelo consiste en limitar el acceso a las acciones de crear, leer, modificar o eliminar instancias, y ocultar la aparición de determinados atributos en la interfaz de la aplicación generada.

En la Figura 23 podemos un ejemplo de modelo de servicios con un único servicio (*VistaClientes*), el cual proporciona una vista alternativa para la clase *Cliente*. El servicio *VistaClientes* podría, por ejemplo, ocultar los atributos *login* y *password* del cliente (ocultación de atributos) e impedir la eliminación de pedidos de los clientes (restricción en el uso de operaciones).

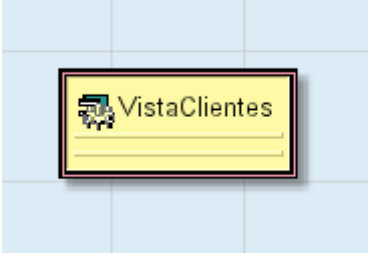


Figura 23. Modelo de Servicios

La ventaja principal de este modelo es que permite generar automáticamente *Beans* de tipo *Session* en el modelo de EJB, aunque la realidad es que no ofrece mucha flexibilidad a la hora de definir las vistas.

Patrones de Dominio

OptimalJ permite usar **Patrones de Dominio** (*Domain Patterns*) para reutilizar el conocimiento del dominio del negocio. Un patrón de dominio es un modelo del dominio que puede ser reutilizado, de manera que podemos construir un nuevo modelo del dominio rápidamente basado en el conocimiento existente.

Por defecto, todos los elementos del patrón del dominio se copian a nuevos elementos en el modelo destino. Pero también es posible asignar un elemento de un modelo existente a un elemento del patrón. Cuando se aplica el patrón, el elemento del patrón se *mezcla* con el elemento del modelo destino. Esta técnica se conoce como *entramado* (*weaving*).

Los patrones de dominio se pueden aplicar usando un asistente para la aplicación de patrones (*Pattern Application Wizard*). Podemos ver este asistente en la Figura 24. Después de seleccionar un patrón de dominio, el desarrollador especifica cómo este patrón debe ser aplicado al modelo del dominio destino. En el caso de la Figura 24 no puede aplicarse *weaving* porque el modelo destino está vacío, así que directamente se copiarán los elementos del patrón en el modelo.

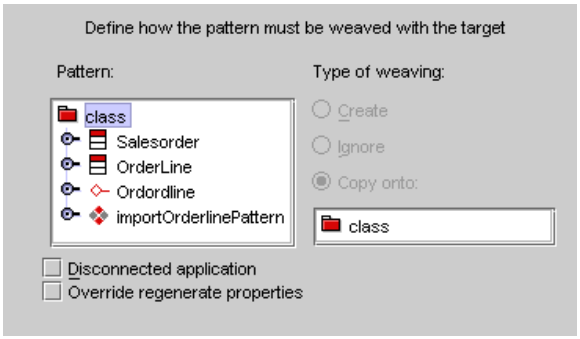


Figura 24. Asistente para la aplicación de patrones de dominio

Los patrones de dominio proporcionan un número de beneficios clave a los diseñadores:

1. Los diseñadores pueden reutilizar modelos del dominio, o un subconjunto de ellos, en diferentes aplicaciones, reduciendo el número de errores cometidos en la fase de modelado.

2. Los patrones del dominio fuerzan a los diseñadores a modelar aplicaciones consistentes, basadas en estándares y *best practices*.

3. Incrementan la productividad en la fase de modelado.

Modelo de Aplicación (*Application Model*)

Una vez construido el modelo del dominio, OptimalJ permite generar automáticamente el **Modelo de Aplicación** (*Application Model*), modelo que consta de varios PSMs orientados hacia la plataforma J2EE. Esta transformación la llevan a cabo los **Patrones de Tecnología**.

El modelo de aplicación generado contiene tres tipos de modelos: el **Modelo de Presentación**, el **Modelo de Negocio** y el **Modelo de Base de datos**. Describiremos estos modelos en los próximos apartados.

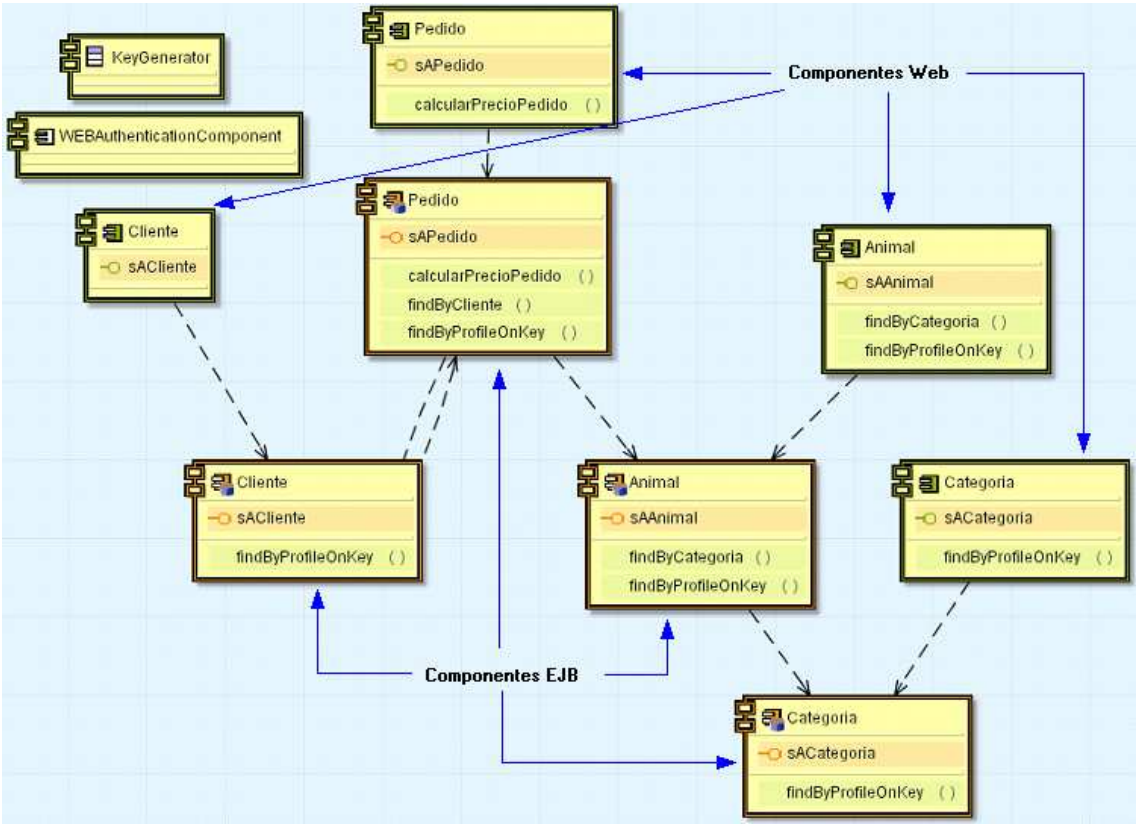


Figura 25. Modelo de Presentación (Web)

Modelo de Presentación (Web)

Este modelo contiene la información necesaria para generar una capa Web para nuestra aplicación, según la plataforma J2EE[10]:

- *Módulos Web*: contenedores de componentes web.
- *Web Data Schemas*: definen una colección de *Data Classes*.
- *Componentes Web*: definen una interfaz de usuario para un *Data Schema* dado.
- *Componentes de Autenticación Web*: definen métodos de autentificación para la aplicación.

[10] Muchos de los conceptos aquí mencionados pertenecen a la plataforma J2EE. Para cualquier duda, consúltese la documentación oficial de la plataforma J2EE [14].

- *Tipos de Presentación Web*: definen distintos tipos de presentación para los datos y validación de los mismos.

Podemos mejorar el aspecto de la aplicación generada editando las plantillas, CSSs (*Cascading Style Sheets*[11]) y JSPs que se generarán en el modelo de código. También se puede refinar el modelo Web de manera sencilla mediante el uso de asistentes (*wizards*).

En la Figura 25 puede verse el modelo de presentación para el *Pet Store*. Nótese que cada componente Web está asociado con su correspondiente componente de negocio (EJB) para permitir la comunicación entre ambas capas.

Modelo de Negocio (EJB)

El modelo de *Enterprise Java Bean* (EJB) [7] es una capa *middleware* encargada, entre otras cosas, de las transacciones, la seguridad, la persistencia y la escalabilidad de la aplicación. OptimalJ usa las definiciones contenidas en el modelo del dominio para generar automáticamente el modelo de EJB y los componentes de tipo entidad y sesión (*entity beans* y *session beans*). El modelo de EJB está formado por componentes de tipo entidad, *Data Schemas*, *Key Classes* y otros componentes relacionados con la tecnología EJB [7].

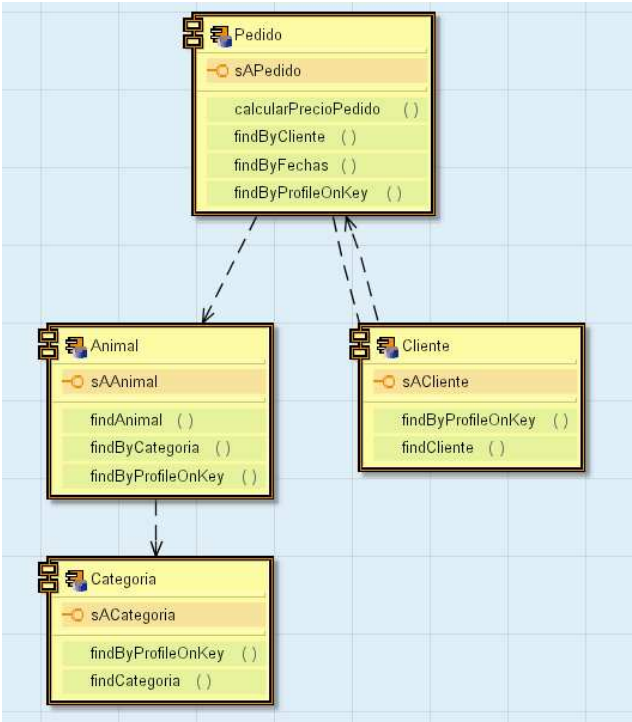


Figura 26. Modelo de Negocio (EJB)

Aunque el modelo del dominio guía todo el proceso de desarrollo, el desarrollador puede mejorar el modelo EJB generado definiendo componentes adicionales, como métodos de búsqueda, métodos de negocio, métodos *home* y métodos de selección.

[11] CSS es un mecanismo sencillo para añadir estilo (p. e. fuentes, colores, espaciado) a documentos Web.

La Figura 26 muestra el modelo de negocio generado a partir del modelo de clases de la Figura 22. Podemos ver cuatro componentes EJB interrelacionados, que representan EJBs de tipo entidad. También vemos que cada para cada componente se muestra, entre otros datos, sus métodos de negocio (p. e. `calcularPrecioPedido()`) y sus métodos de búsqueda (p. e. `findByCliente()`, `findByFechas()`).

Modelo de Base de Datos

La persistencia de los datos es un aspecto importante en el desarrollo de aplicaciones J2EE. La aproximación más común es usar bases de datos relacionales para almacenar los datos. Para hacer esto, es necesario una conversión objeto-relacional entre el modelo de objetos, representado por el modelo del dominio, y la base de datos.

OptimalJ maneja esta correspondencia generando automáticamente un **Modelo de Base de Datos** a partir del modelo del dominio usando **Patrones de Tecnología**. El modelo de base de datos de OptimalJ se usa para modelar todas las definiciones relevantes de la base de datos. Este modelo soporta esquemas, tablas, columnas, filas, claves ajenas, claves primarias y restricciones de unicidad.

La Figura 27 presenta el modelo de base de datos para el ejemplo del *PetStore*.

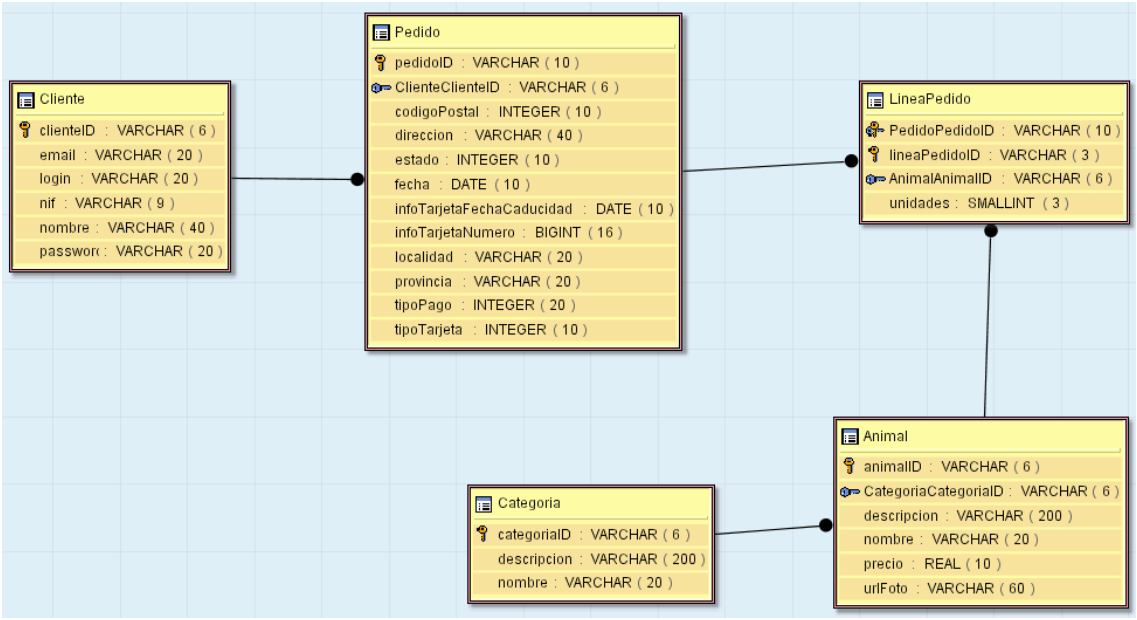


Figura 27. Modelo de Base de Datos

Modelo de Código (Code Model)

Una vez generado el modelo de aplicación a partir del modelo del dominio, OptimalJ automáticamente transforma el Modelo de la Aplicación en **código** haciendo uso de los **Patrones de Implementación**.

El código para la lógica de negocio (EJB), base de datos y presentación (Web) puede generarse automáticamente de forma separada, o todos a la vez. Este proceso de generación automática a partir del modelo de aplicación asegura la consistencia con el modelo del dominio, ahorrando una gran cantidad de tiempo y reduciendo el riesgo

potencial de errores en la programación. Naturalmente, el modelo Web depende del modelo EJB, y éste depende del Modelo de Base de Datos. OptimalJ mantiene esta interdependencia entre modelos y la traslada también al código generado.

El código que genera OptimalJ es totalmente operativo y conforme a las especificaciones de J2EE. El código generado consiste en ficheros Java, JSPs y XML, *beans* de tipo entidad y de tipo sesión, y código SQL.

Todo el código generado por OptimalJ se sitúa en **bloques protegidos**. Esto significa que los desarrolladores no pueden modificarlo. No obstante, OptimalJ deja disponibles **bloques libres** donde los desarrolladores pueden añadir sus propias extensiones al código generado. Cuando la aplicación es regenerada, por ejemplo, por un cambio en el modelo del dominio, la herramienta preserva el código escrito en los bloques libres.

Ejemplo:

En la Figura 28 *podemos ver el cuerpo del método de negocio* `calcularPrecioPedido()` *en el fichero de código PedidoBean.java generado por OptimalJ. Puede verse cómo el editor de código de OptimalJ colorea en azul los bloques protegidos (cabecera y retorno del método), de manera que el desarrollador no pueda modificar estas secciones. Sin embargo, el cuerpo del método es un bloque libre y aparece de color blanco. Así, el desarrollador puede implementar el método de negocio en este bloque libre, de modo que dicha implementación se preservará aunque se regenere el código de toda la aplicación.*

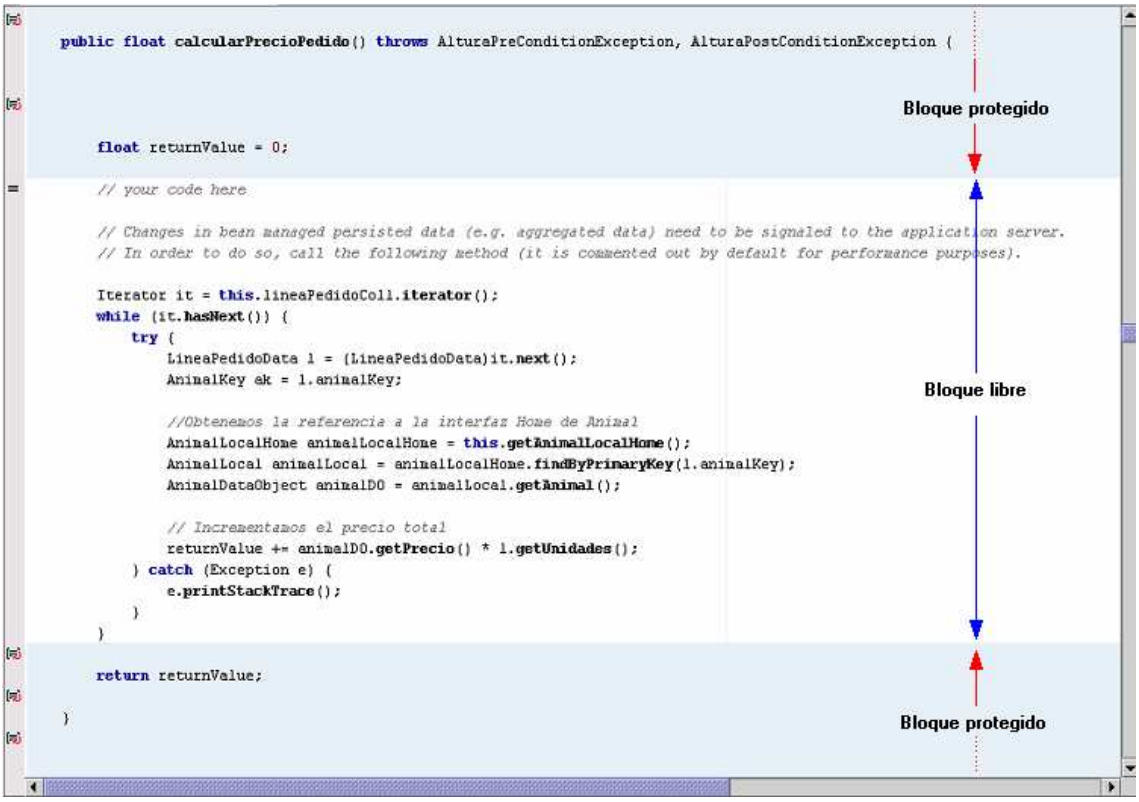


Figura 28. Distinción entre bloques libres (blanco) y bloques protegidos (azul)

OptimalJ también incorpora un completo editor de código con el que los desarrolladores pueden personalizar el código generado por la herramienta sin necesidad de recurrir a otros entornos de programación.

Implementación del modelo de EJB

Los EJBs generados por OptimalJ, tanto para EJB 1.1 como para EJB 2.0, son consistentes con las definiciones del modelo del dominio. También incorporan la funcionalidad necesaria para interactuar con la capa de presentación (JSPs), con la base de datos y con el servidor de EJB. OptimalJ genera los siguientes elementos principales de EJB:

- *Modulo EJB*: el contenedor para todos los elementos del modelo EJB.
- *EJB Data Schema*: especifica la estructura de datos de componentes EJB y contiene *EJB Data Classes* y *EJB Data Associations*.
- *EJB Key Class*: representa la restricción de clave primaria única. Los *key attributes* en las *EJB Key Classes* se derivan de la clave primaria del Modelo del Dominio.
- *Componente EJB Entidad*: representa datos en la base de datos y métodos de negocio que actúan sobre esos datos.
- *Componente EJB Sesión*: componente orientado al servicio que lleva a cabo operaciones sobre componentes de tipo entidad. Esto significa que no interactúa directamente con la base de datos. Permite agrupar datos relacionados y representarlos como una única interfaz.
- *EJB Message-driven Component*: representa a un consumidor de mensajes de JMS, diseñado en el modelo de EJB. Puede consumir un mensaje específico o consumir todos los mensajes de un particular destinatario (*topic* o *queue*). Este componente dará lugar a una implementación de un *bean* orientado a mensajes de J2EE.

Para incrementar la productividad, OptimalJ también incorpora su propio **entorno integrado de prueba y despliegue de EJB**.

Implementación del Modelo de Base de Datos

El desarrollo de una nueva aplicación requiere la elaboración de *scripts* para crear las tablas de la base de datos. OptimalJ genera automáticamente los comandos SQL necesarios para crear/destruir las tablas de la base de datos correspondientes al Modelo de Base de Datos.

OptimalJ también ofrece un banco de pruebas de SQL integrado para ejecutar los *scripts* SQL, que podemos ver en la Figura 29. Para el acceso a la base de datos, OptimalJ usa JDBC (*Java Database Connectivity*).

La herramienta también genera un fichero descriptor de despliegue (DAR), requerido para desplegar aplicaciones J2EE en un contenedor de J2EE.

OptimalJ ofrece también la posibilidad de usar *Data Access Object* (DAO) en lugar de EJBs para la lectura de los datos. Para almacenar los datos, sin embargo, EJBs son la

mejor elección, ya que garantizan que las transacciones se almacenarán de forma segura.

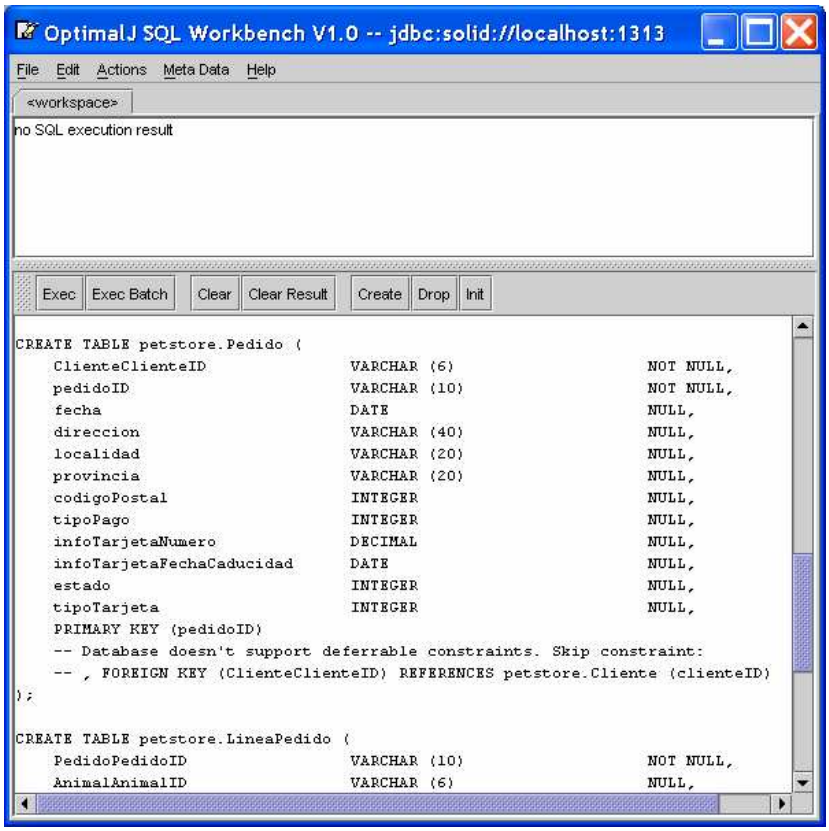


Figura 29. Banco de pruebas para SQL integrado en OptimalJ

Para la realización de pruebas desde la misma herramienta, el paquete de instalación de OptimalJ incorpora el **gestor de bases de datos *Solid Embedded Engine***, que nos permite probar nuestra base de datos sin recurrir a herramientas de terceros.

Implementación del Modelo de Presentación

La capa de presentación define un conjunto de componentes de presentación por defecto, principalmente en forma de JSPs y servlets. El modelo de la capa de presentación deriva directamente del modelo del dominio, lo cual asegura la consistencia y la calidad de los componentes de la capa de presentación.

La separación entre la capa de presentación (Web) y la capa de negocio (EJB) permite a los desarrolladores crear nuevas capas Web sobre una capa EJB existente. Es más, OptimalJ incorpora un conjunto de **Patrones de Presentación**. Seleccionando distintos Patrones de Presentación, por ejemplo, los mismos datos pueden mostrarse tanto en un componente web basado en HTML como en una interfaz de usuario basada en clases *Swing* de Java, usando la misma capa EJB. Como los componentes Web están directamente ligados a los componentes EJB, acciones como LEER, CREAR, BORRAR o MODIFICAR están disponibles directamente tras la generación de código.

OptimalJ implementa el modelo de presentación usando el **framework *Struts* de Apache** [1], basado en la arquitectura Modelo-Vista-Control (MVC). La arquitectura MVC separa la interfaz de la lógica de negocio y los datos, y ofrece múltiples beneficios sobre una solución basada exclusivamente en JSPs. Editando los JSPs, CSS y las plantillas JSPs podemos cambiar el aspecto general de nuestra aplicación. Los diseñadores también pueden definir interacciones del usuario con la aplicación usando las acciones Web (*Web actions*) disponibles, que pueden ser editadas.

Para modificar la navegación Web básica proporcionada por la herramienta tenemos dos alternativas:

- Crear nuestro propio **Patrón de Presentación**. Esto es posible con la herramienta *OptimalJ Architecture Edition*. Dado un modelo Web, podemos definir el patrón que mejor se adapte a nuestras necesidades. Esta es una buena elección si queremos producir el mismo tipo de interfaz para múltiples aplicaciones. No obstante, la herramienta *OptimalJ Architecture Edition* no se ha probado en este estudio.
- Modificar manualmente la aplicación Web, creando nuevas acciones, servlets y JSPs y modificando los existente para cumplir nuestros requisitos.

Los diseñadores también pueden mejorar la interacción con los formularios de HTML usando JavaScript dentro de los JSPs. OptimalJ ofrece un *plug-in* para Macromedia's Dreamweaver para realizar cambios de diseño, permitiendo a un diseñador Web alterar el aspecto de los componentes Web.

La Figura 30 muestra un ejemplo de fichero JSP generado por OptimalJ. Hay que destacar que los ficheros generados para la capa de presentación también presentan la separación entre bloques libres y bloques protegidos que vimos para los ficheros de la capa de negocio.

```
<html:link page="/ClienteEditFilterFindCliente.do">
    <bean:message key="weblabel.QueryFindCliente"/>
</html:link>


<html:link page="/ClienteNew.do">
    <bean:message key="weblabel.New"/>
</html:link>




</div>


<div id="errormessage" class="errormessage">
<html:errors />


</div>

<div class="content">
<html:form action="ClienteStore.do" name="clienteClienteForm" type="petstore.application.web.ClienteClienteForm" scope="reque
<% if ( clienteClienteBusinessFacade.isChanged() ) {  %>
  <html:submit property="altura.action"><bean:message key="weblabel.Store"/></html:submit>
<% } else { %>
```

Figura 30. Ejemplo de un fichero JSPs generado para la capa de presentación

OptimalJ también incorpora el **servidor de servlets y JSPs *Tomcat***, que junto con el servidor de EJB y la base de datos *Solid* incorporada nos permiten probar la aplicación generada desde el mismo entorno de desarrollo.

6.2 Construcción de la Aplicación Pet Store con OptimalJ

El proceso de desarrollo de una aplicación básica con OptimalJ es extremadamente sencillo. Lo resumiremos brevemente en este apartado.

En primer lugar, debemos construir el modelo de clases de la aplicación (PIM), que ya mostramos anteriormente en la Figura 22. A partir de este modelo de clases, la herramienta genera automáticamente los tres PSMs de la aplicación:

- PSM para la capa Web, mostrado en la Figura 25.
- PSM para EJB, mostrado en la Figura 26.
- PSM para la base de datos, mostrado en la Figura 27.

Si generamos el código directamente e implementamos los métodos de negocio, sin realizar cambios adicionales en estos PSMs, obtendremos una aplicación sencilla que básicamente permite la creación, eliminación, modificación y búsqueda por clave primaria de instancias de las clases que aparecen en el PIM.

Esta aplicación generada por defecto, aunque es útil, dista mucho de ser una aplicación "real". Por ello, hemos enriquecido la aplicación realizando algunas mejoras adicionales, consistentes en:

- Incluir **nuevos métodos de búsqueda**. Por ejemplo, búsqueda de clientes por nombre y apellidos, búsqueda de pedidos por fechas, búsqueda de animales por categorías, etc.
- Traducir la interfaz web generada al **castellano**, ya que el idioma por defecto es el inglés.
- Generar **identificadores únicos** para las nuevas instancias.
- Establecer **restricciones** para algunos atributos. Por ejemplo, chequear el formato del email, el DNI, etc.
- Establecer **roles de seguridad** para permitir distintas acciones a distintos tipos de usuarios. Permitir también la entrada de clientes mediante **login/password** de manera que puedan modificar sus datos personales, consultar y modificar sus pedidos, crear nuevos pedidos, etc.
- Insertar **imágenes de los animales** en la aplicación web.
- Modificar el **esquema de navegación** para hacer más manejable la aplicación.

Muchas de estas mejoras se pueden realizar de manera sencilla mediante los asistentes (*wizards*) que incorpora OptimalJ. Un ejemplo de estos asistentes lo tenemos en la Figura 31, donde puede verse el asistente para la creación de un nuevo método de búsqueda para el componente EJB *Pedido*.

No obstante, otras mejoras requieren modificar el código fuente generado, sobre las mejoras en la capa de presentación. Esto nos obliga a conocer perfectamente la estructura del código generado para saber dónde debemos aplicar el cambio.

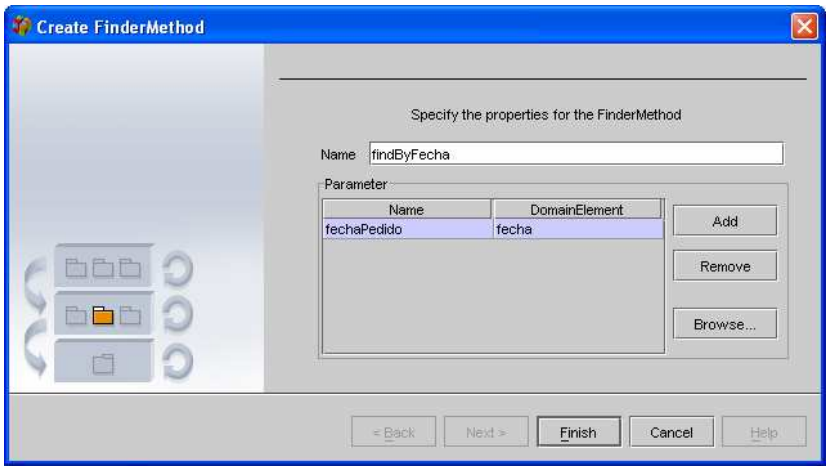


Figura 31. Asistente para la creación de un método de búsqueda de EJB

6.3 Interfaz de la aplicación generada

En este apartado se mostrará la interfaz de usuario de nuestra aplicación, generada por OptimalJ.

La herramienta proporciona una navegación por defecto para cada componente Web, proporcionada por el patrón *HTML Default*. Básicamente esta navegación permite crear, recuperar, modificar y eliminar instancias de cada componente Web, como veremos en los próximos apartados.

Comprobación de seguridad

La primera pantalla de la aplicación, mostrada en la Figura 32, consiste en una comprobación de seguridad donde el usuario debe autenticarse. Esta pantalla se genera automáticamente al crear distintos roles en OptimalJ para el acceso a la aplicación. Para nuestra aplicación sólo existen dos tipos de usuarios:

- **admin**: tiene acceso total a todos los componentes de la aplicación. Su password es también *admin*.

- **user**: tan sólo tiene acceso a los clientes y a los pedidos. Su password es *user*.

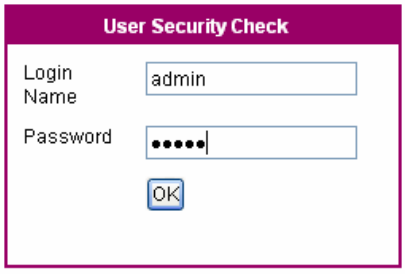


Figura 32. Comprobación de seguridad

Página principal de la aplicación

Tras la autentificación, accedemos a la página principal de nuestra aplicación.

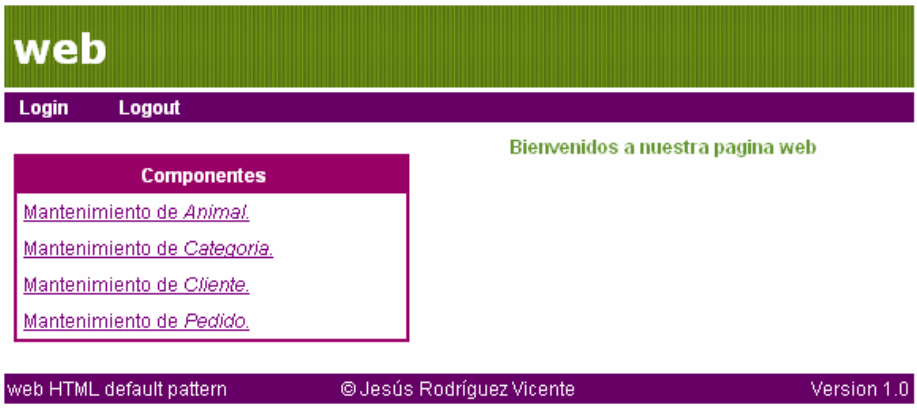


Figura 33. Página principal de la aplicación

La Figura 33 muestra la página principal. Como vemos, aparece la lista de *componentes* de la aplicación, que no son más que los distintos *beans* de tipo entidad definidos en el modelo EJB.

Mantenimiento de Animal

Al seleccionar el mantenimiento de cualquier componente, aparecerá siempre la pantalla de búsqueda por clave primaria del componente, como podemos ver en la Figura 34.

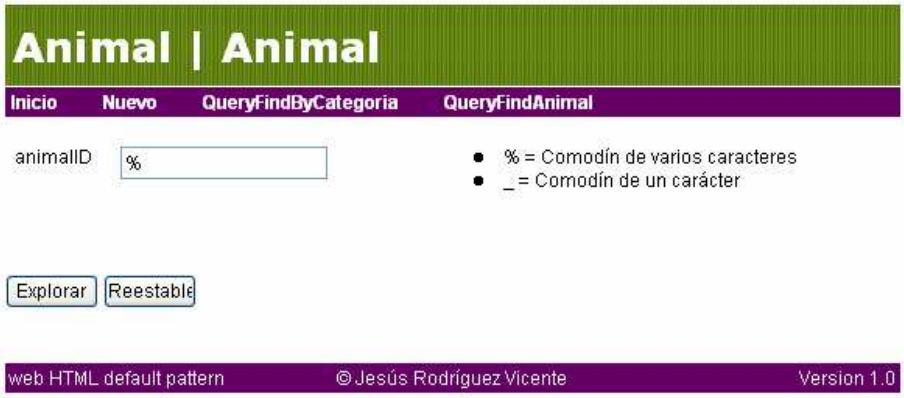


Figura 34. Búsqueda de Animal por clave primaria (animalID)

En la parte superior de la página aparecen los siguientes enlaces:

- **Inicio**: regresa a la página principal de la aplicación.
- **Nuevo**: formulario de creación de una nueva instancia del componente.
- **QueryFindCategoría**: página de búsqueda de animales por categorías.
- **QueryFindAnimal**: página de búsqueda de animales por nombre y/o descripción.

El carácter '%' hace de comodín de varios caracteres, mientras que el carácter '_' se emplea como comodín de un solo carácter. Si pulsamos el botón *Explorar* escribiendo '%' en el campo *animalID*, aparecerá la lista completa de animales registrados, como puede observarse en la Figura 35.

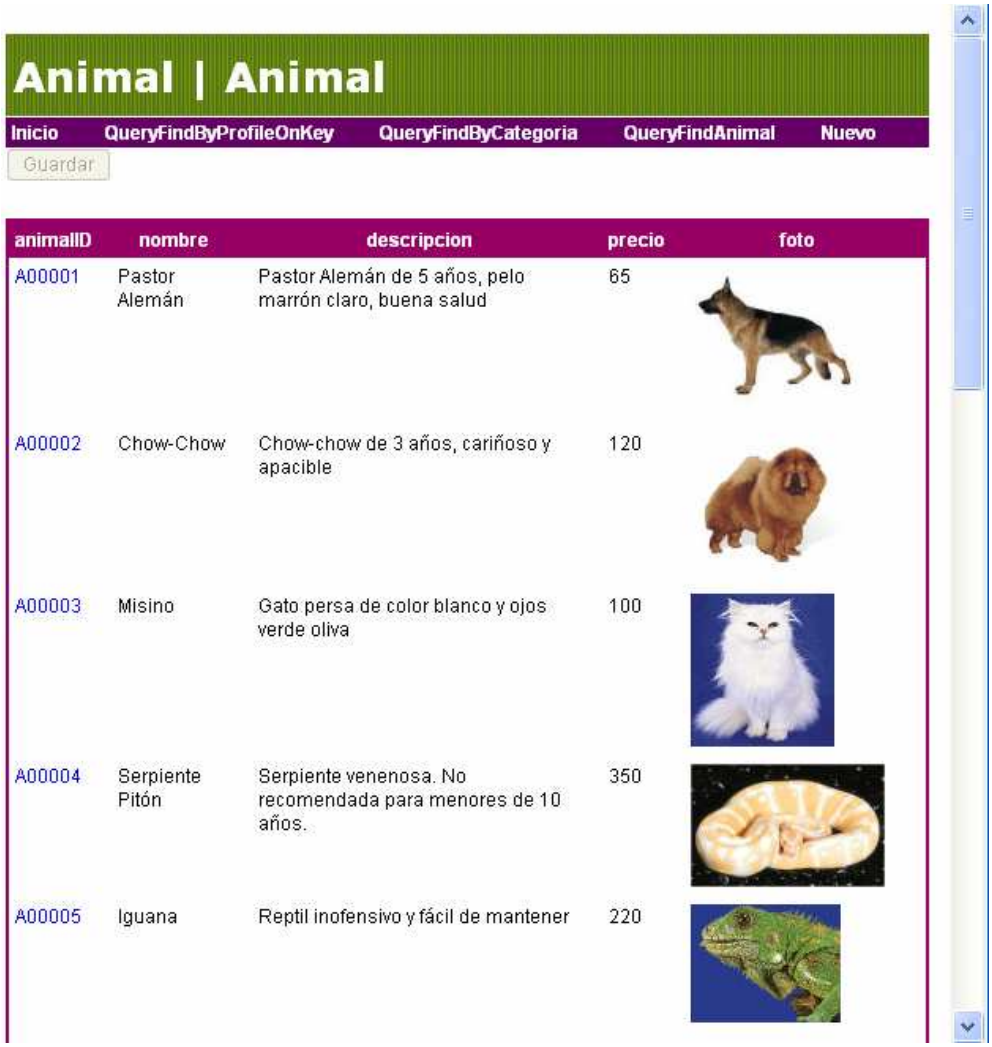


Figura 35. Listado de animales

En esta página podemos ver toda la información de cada animal, y pinchando sobre su identificador podemos modificar sus datos.

Para crear una nueva instancia de Animal pulsaremos el enlace **Nuevo**, apareciendo una página como la mostrada en la Figura 36.

Con el símbolo asterisco '*' se marcan los campos obligatorios. Lógicamente, el identificador **siempre** es obligatorio. Gracias al generador de identificadores únicos implementado, este campo de identificador se rellena automáticamente y no es necesario modificarlo.

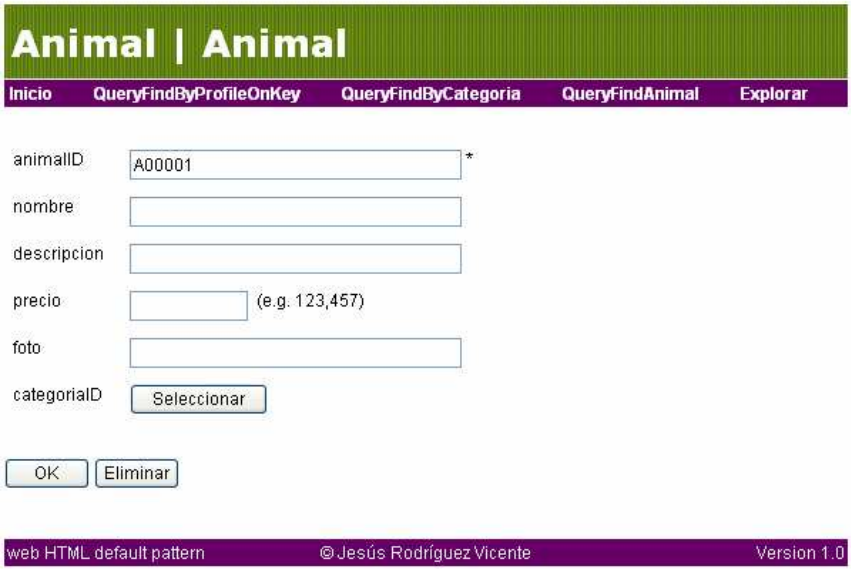


Figura 36. Creación de una instancia de Animal.

Vemos como para ciertos tipos de campos, como los **campos numéricos** como el *precio*, aparece a la derecha un ejemplo de uso. También aparecerá un ejemplo de uso en los campos de tipo **fecha**.

Por último, aparece un botón para seleccionar la categoría a la que pertenece el animal, derivado de la relación entre las clases *Animal* y *Categoría* del Modelo del Dominio.

Las páginas de búsqueda, correspondientes a los *Finder Methods* definidos en el modelo EJB, tienen siempre el mismo formato, similar al de la Figura 37, con un campo editable por cada parámetro del método de búsqueda definido en el modelo EJB. Vemos que para los campos de tipo cadena de caracteres podemos siempre usar los caracteres comodín '%' (comodín de varios caracteres) y '_' (comodín de un sólo carácter).

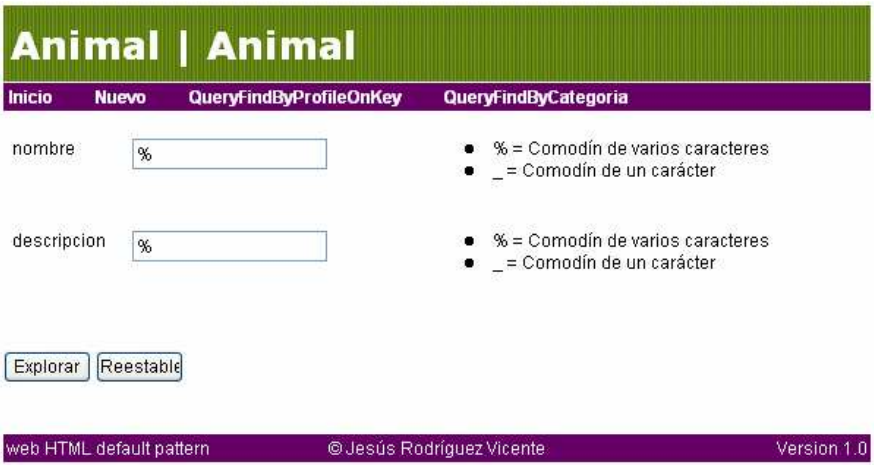


Figura 37. Búsqueda de animales por nombre y/o descripción

Mantenimiento de Categoría

Al seleccionar la opción *Mantenimiento de Categoría* en la página principal, aparecerá la página de búsqueda de categorías por clave primaria, de manera similar a como ocurría con el *Mantenimiento de Animal*.

Al pulsar el botón *Explorar* aparece la lista completa de categorías, con la información de cada categoría, un enlace para modificar cada categoría y un enlace a la lista de animales de cada categoría (hipervínculo **[Ver Animales]**). En la Figura 38 podemos ver la página con el listado de categorías.

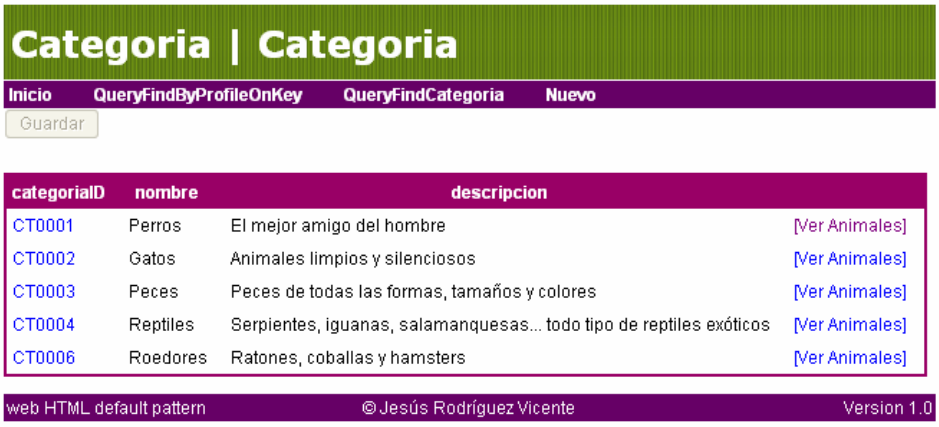


Figura 38. Listado de Categorías

La pantalla de creación de nuevas categorías es muy sencilla y se muestra en la Figura 39.

Figura 39. Creción de una instancia de Categoría

Por último, disponemos de una página de búsqueda por nombre y/o descripción (**QueryFindCategoría**) idéntica a la de búsqueda de animales de la Figura 37.

Mantenimiento de Cliente

El mantenimiento de clientes es muy similar a los dos anteriores. A las pantallas típicas de creación, modificación y búsqueda por clave primaria hay que añadir la búsqueda por nombre, nif y/o email (**QueryFindCliente**).

La Figura 40 muestra la página de creación de un nuevo usuario. A continuación, en la Figura 41 podemos ver el listado de clientes.

Figura 40. Creación de una instancia de Cliente.

Figura 41. Listado de Clientes

En la Figura 42 observamos cómo junto a los datos del cliente aparece también los encargos realizados por el cliente, pudiendo suprimirlos fácilmente desde la misma página.

Figura 42. Modificación de la información de un cliente

Mantenimiento de Pedido

El mantenimiento de pedidos tiene algunos elementos que no aparecen en los anteriores, así que nos detendremos un poco más aquí.

En primer lugar examinaremos la página de creación de pedidos de la Figura 43.

Podemos observar como el campo *fecha* se autocompleta con la fecha actual del sistema. Otro aspecto a destacar es que los tipos enumerados definidos en el modelo del dominio los muestra como elementos de tipo *choice*, que permiten elegir sólo entre los valores enumerados definidos (campos *tipoPago*, *estado* o *tipoTarjeta*).

Por otro lado, al introducir un nuevo pedido debemos seleccionar el cliente al que pertenece el pedido.

Por último, debemos especificar las distintas líneas del pedido pulsando el botón *Crear*. Aparecerá una página como la de la Figura 44.

Aquí deberemos seleccionar el animal que queremos adquirir, pulsando el botón *Seleccionar*, lo que nos llevará al listado de **todos los animales**. Iremos introduciendo las líneas de pedido una a una hasta completar el pedido.

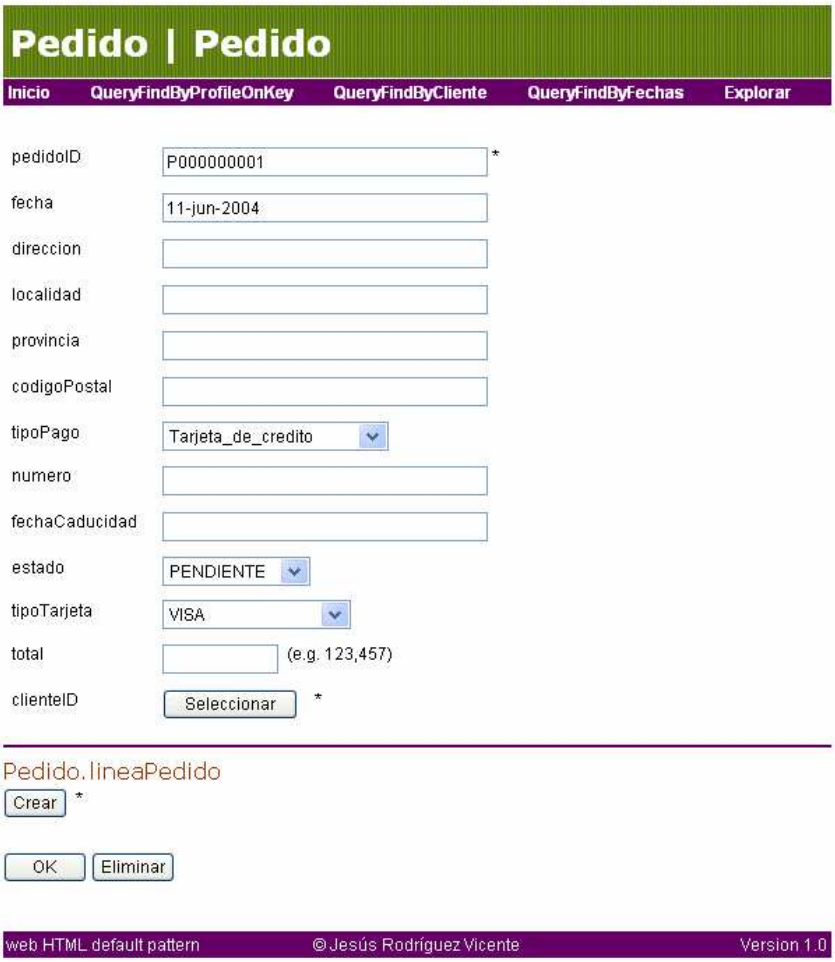


Figura 43. Creación de una instancia de Pedido.

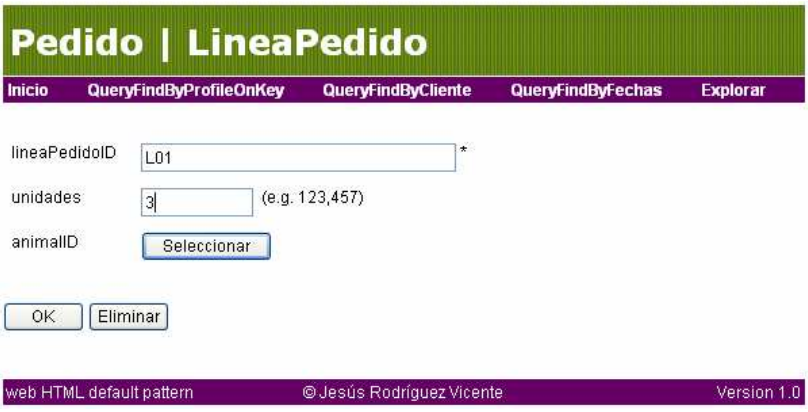


Figura 44. Creación de una linea de pedido.

6.4 Evaluación de OptimalJ

En este apartado evaluaremos la herramienta OptimalJ según los criterios expuestos en el apartado Criterios para Evaluar Herramientas MDA, (pág. 32). La siguiente tabla muestra las puntuaciones asignadas para cada una de las propiedades objeto de evaluación:

Id	Propiedad	Puntuación	Comentarios
P01	Soporte para PIMs	4	OptimalJ posee un fuerte soporte para especificar sistemas mediante PIMs, convirtiéndose estos en la base de todo el desarrollo.
P02	Soporte para PSMs	3	Soporte medio para los tres tipos principales de PSMs (Base de datos, EJB y Web). El modelo EJB es bastante flexible, pero el modelo Web es todavía, en nuestra opinión, muy limitado.
P03	Permite varias implementaciones	0	El programa permite la generación de varios PSMs a partir del mismo PIM, pero todos van dirigidos hacia la misma plataforma (J2EE). Además, la herramienta no permite añadir nuevas implementaciones a las disponibles.
P04	Integración de Modelos	4	Los distintos PSMs se integran perfectamente entre sí de forma transparente y automática, interaccionando sin problemas todas las capas entre sí. Además, permite comunicar nuestra aplicación con aplicaciones CORBA o con Servicios Web, aunque esta funcionalidad no ha sido probada.
P05	Interoperabilidad	4	OptimalJ puede exportar e importar modelos vía XMI de manera sencilla.
P06	Acceso a la definición de las transformaciones	2	En *OptimalJ Proffessional Edition* no se permite al usuario modificar las definiciones de transformaciones. Sin embargo, podemos tener acceso a los "patrones de transformación" usando otra herramienta, *OptimalJ Architecture Edition*.
P07	Verificador de modelos	3	La herramienta incluye buenos verificadores de modelos, tanto para el PIM como para cada uno de los distintos PSMs, que son ejecutados antes de intentar una transformación. Estos verificadores previene al desarrollador de posibles fallos potenciales en la generación de PSMs o de código.
P08	Expresividad de los modelos	2	Buena representación del PIM. Con respecto a los PSMs, especialmente la capa de presentación (Web), no posee la expresividad que cabría esperar, dando poca flexibilidad al desarrollador a la hora de describir la interfaz de la aplicación.
P09	Uso de patrones	4	OptimalJ hace uso de múltiples patrones, tanto para la transformación entre modelos como para la generación del código fuente. Con la herramienta *OptimalJ Architecture Edition* pueden definirse nuevos patrones y usarlos en nuestro proyecto.
P10	Soporte para la regeneración de modelos	4	La herramienta gestiona de manera excelente la regeneración de modelos, tanto a nivel de PSMs como de código. A nivel de modelo, un cambio en el PIM se traslada a los PSMs automáticamente y viceversa. A nivel de código, la separación entre *bloques libres* y *bloques reservados* permiten al desarrollador conservar los cambios realizados "manualmente" en el código aunque se regenere toda la aplicación.
P11	Transformaciones intra-modelo	2	En general, una vez construidos los PSMs, un cambio en un PSM no afecta al resto de PSMs. Sólo algunos cambios hechos en un PSM se reflejan en otro, por ejemplo, los cambios en el modelo de base de datos se reflejan en el modelo EJB.
P12	Trazabilidad	3	OptimalJ permite conocer para cualquier elemento del PSM su origen en el PIM y su destino en el código

			generado.
P13	Ciclo de vida	3	La herramienta engloba casi todas las fases del ciclo de vida, abarcando análisis, diseño, codificación, depuración, prueba, despliegue y mantenimiento. No posee soporte para la fase de recogida de requisitos.
P14	Estandarización	4	OptimalJ usa los principales estándares relativos a MDA. Usa UML como lenguaje de modelado y puede importar/exportar modelos vía XMI. Almacena también todos sus modelos en un repositorio MOF.
P15	Control y refinamiento de las transformaciones	2	Aunque de manera limitada, la herramienta permite refinar ciertos aspectos de las transformaciones. Por ejemplo, permite especificar en el PIM el tamaño que tendrán los atributos en la base de datos.
P16	Calidad del código generado	3	El código generado por la herramienta está bien documentado e incorpora buenos patrones de código. No obstante, esos mismos patrones de código empeoran la legibilidad del código y dificultan su modificación. El desarrollador debe tener un alto conocimiento de los patrones empleados en la generación para adaptar el código a sus requisitos.
P17	Herramientas de soporte	3	Aparte del editor de modelos, OptimalJ incorpora un completo entorno IDE para editar los ficheros fuente. Incluye también un servidor de bases de datos (SolidDB), un servidor de EJB (JBoss) y otro de servlets/JSPs (Tomcat) para realizar las pruebas.

7 La Herramienta ArcStyler

ArcStyler de *Interactive Objects* [12] *e*s otra de las herramientas que implementa MDA, permitiendo desarrollar software de manera efectiva y eficiente mediante el uso de modelos.

Detallaremos en los próximos apartados la arquitectura de esta herramienta, y veremos cómo hemos construido nuestro *Pet Store* con ella. Finalmente realizaremos una evaluación de ArcStyler según los criterios expuestos en el apartado 5.

La versión usada para realizar este estudio es ***ArcStyler Versión 4.0.90***.

7.1 Arquitectura de ArcStyler

Introducción

ArcStyler permite la transformación de modelos a otros modelos, a código fuente y a ficheros relacionados con las pruebas o el despliegue, así como ficheros de proyecto para editores de código como JBuilder. Las transformaciones en ArcStyler están ligadas a los llamados *MDA-Cartridges* que pueden descargarse de Internet y "conectarse" como *plugin* en cualquier instalación de la herramienta. ArcStyler también soporta la creación, edición y mantenimiento de estos *MDA-Cartridges* como una de sus principales características, permitiéndonos definir nuestras propias reglas de transformación de modelos. Actualmente existen numerosos *MDA-Cartridges* para diferentes arquitecturas o plataformas, como J2EE, EJB, .NET, aplicaciones Web, etc.

ArcStyler permite definir y administrar potentes transformaciones de modelos y también permite controlar el proceso completo de desarrollo de software mediante múltiples perspectivas integradas dentro de la misma herramienta.

Para lograr su objetivo, los MDA-Cartridges deben poder acceder a los modelos. ArcStyler permite el acceso a los repositorios de modelos mediante el uso del estándar *Java Metadata Interface* (JMI) y, para mantener la compatibilidad con versiones anteriores de ArcStyler, también usa el llamado API C-MOD (una variante de UML 1.3 con interfaces de acceso similares a JMI). La Figura 45 muestra cómo ArcStyler une todos estos componentes, proporcionando de manera transparente un entorno integrado y abierto de MDA.

ArcStyler integra numerosas herramientas en el mismo entorno de desarrollo, incluidas:

- Una completa herramienta de modelado UML.
- Una herramienta de ingeniería inversa que transforma ficheros fuente de Java y ficheros JAR/EAR en modelos UML.
- Un explorador genérico basado en MOF que puede usarse para modelos de cualquier metamodelo.

- Una herramienta ANT[12] para administrar reglas de construcción y despliegue.

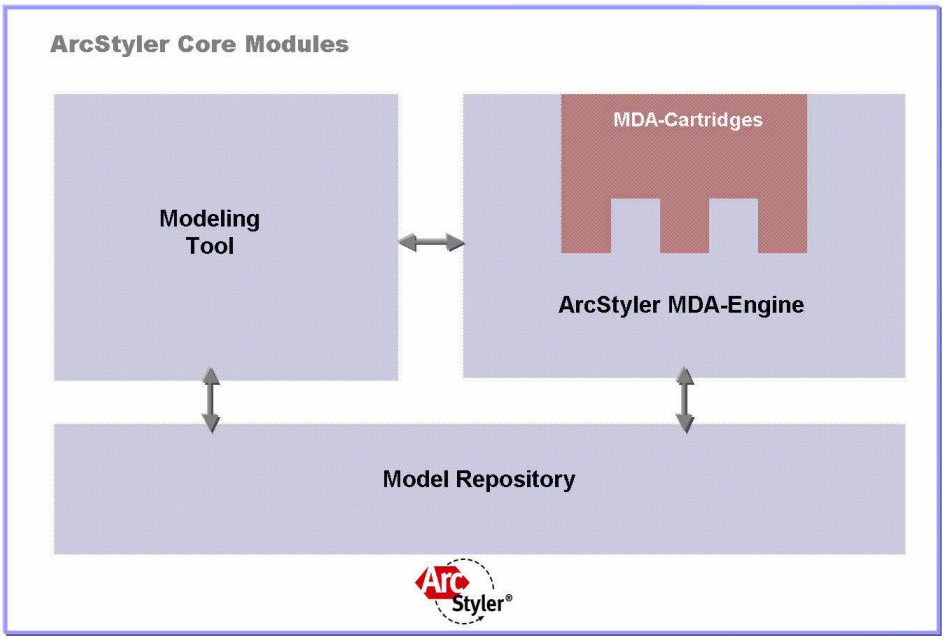


Figura 45. Módulos principales de ArcStyler, extraido de [13]

Un dato muy interesante es que podemos añadir nuestras propias herramientas al entorno ArcStyler siguiendo la guía *MDA-Cartridge Development and Extensibility Guide* [18]. De este modo podemos, por ejemplo, incluir nuestra herramienta favorita de modelado, un editor de código, etc.

MDA en ArcStyler

En este apartado explicaremos brevemente cómo los estándares involucrados en MDA y otros estándares son utilizados directa o indirectamente en ArcStyler.

UML

Como ya sabemos, **UML** es el lenguaje estándar para la construcción de modelos en MDA. ArcStyler incorpora una completa herramienta de modelado conforme a UML 1.4, incorporada en el entorno de ArcStyler La herramienta de UML incorporada en ArcStyler soporta la última versión de UML, la versión 1.4.

En la Figura 46 puede verse el diagrama de componentes para el ejemplo del *Pet Store* en esta herramienta de modelado.

[12] Apache Ant es una herramienta basada en Java para la construcción de programas. Parecida a Make, pero mejorando muchos aspectos, está escrita pensando en el desarrollo multiplataforma, por lo que los ficheros de configuración no contienen comandos de shell sino que tienen formato XML.

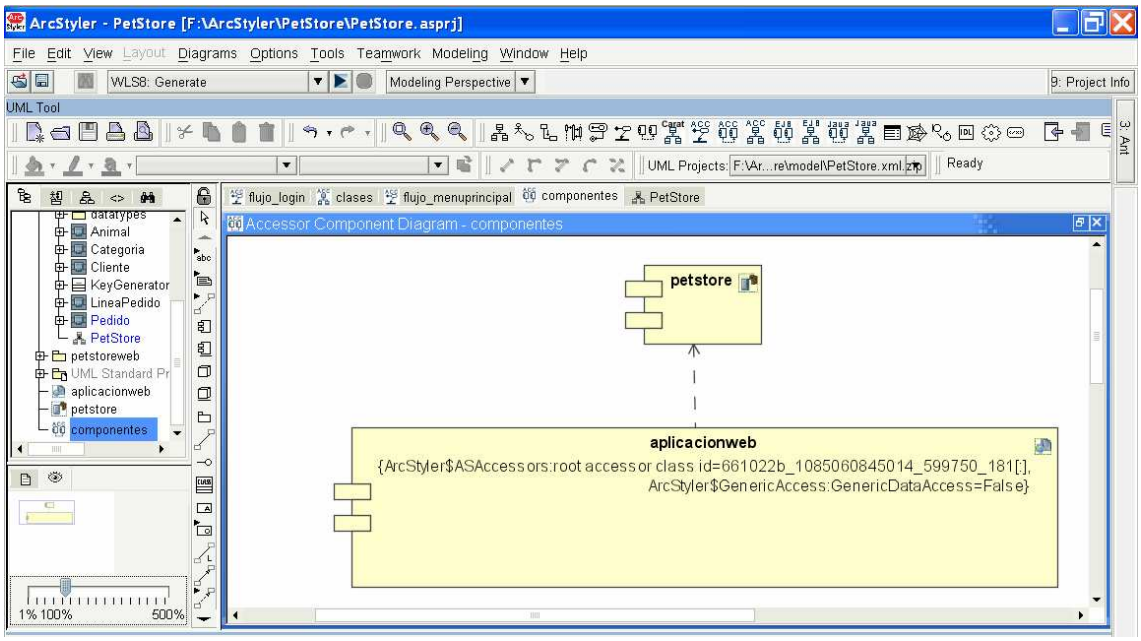


Figura 46. Herramienta de Modelado de ArcStyler

MDA Profiles

ArcStyler usa el mecanismo de *Perfiles UML* descrito en el apartado Perfiles UML para proporcionar extensiones estándar que soporten el modelado de MDA. Estas extensiones se conocen como ***MDA profiles***. Un *MDA profile* de ArcSyler consta de los siguientes elementos de UML:

- Estereotipos.
- Tipos de datos.
- Marcas.

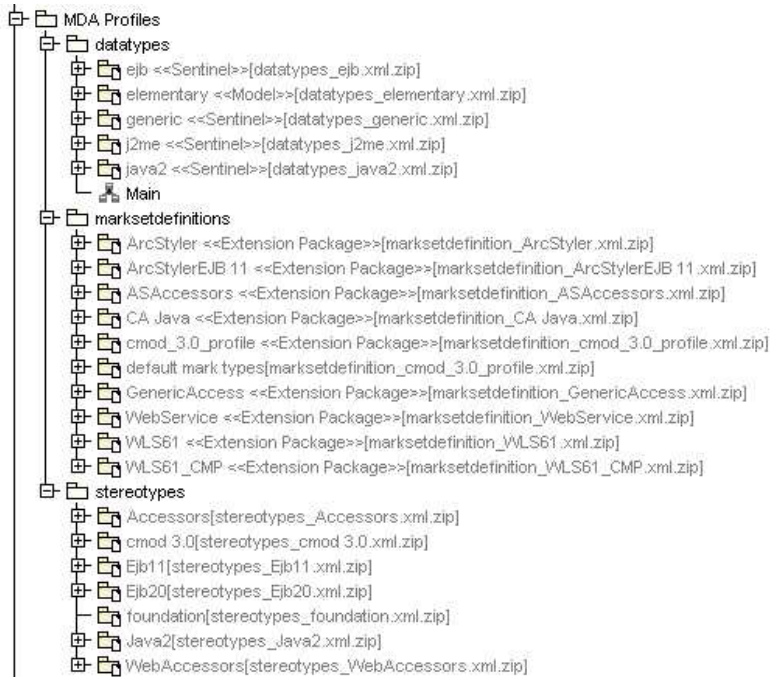


Figura 47. MDA Profiles en ArcSyler

Como veremos mas adelante, las marcas son básicamente valores etiquetados de UML junto con características adicionales, que pueden ser agrupados en conjuntos predefinidos y proporcionar valores por defecto (algo no posible con los valores etiquetados).

Los *MDA profiles* se almacenan siempre en un paquete especial del nivel superior en el modelo UML. Los estereotipos, tipos de datos y marcas definidos en un *profile* pueden aplicarse a un elemento del modelo para dar soporte a una plataforma específica. La Figura 47 muestra este paquete especial con algunos *MDA profiles* de ArcStyler. Podemos ver en la figura *profiles* para EJB, Accessor o Java2.

Cada *MDA Cartridge* incluye la definición de *MDA profiles* para permitir incluir en los modelos los aspectos específicos de la plataforma.

MOF y XMI

ArcStyler usa **MOF** internamente para dar soporte, entre otros, a UML, XMI y JMI.

Como vimos en el apartado MOF (pág. 18), **XMI** es la parte del estándar MOF que define cómo representar los modelos de un determinado metamodelo usando XML. ArcStyler soporta las versiones de XMI 1.0 y 1.1. Podemos usar este soporte para importar modelos existentes en formato XMI, o para exportar modelos creados en ArcStyler de cara a usarlos en otras herramientas con soporte para XMI.

JMI

El *Java Metadata Interface* o JMI es un estándar surgido del *Java Community Process* (JCP) de Sun. De manera similar a cómo XMI define el intercambio de modelos, JMI define la interfaz de acceso a esos modelos y a sus metadatos usando Java. Esto incluye la navegación por el repositorio de modelos, creación y eliminación de elementos del modelo y la inspección de un tipo de elemento del modelo mediante reflexión.

ArcStyler se basa íntegramente en JMI. Es la base común para el acceso a cualquier modelo a través de la herramienta. El principal beneficio de esto es que un gran número de repositorios son conformes a JMI, con lo que sólo es necesario un pequeño esfuerzo para integrar esos repositorios en ArcStyler. Es más, otras herramientas también basadas en interfaces JMI pueden integrarse fácilmente en ArcStyler.

Marcas MDA

Para hacer que los modelos mantengan su valor con el paso del tiempo, lo mejor es mantenerlos independientes de cualquier plataforma. Esto hace posible portar más adelante estos modelos a otras plataformas con poco esfuerzo.

Entonces, ¿cómo añadimos la información específica de una plataforma sin ligar el modelo a dicha plataforma? La respuesta está en el concepto de las ***Marcas MDA*** o ***MDA Marks***. Se trata de anotaciones asociadas a los elementos del modelo que contienen cualquier información útil que sirve como entrada a los *MDA-Cartridges* (los veremos más adelante). Estas marcas son las que contienen la información específica de

la plataforma. Como se pueden adherir o separar de un modelo fácilmente, no "contaminan" el modelo con especificaciones dependientes de una plataforma. En lugar de esto, pueden coexistir conjuntos separados de marcas en un mismo modelo, haciendo posible transformar el modelo a diferentes plataformas sin tener que cambiar el modelo mismo.

La Figura 48 muestra el cuadro de diálogo típico para introducir marcas para un elemento del modelo, en este caso para una clase. Aquí podemos establecer marcas para todos los MDA-Cartridges usados en nuestro proyecto. Vemos como para la plataforma EJB podemos dirigir el proceso de transformación estableciendo las marcas adecuadas, por ejemplo:

- *BeanType*: permite establecer el tipo de bean (*Session* o *Entity*).
- *GenLocalInterfaces*: indica se generarán las interfaces locales además de las remotas.
- *PersistenceManagement*: establece el tipo de persistencia (*CMP*, *BMP* o la que disponga el MDA-Cartridge por defecto).
- *StateManagement*: indica si el *bean* va a ser con estado o sin estado (*Stateful* o *Stateless*).

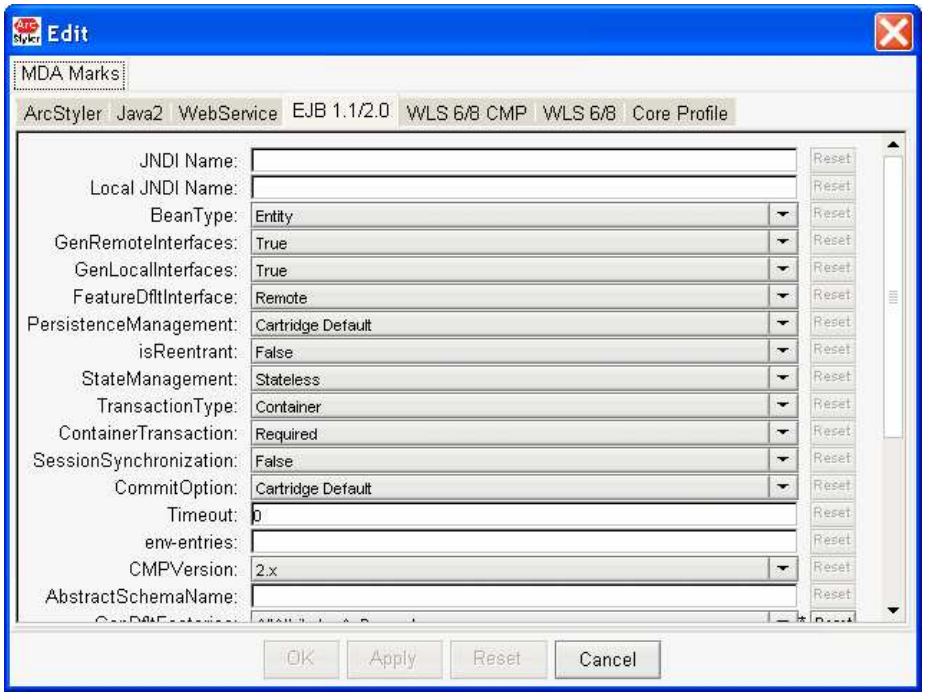


Figura 48. Cuadro de diálogo para la introducción de marcas MDA

Muchas herramientas usan valores etiquetados para implementar las marcas. Sin embargo, esta aproximación tiene serios inconvenientes. Los valores etiquetados de UML no soportan enumerados, no permiten valores por defecto, y la mayoría de herramientas UML no permiten añadir un conjunto de valores etiquetados a un elemento del modelo de forma automática. Es más, el soporte para valores etiquetados ha cambiado sustancialmente con cada nueva versión de UML (1.3, 1.4, 2.0), dificultando la labor del desarrollador.

Por ello, ArcStyler define las marcas independientemente de su representación en UML. Cualquier *MDA-Cartridge* proporciona una definición de los conjuntos de marcas que

espera para un modelo de entrada. La definición del conjunto de marcas proporciona, para cada marca, la siguiente información:

- Tipo de dato asociado.
- Elementos del modelo a los que se aplica la marca.
- Valor por defecto.

Un editor inteligente de conjuntos de marcas evalúa las definiciones del conjunto de marcas en tiempo de modelado y proporciona el soporte apropiado para marcar los modelos para una plataforma específica.

Conviene resaltar que este tipo de transformación guiado por marcas corresponde al enfoque de **transformación de instancias** que vimos en el apartado Definiciones de Transformación (pág. 23). ArcStyler utiliza conjuntamente los enfoques de transformación de tipos (*model type mapping*) y transformación de instancias (*model instance mapping*) definidos en [21] para llevar a cabo sus transformaciones.

Funciones de transformación de modelos

Las **funciones de transformación** que transforman modelos en otros modelos o en código fuente constituyen el núcleo de MDA. Una función de transformación puede usar uno o más modelos como entrada y producir un modelo de salida. Un tipo especial de "modelo" de salida es un conjunto de ficheros de texto que cumplen un conjunto estricto de reglas, por ejemplo, la definición sintáctica de un lenguaje o el DTD de alguna definición de datos basada en XML. Los principales ficheros de salida serán, lógicamente, ficheros de código fuente en algún lenguaje de programación. Otros ficheros de salida podrían ser, por ejemplo, ficheros de despliegue para la aplicación, scripts de bases de datos, ficheros de interfaces Web (JSPs, HTML), etc.

Además de los modelos de entrada, las funciones de transformación usan **marcas** como entrada, como vimos en la sección anterior. Estas marcas mantienen información para dirigir la transformación del modelo a una plataforma específica. Dicha información no debería formar parte de ninguno de los modelos de entrada, ya que lo haría específico de una plataforma.

ArcStyler usa los *MDA-Cartridges* como componentes desplegables que encapsulan una función de transformación. La siguiente sección da una visión general de qué son estos "cartuchos", cómo pueden usarse y cómo pueden ser creados.

MDA-Cartridges

Un ***Cartucho MDA*** o ***MDA-Cartridge*** contiene las reglas necesarias para realizar una transformación de modelos. Puede instalarse como *plugin* en cualquier versión de ArcStyler, descargarse de Internet, y editarse o extenderse si es necesario. Los *MDA-Cartridges* opcionalmente pueden incorporan verificadores de modelo que aseguran que los modelos de entrada son válidos para realizar la transformación implementada por el *cartridge*.

Además de la tranformación y la verificación de modelos, un *MDA-Cartridge* puede incorporar asistentes o *wizards* que se integran en la herramienta de modelado de ArcStyler. Por ejemplo, el cartucho para EJB proporciona un asistente que facilita la creación de un elemento del modelo que representa a un componente EJB.

ArcStyler define una completa arquitectura para la definición de cartuchos llamada ***CARAT***: *The CARtridge ArchiTecture*. CARAT se basa en la idea de aplicar MDA a la creación de funciones de transformación. En otras palabras, ArcStyler usa modelos para especificar reglas de transformación.

ArcStyler nos permite elegir de entre un extenso conjunto de *MDA-Cartridges*, disponibles para descargar e instalar en ArcStyler. La Figura 49 muestra el amplio abanico de MDA-Cartridges disponibles para ArcStyler. Como vemos, existen *cartuchos* para plataformas como CORBA, .NET, EJB o Java2.

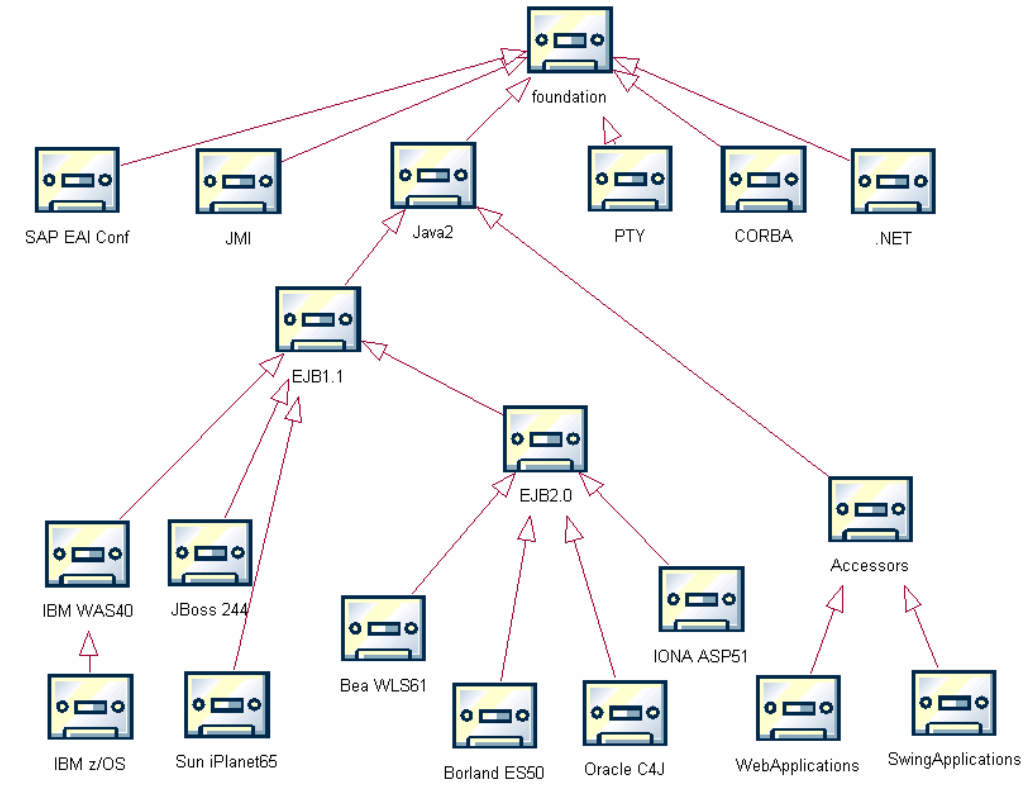


Figura 49. Jerarquía de MDA-Cartridges

Si necesitamos modificar reglas de transformación, ArcStyler ofrece un completo soporte para el desarrollo de *MDA-Cartridges*. Podemos de este modo usar el potente concepto de herencia de *MDA-Cartridges* para obtener un "hijo" de un *MDA-Cartridge* existente, y redefinir o cambiar las reglas de transformación que nos interesen. Por supuesto, si ninguno de los *MDA-Cartridges* satisface nuestros requisitos, también podemos crear nuestros *MDA-Cartridges* desde cero.

Vemos que todos los *MDA-Cartridges* de la Figura 49 utilizan este concepto de herencia para extender otros *cartridges*. Por ejemplo, el *cartridge* para EJB1.1 hereda

del *cartridge* para Java2, y el de Java2 hereda del *cartridge foundation*, raiz de toda la jerarquía.

Framework *Accessor*

El framework *Accessor* de ArcStyler [16] define un proceso bien definido para elaborar la capa de presentación de las aplicaciones. En el modelo Accessor se define el flujo de control y el flujo de datos de la capa de presentación de un modo abstracto. Esto incluye mostrar información, leer e interpretar las acciones del usuario, invocar la capa de negocio, etc. En el proceso de generación de código, el modelo Accessor se implementa para una tecnología determinada. En el caso del *cartridge WebAccessor* usado para el *Pet Store*, se generarán ficheros JSP y Java servlets.

Un modelo *Accessor* es una instancia del **metamodelo *Accessor***, extensión del metamodelo de UML. El metamodelo *Accessor* resuelve los aspectos de *Vista* y *Control* del paradigma *Modelo-Vista-Control* (MVC). Este paradigma se usa para modelar las interacciones de la capa de presentación con la lógica de negocio, que representa la parte de *Modelo* del concepto MVC. El metamodelo *Accessor* define un número de abstracciones clave que describiremos brevemente en las próximas líneas.

Un ***Accessor*** representa un *Controlador* en la arquitectura MVC, y describe el flujo de control de una interfaz externa. Este comportamiento se modela en un diagrama de estados de UML, como veremos un poco más adelante.

Un ***Representer*** se usa para describir elementos básicos de la interfaz de usuario, es decir, desempeña el rol de *Vista* en la arquitectura MVC. Representan, por ejemplo, páginas JSP o ASP.NET. Los *Representers* se modelan mediante hojas de propiedades especiales proporcionadas por ArcStyler. Un ejemplo de estas hojas de propiedades puede verse en la Figura 50.

Tanto los *Accessors* como los *Representers* son especializaciones de clases UML, así que pueden contener atributos y métodos y asociarse con otras clases.

Un **componente de aplicación Web** representa la unidad desplegable de la aplicación Web. En la tecnología JSP, corresponde a un fichero WAR que empaqueta los componentes *Accessors* y *Representers* de la aplicación.

La tarea principal del modelado de *Accessors* consiste en modelar el comportamiento dinámico o flujo de control. Este flujo de control se modela mediante un **diagrama de estados UML** con algunas extensiones. Cada *Accessor* tiene su propio diagrama de estados. Este diagrama de estados extendido consta de algunas especializaciones de un *estado* UML:

- ***RepresenterState***: permite asociar *Representers* (vistas) al estado, de manera que se muestren cuando se alcance dicho estado.
- ***EmbeddedAccessorState***: describe un estado especial en el diagrama de estados compuesto por varios *accessors*. Una transición a un *EmbeddedAccessorState* inicializa y activa el *accessor* subordinado.

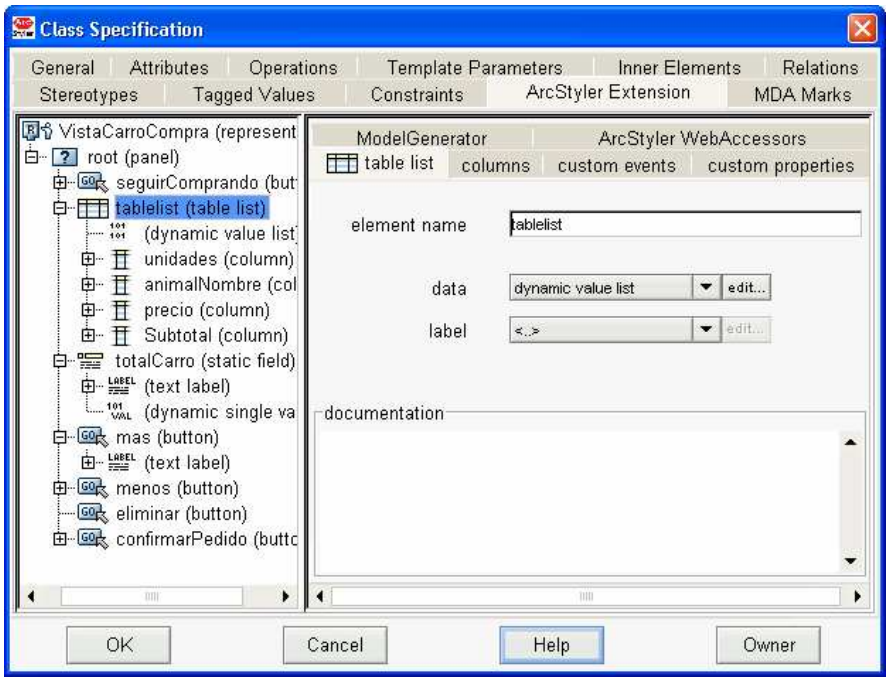


Figura 50. Hoja de propiedades especial para definir *Representers*

Un ejemplo de estos diagramas de estado para el *Pet Store* se muestra en la Figura 51, donde se describe el comportamiento de un *accessor* que maneja la entrada de un usuario en el sistema y el registro de nuevos usuarios. En el diagrama podemos ver tres *RepresenterStates* representando las tres vistas usadas por el *accessor*, *mostrarVistaLogin*, *mostrarVistaNuevoUsuario* y *mostrarVistaNuevoUsuarioOK*. Cada uno de estos estados llevará asociada un *representer*, que se mostrará al alcanzarse el estado. El *representer* asociado al estado *mostrarVistaNuevoUsuario* puede verse en la Figura 52.

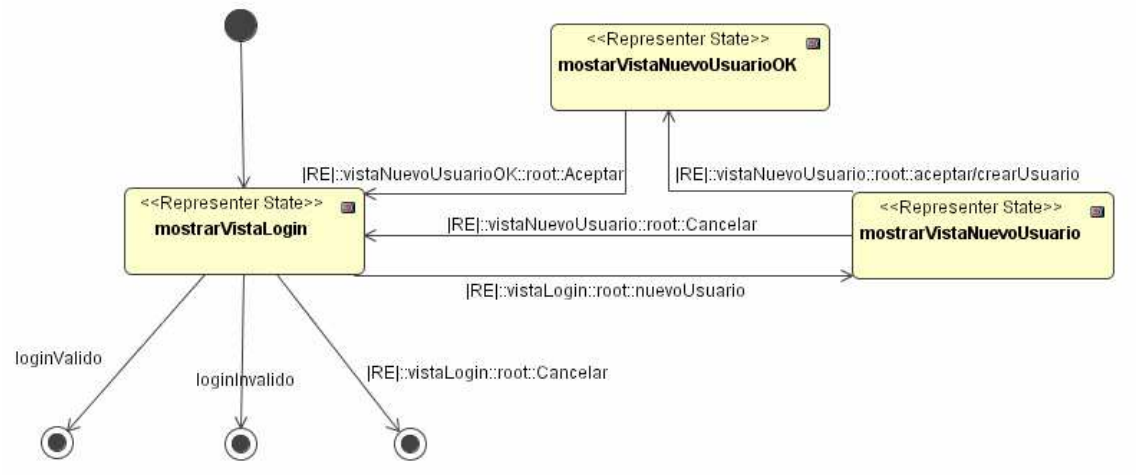


Figura 51. Diagrama de estado para un *Accessor*

Las transiciones entre estados pueden asociarse a eventos de la interfaz, como pinchar un botón, seleccionar un elemento de una lista desplegable, etc. Por ejemplo, la transición etiquetada con `|RE|::vistaLogin::root::nuevoUsuario` se producirá cuando

el usuario presione el botón *Registro* del *representer* de la Figura 52, mostrando entonces la pantalla de registro de nuevo usuario.

También pueden definirse acciones a realizar por el *accessor* ante un evento determinado, que se implementarán una vez generado el código fuente. Por ejemplo, la acción *crearUsuario* asociada a la transición `|RE|::vistaNuevoUsuario::root::aceptar` se encargará de añadir el nuevo usuario a la base de datos una vez rellenado el formulario de registro.

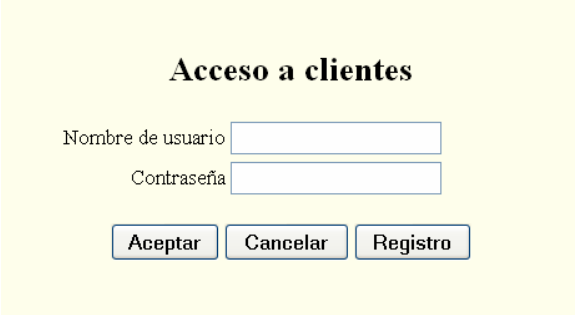


Figura 52. Representer para el acceso al sistema

Este framework es realmente interesante y en nuestra opinión es uno de los motivos por los que *ArcStyler* está por encima de *OptimalJ*, pues permite definir con total flexibilidad interfaces de usuario, algo de lo que carece *OptimalJ*.

7.2 Construcción de la aplicación Pet Store con ArcStyler

En este apartado explicaremos a grandes rasgos los pasos seguidos para construir la aplicación *Pet Store* con ArcStyler, indicando los *MDA-Cartridges* utilizados, tipos de diagramas y otros aspectos generales del desarrollo de nuestra aplicación.

Esta sección no pretende ser una guía de uso de la herramienta, simplemente dar una visión general de cómo se construye una aplicación de este tipo en ArcStyler. Para más información, consúltese el manual de usuario de ArcStyler [14].

Modelo EJB

Para generar la capa EJB para nuestro *Pet Store*, hemos usado el *MDA-Cartridge* para el servidor de EJB *Bea Weblogic 8.1* (**WSL8**), el cual hereda del MDA-Catridge para EJB 2.0 como se vió en la Figura 49.

ArcStyler proporciona un nuevo patrón para modelar EJBs llamado *Compact Bean*. Con el patrón *Compact Bean* todas las propiedades relativas a EJB se modelan en una única clase UML, por ejemplo, las propiedades de la interfaz home, de la interfaz remota y de la clase implementación. Un *Compact Bean* se modela como una clase UML con el estereotipo `<<ComponentSegment>>`. Para cada *Compact Bean* se generan los siguientes elementos:

- Interfaz remota del componente (opcional).

- Interfaz local del componente (opcional).
- Interfaz *Home* remota (opcional).
- Interfaz *Home* local (opcional).
- Clase implementación del Bean (obligatorio).
- Información estándar de despliegue (obligatorio).
- *Key class* (sólo para beans con *container-managed persistence* y clave primaria compuesta).

Para construir el modelo EJB debemos seguir una serie de pasos. En primer lugar hemos construido el diagrama de clases de la Figura 53 con la herramienta UML de ArcStyler.

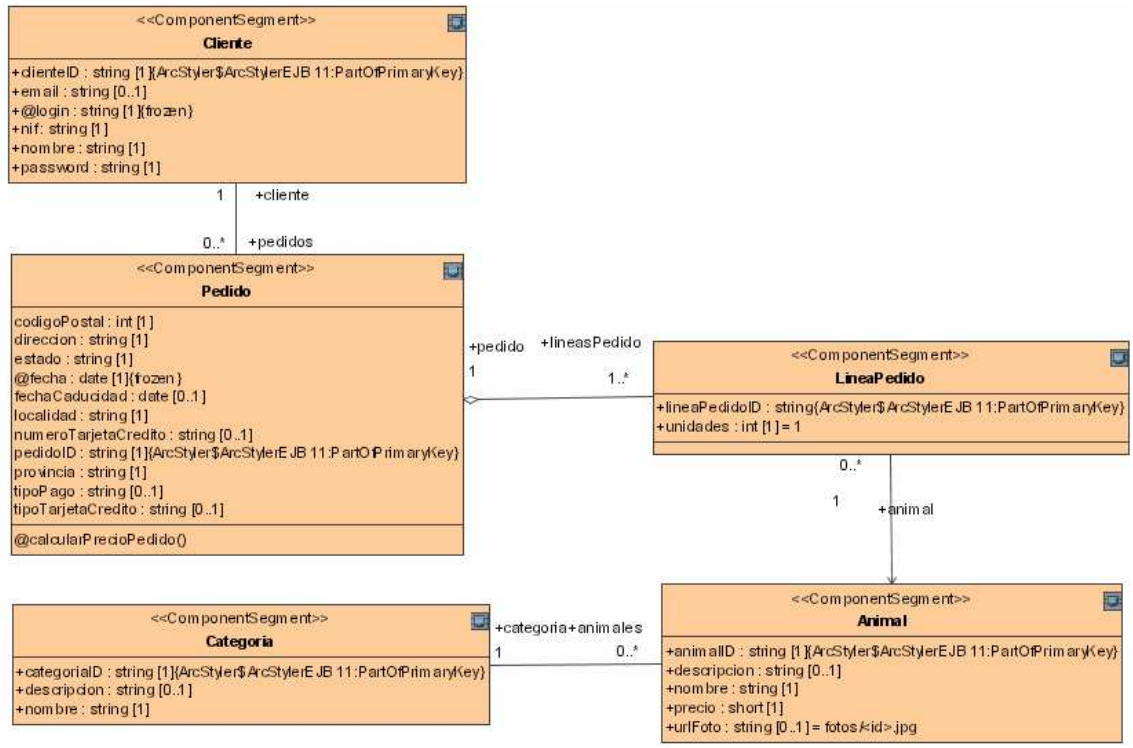


Figura 53. PIM del Pet Store en ArcStyler

Para cada clase que vaya a convertirse en un EJB, como ya hemos dicho, debemos asignarle el estereotipo `<<ComponentSegment>>`. A continuación, debemos modificar las marcas necesarias para los elementos del modelo (clases, atributos, operaciones, etc.) desde el panel de marcas de WSL8 o de cualquiera de sus *cartridges* "padres" (EJB1.1/2.0, Java2).

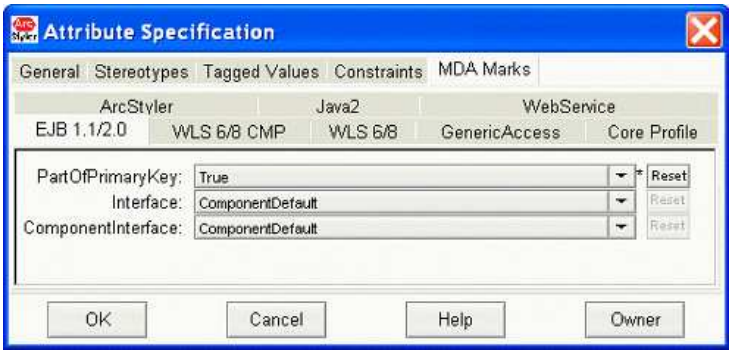


Figura 54. Panel de marcas para un atributo

Ejemplo:

La Figura 54 muestra el panel de marcas de EJB1.1/2.0 para el caso de un atributo. Desde aquí podemos indicarle al MDA-Cartridge que el atributo forma parte de la clave primaria, estableciendo la marca PartOfPrimaryKey a true.

Para que ArcStyler genere los ficheros de despliegue adecuados para nuestro modelo EJB, debemos crear un nuevo componente (*Component*) y asignarle como elementos residentes (*Resident Elements*) los elementos de nuestro modelo. La Figura 55 muestra el componente EJB creado para nuestra aplicación y su correspondiente especificación.

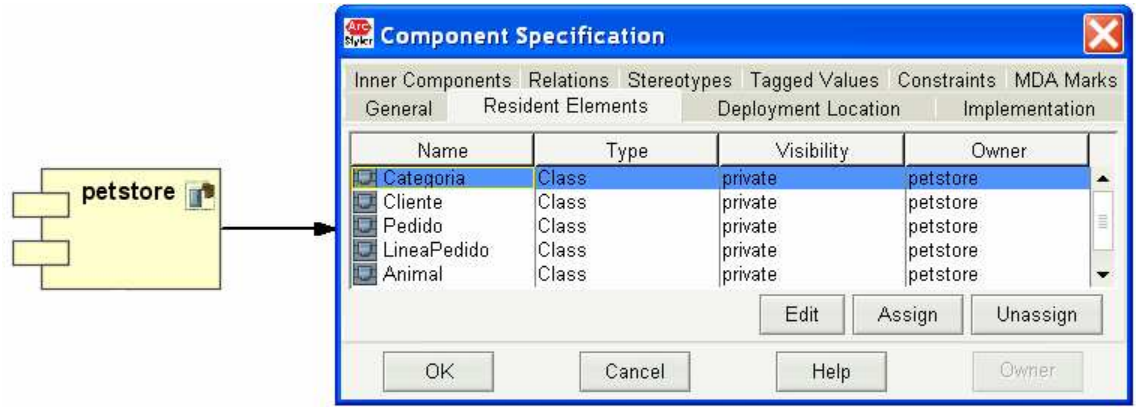


Figura 55. Componente y especificación asociada

Una vez completado el modelo podemos generar el código correspondiente a la capa EJB, y probar los componentes generados mediante la herramienta ANT incorporada en ArcStyler. Como ya dijimos, Apache Ant es una herramienta basada en Java para la construcción de programas, parecida al Make de Unix, pero usando ficheros en formato XML.

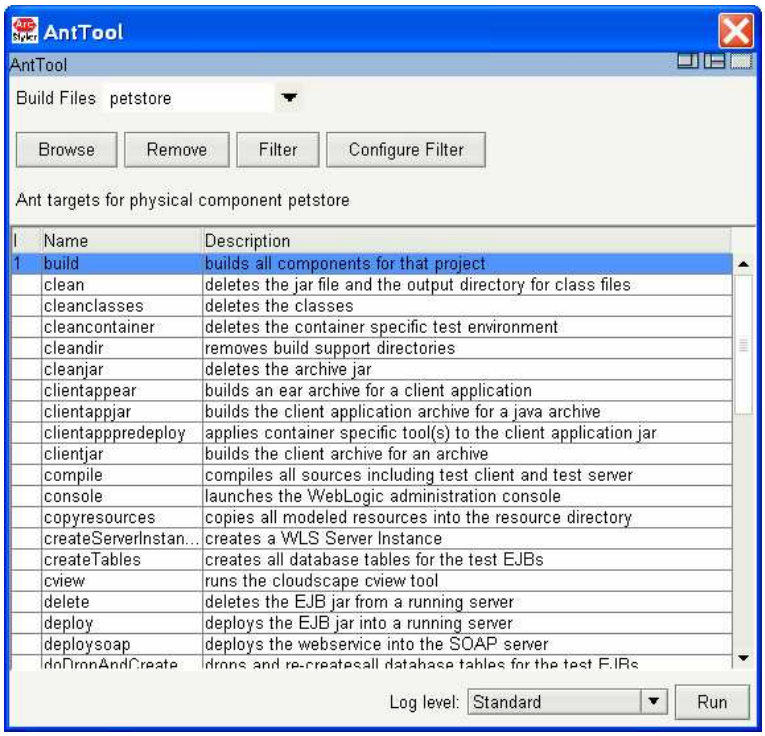


Figura 56. La herramienta ANT incorporada en ArcStyler

Desde la herramienta ANT y gracias a los ficheros de script generados por ArcStyler para esta herramienta podemos, entre otras cosas, compilar todo el código generado, iniciar el servidor de bases de datos, crear las tablas de bases de datos, iniciar el servidor Bea WebLogic o ejecutar una clase Java de prueba. En la Figura 56 podemos ver el aspecto de esta herramienta.

Modelo de Base de Datos

Al contrario que en OptimalJ, en ArcStyler no existe un modelo explícito de base de datos, sino que este está implícito en el modelo EJB. El *cartridge* usado para el modelo EJB (WLS8) crea una tabla en la base de datos por cada componente EJB, y una columna por cada atributo y por cada extremo de asociación.

¿Cómo podemos controlar entonces los aspectos de la base de datos? Mediante las **marcas** del modelo EJB. Por ejemplo, para el nombre de la tabla se utiliza por defecto el nombre del componente EJB. Sin embargo, se puede cambiar el nombre de la tabla mediante la marca *TableName* del panel de marcas de WSL8, como puede verse en la Figura 57.

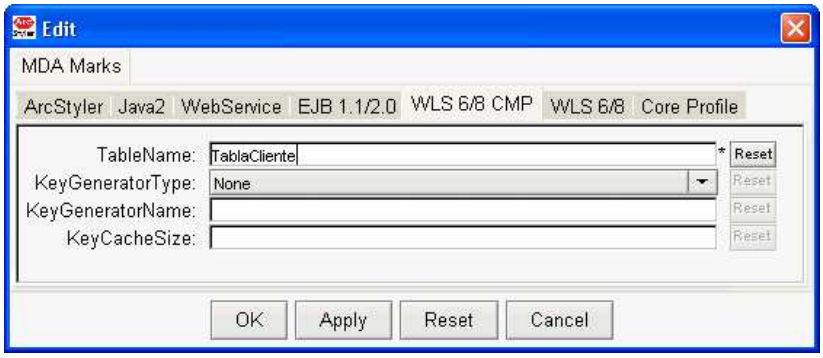


Figura 57. Marcas para manejar la persistencia de las clases del PIM

Para las columnas de la base de datos podemos, por ejemplo, modificar el nombre (*ColumnName*) y el tipo (*ColumnType*), permitir el valor *null* (*AllowsNull*) o imponer valores únicos (*isUnique*), como vemos en la Figura 58.

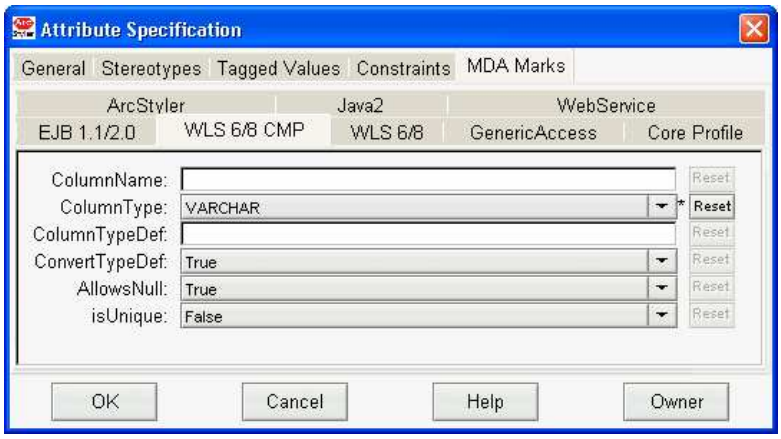


Figura 58. Marcas para manejar la persistencia de atributos

Modelo Web

La capa Web para nuestra aplicación *Pet Store* se ha realizado con el *cartridge WebAccessor*. Este *MDA-Cartridge* usa el framework *Accessor* (Véase la sección Framework *Accessor* en pág. 64) para definir interfaces Web, generando ficheros JSP y servlets que pueden ejecutarse en un servidor de JSPs como *Tomcat*.

Para la capa Web hemos definido dos *Accessor* y trece *Representers*. No detallaremos la construcción de este modelo, ya que, aunque no es excesivamente complicado, resulta demasiado extenso para incluirlo en este documento y no tiene demasiado interés para el proyecto. En la Figura 59 puede verse el complejo diagrama de clases con todos los elementos involucrados en el modelo *Accessor*.

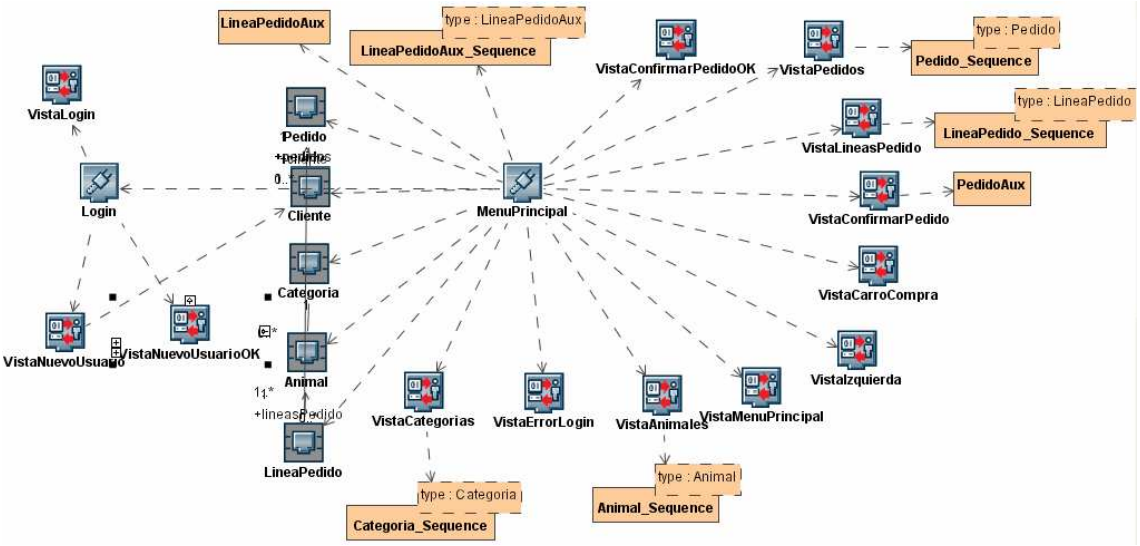


Figura 59. Diagrama de clases para el modelo *Accessor*

Es interesante comentar cómo los *Accessor* pueden corresponderse con los **casos de uso** de una aplicación. Un caso de uso es una descripción de la secuencia de interacciones que se producen entre un usuario y el sistema, cuando el usuario usa el sistema para llevar a cabo una tarea específica. Esta secuencia de interacciones es lo que se define en el diagrama de estados asociado a un *Accessor*. Esta analogía resulta muy interesante, ya que se podría usar el framework *Accessor* para modelar la capa de presentación en un desarrollo dirigido por casos de uso.

Para nuestro *Pet Store*, podemos identificar dos casos de uso principales: uno para el acceso al sistema (*Login*) y otro para la realización de pedidos (*RealizarPedido*). De ahí que hayamos definido dos *Accessor* para nuestra aplicación:

- *Login*: controla el acceso al sistema y el registro de nuevos usuarios.
- *RealizarPedido*: controla el resto de la aplicación, consistente en consultar animales y realizar pedidos.

Cada uno de estos *Accessor* tiene asociado un diagrama de estados asociados para definir el flujo de control. La Figura 60 muestra el diagrama de estados asociado al *Accessor RealizarPedido*. Este diagrama define la mayor parte del flujo de control de la

aplicación[13], que se detalla en el apartado Interfaz de la aplicación generada (pág. 71).

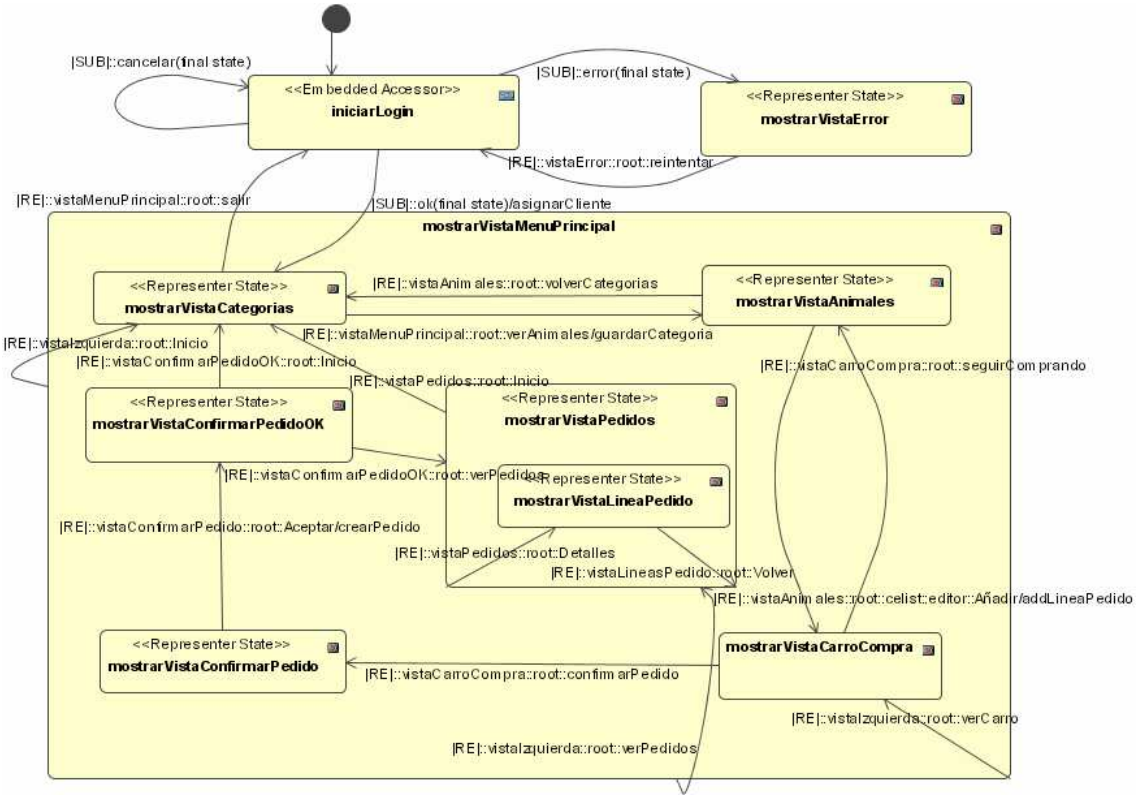


Figura 60. Diagrama de estados del *Accessor RealizarPedido*

7.3 Interfaz de la aplicación generada

En este apartado describiremos la interfaz generada para la aplicación a partir del modelo *Accessor* del *Pet Store*.

Acceso del usuario al sistema

Al iniciar la aplicación, se muestra la pantalla de entrada al sistema de la Figura 61. Para realizar las pruebas, hemos creado una cuenta de usuario invitado con nombre de usuario "*usuario"* y contraseña "*usuario"*.

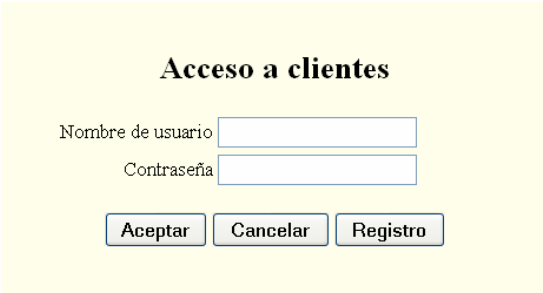


Figura 61. Entrada al sistema

[13] Todo el flujo de control excepto el acceso al sistema y el registro de nuevos usuarios, controlado por el *Accessor Login*.

Desde esta pantalla también podemos registrar nuevos usuarios presionando en el botón *Registro*. El formulario de registro de nuevo usuario puede verse en la Figura 62. Gracias al generador de claves primarias que hemos implementado para la aplicación, el campo *Identificador del Cliente* se completa automáticamente.

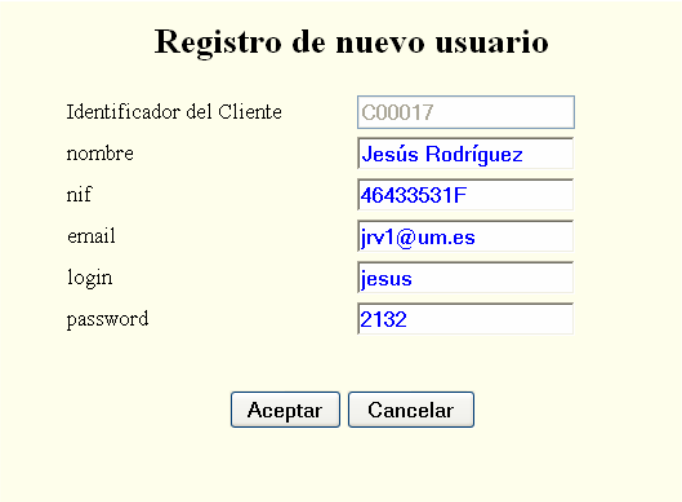


Figura 62. Registro de nuevo usuario

Con el botón *Aceptar* crearemos el nuevo usuario en el sistema y regresaremos a la pantalla de acceso al sistema.

Página Principal de la Aplicación

Una vez introducidos el nombre de usuario y la contraseña, aparecerá la página principal de la aplicación mostrada en la Figura 63. En esta página se muestran las distintas categorías de animales disponibles junto con su descripción. Seleccionando una de las categorías y presionando el botón *Ver Animales* accederemos a los animales de la categoría seleccionada.

Si pulsamos el botón *salir* finalizaremos la sesión actual y regresaremos a la pantalla de acceso al sistema.

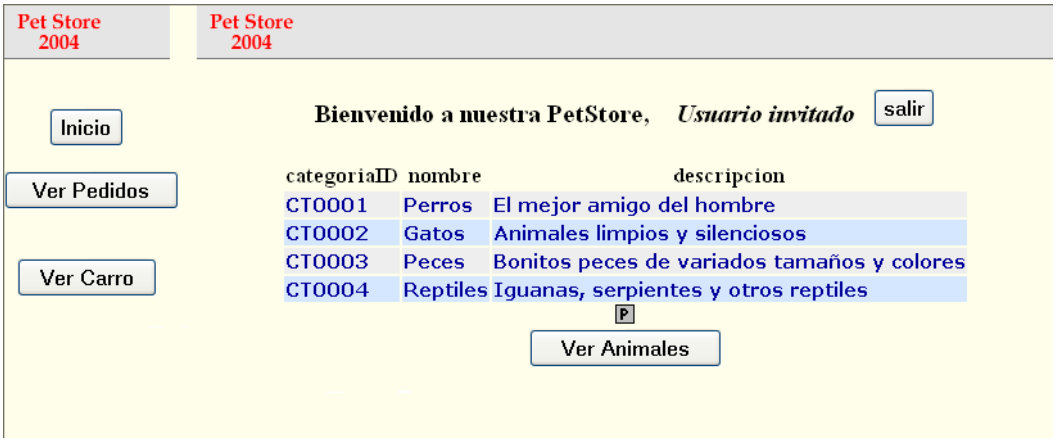


Figura 63. Página principal de la aplicación

Por otro lado, en la parte izquierda de la pantalla hay un *frame* accesible desde todas las páginas de la aplicación, desde el que podemos realizar las siguientes acciones:

- Regresar a la página principal de la aplicación (botón *Inicio*).
- Ver los pedidos realizados por el usuario (botón *Ver Pedidos*).
- Ver el carro de la compra actual (botón *Ver Carro*).

Consulta de Animales

Una vez seleccionada una categoría, aparecerán los animales disponibles para dicha categoría, como muestra la Figura 64 para el caso de los perros. Para cada animal se muestra el identificador, el nombre, la descripción, el precio y su foto.

Con el botón *Añadir* que aparece junto con la información de cada animal podemos añadir un animal al carro de la compra, mientras que con el botón *Volver a Categorías* regresaríamos a la página principal de la aplicación.

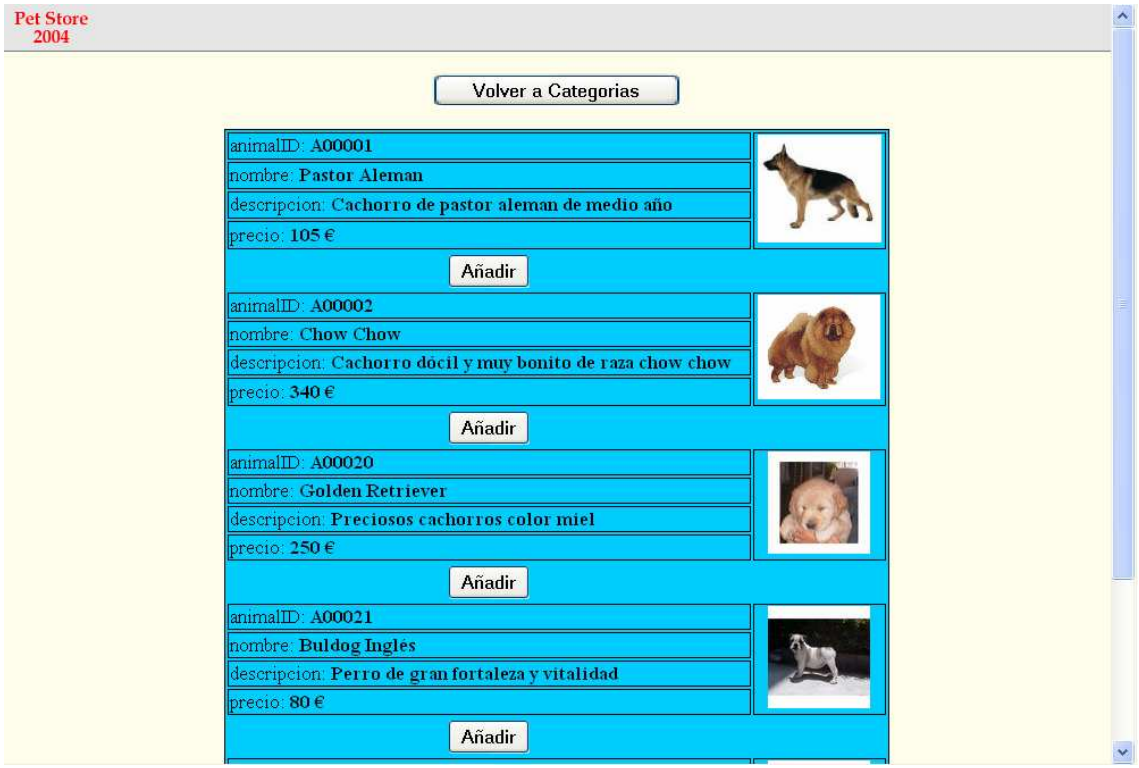


Figura 64. Listado de animales de la categoría *Perros*

Carro de la compra

Cada vez que se añada un animal al carro de la compra actual, se mostrará el contenido del carro, de manera similar al de la Figura 65. Desde aquí podremos añadir (botón '+') o restar (botón '-') unidades al carrito, eliminar productos (botón *Eliminar*) y conocer el importe total del pedido (campo *Total Carro*).

Con el botón *Seguir Comprando* regresaremos a la página previa a la actual para continuar la compra.

Para confirmar el pedido, debemos presionar el botón *Confirmar Pedido*.

Figura 65. Carro de la compra

Confirmación de Pedido

Al pulsar el botón *Confirmar Pedido* desde el carro de la compra, aparecerá el formulario de confirmación de pedido de la Figura 66, donde rellenaremos los datos relativos a la dirección de entrega y a la forma de pago.

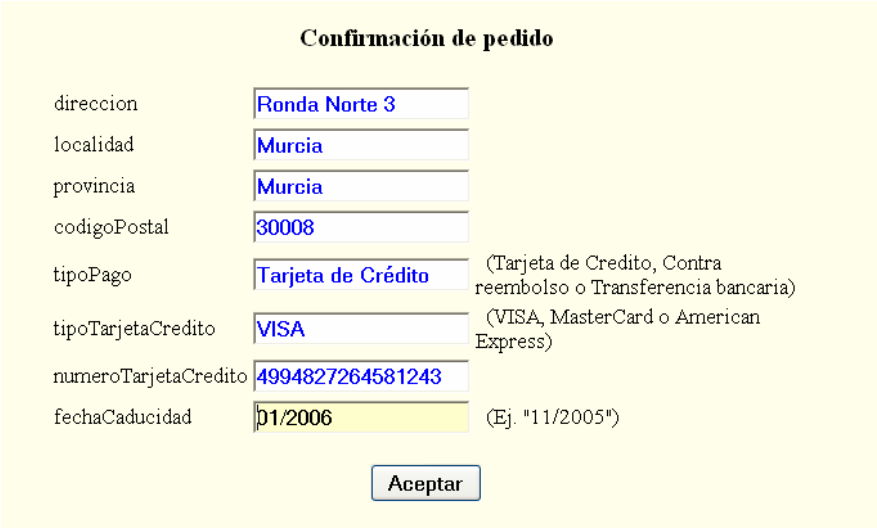


Figura 66. Confirmación de pedido

Presionando el botón *Aceptar* aparecerá el mensaje de la Figura 67, indicando que el pedido se ha registrado correctamente. Con el botón *Inicio* de esta figura regresaremos a la página principal, mientras que con el botón *Ver Pedidos* iremos a la página de consulta de pedidos.

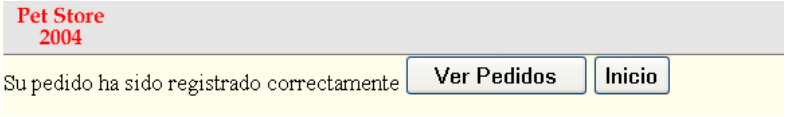


Figura 67. Registro correcto de un pedido

Consulta de Pedidos

Mediante el botón *Ver Pedidos* de la Figura 67 o el del *frame* de la parte izquierda de la Figura 63 podemos consultar los pedidos realizados por el usuario. La Figura 68 muestra la página de consulta de pedidos.

Mediante el botón *Anular* podemos anular un pedido (el estado del pedido pasaría a ANULADO).

Figura 68. Consulta de los pedidos realizados

También podemos ver las líneas de las que consta el pedido seleccionando un pedido y presionando el botón *Detalles*. La información mostrada para cada pedido puede verse en la Figura 69.

Figura 69. Detalles del pedido

7.4 Evaluación de ArcStyler

A continuación evaluaremos la herramienta *ArcStyler* según los criterios presentados en el apartado Criterios para Evaluar Herramientas MDA.

Id	Propiedad	Puntuación	Comentarios
P01	Soporte para PIMs	4	ArcStyler permite definir completos PIMs mediante diagramas de clases UML.
P02	Soporte para PSMs	0	ArcStyler NO tiene soporte para construir PSMs explícitos. En lugar de PSMs, se usa un PIM estereotipado junto con

			las *marcas* para generar directamente el código.
P03	Permite varias implementaciones	4	La herramienta incorpora de partida numerosas implementaciones (Java2, EJB, Servicios Web, CORBA, .NET, etc.). Además, gracias a CARAT (*CARtridge ArchiTecture*), es posible definir nuevos *MDA-Cartridges*, ampliando el abanico de plataformas disponibles.
P04	Integración de modelos	2	La herramienta permite que elementos de un modelo interactúen con los elementos de otro, por ejemplo los elementos de los WebAccessors pueden acceder a los EJB, generando puentes que facilitan la comunicación. Sin embargo, se necesita bastante código "pegamento" para que la integración sea completa.
P05	Interoperabilidad	4	ArcStyler también puede exportar e importar modelos vía XMI.
P06	Acceso a la definición de las transformaciones	4	La aplicación permite crear nuevos MDA-Cartridges o extender los existentes. En estos MDA-Cartridges están definidas todas las transformaciones para una implementación determinada. Existe una completa arquitectura, llamada CARAT, para definir nuevos *cartridges* e incorporarlos a la herramienta.
P07	Verificador de modelos	2	Cada MDA-Cartridge incorpora su propio verificador de modelos. Los verificadores que hemos usado, para las plataformas EJB y WebAccessor, son poco eficaces. Muchos errores pasan desapercibidos y generalmente se trasladan a la fase generación de código, donde es difícil encontrar el elemento concreto del modelo que produce el fallo.
P08	Expresividad de los modelos	4	Los modelos usados (EJB y WebAccessors) son muy expresivos y, junto con las marcas, permiten controlar numerosos aspectos de la plataforma destino.
P09	Uso de patrones	3	Cada *cartridge* puede implementar los patrones que desee. De los *cartridges* usados para el estudio (EJB y WebAccessors) destacamos los patrones *Front-Controller* para el control de la interfaz de usuario, y el patrón *Compact Bean* para agrupar en una clase todo lo referente a un EJB.
P10	Soporte para la regeneración de modelos	3	ArcStyler maneja bastante bien la regeneración de código. El desarrollador puede escribir código en *áreas protegidas*, que se conservará en sucesivas regeneraciones de código. No obstante, en ocasiones la modificación de *áreas protegidos* impide la regeneración completa de algunos tipos de ficheros (por ejemplo, los JSPs).
P11	Transformaciones intra-modelo	2	La herramienta permite generar Accessors genéricos a partir del modelo EJB, gracias al framework *Generic Accessor* [16]. No obstante, no hemos profundizado en esta funcionalidad.
P12	Trazabilidad	1	ArcStyler dispone de una opción para mostrar todos los diagramas que contienen determinado elemento del modelo, por ejemplo, una clase o relación. Sin embargo, no es posible saber el origen de los ficheros de código o el destino en el código de un elemento del modelo.
P13	Ciclo de vida	4	La herramienta engloba varias fases del ciclo de vida: análisis, diseño, prueba, despliegue y mantenimiento. Para la codificación debemos usar herramientas IDE externas (como JBuilder), que pueden integrarse en el entorno de ArcStyler. También incluye el modelado del negocio y de requisitos, aunque no hemos probado este tipo de modelos.
P14	Estandarización	4	ArcStyler usa los principales estándares involucrados en

			MDA: UML como lenguaje de modelado, XMI para intercambiar modelos y repositorios MOF para almacenar los modelos. Otro estándar usado en ArcStyler es JMI, para definir la interfaz de acceso a los modelos usando Java.
P15	Control y refinamiento de las transformaciones	4	Las marcas de cada *cartridge* permiten controlar numerosos detalles de la transformación, proporcionando al desarrollador una gran flexibilidad.
P16	Calidad del código generado	4	El código generado por la herramienta está bien documentado y es bastante legible. Aunque en principio es difícil familiarizarse con la estructura de los ficheros de código, la documentación de la herramienta al respecto es bastante completa y al poco tiempo resulta sencillo insertar nuestro código extra.
P17	Herramientas de soporte	3	Al margen del editor de modelos, ArcStyler incorpora, entre otras herramientas, un servidor de servlets/JSPs (*Tomcat*) y la herramienta *ANT* para el despliegue y las pruebas. Dispone de una herramienta IDE muy básica, así que para suplir esta carencia ArcStyler genera ficheros de soporte para la herramienta JBuilder. El servidor de EJB usado en la evaluación (*Bea Weblogic Server*) y el servidor de base de datos (*PointBase*) hemos tenido que instalarlos de manera independiente. Una característica muy interesante de ArcStyler es que permite incorporar otras herramientas al entorno de desarrollo, aunque esta funcionalidad no ha sido probada.

8 OptimaJ frente a ArcStyler. Estudio comparativo

Tras presentar y evaluar cada herramienta de forma individual, discutiremos en esta sección de forma más extendida las conclusiones extraídas de la evaluación de *OptimalJ* y *ArcStyler*, siguiendo los criterios presentados en el apartado Criterios para Evaluar Herramientas MDA (pág. 32) y haciendo especial hincapié en aquellos aspectos en los que más difieren ambas herramientas.

Por último, comentaremos otros aspectos no relacionados con MDA pero que consideramos de gran importancia, como el **rendimiento**, la **facilidad de uso** o la **documentación** de cada una de las herramientas.

8.1 Aspectos MDA

Soporte para PIMs (P01)

Tanto OptimalJ como ArcStyler permiten crear modelos independientes de plataforma bastante completos.

En el caso de OptimalJ, el PIM del sistema está representado por el *Modelo de Clases* (*Domain Class Model*). Este modelo tan sólo incluye un detalle de plataforma que en nuestra opinión no debería aparecer en el PIM: la clave primaria de una clase y su restricción de unicidad asociada (*primary*), que son aspectos propios del modelo de base de datos. ArcStyler no muestra este detalle en el modelo, sino que establece la clave primaria de una clase mediante una **marca** en el atributo correspondiente.

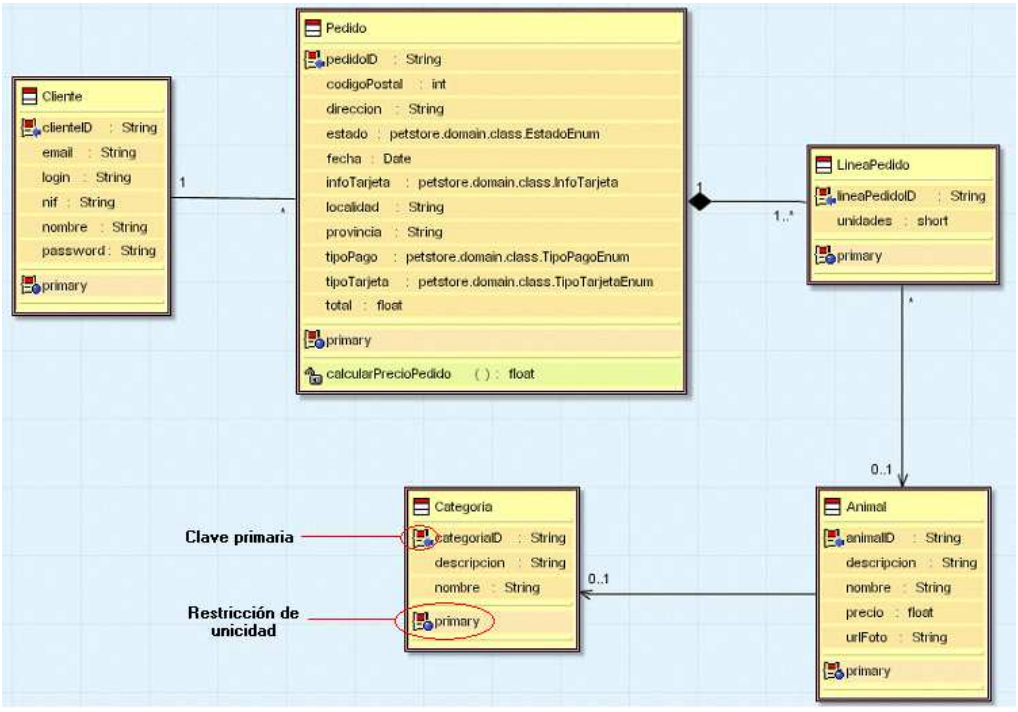


Figura 70. Modelo de Clases (OptimalJ)

En la Figura 70 mostramos el modelo de clases del *Pet Store* donde aparecen señalados estos detalles para la clase *Categoria*.

ArcStyler también permite construir completos PIMs para mostrar la estructura de los datos de la aplicación, con la novedad de que podemos establecer *estereotipos* sobre los elementos del PIM. Estos estereotipos se usan junto con las marcas para dirigir la transformación a la plataforma destino. La diferencia está en que los estereotipos se utilizan en la *transformación de tipos*, mientras que las marcas se usan en la *transformación de instancias*. Estos conceptos se detallan en el apartado Definiciones de Transformación (pág. 23). La Figura 71 muestra la clase *Pedido* del PIM de nuestro *Pet Store* en ArcStyler. Vemos que clase tiene el estereotipo `<<ComponentSegment>>`, que hará que se transforme en un EJB cuando se aplique la transformación de tipos.

Pero al igual que en OptimalJ, el PIM de ArcStyler no está totalmente libre de detalles sobre la plataforma destino. Por ejemplo, los métodos de búsqueda, métodos home o métodos factoría, conceptos de la plataforma EJB, deben definirse en el PIM. Conceptualmente, estos métodos deberían definirse a nivel de PSM, pues representan operaciones que sólo tienen sentido para la plataforma EJB. No obstante, al no disponer en ArcStyler de PSMs explícitos, la única alternativa es definir estas operaciones en el PIM, ensuciándolo con aspectos dependientes de la plataforma.

En la Figura 71 se muestra el método *findByFecha()*. El estereotipo `<<find>>` indica que se trata de un método de búsqueda. Este método de búsqueda, como hemos dicho, representa un concepto de la plataforma EJB que no debería aparecer en el PIM.

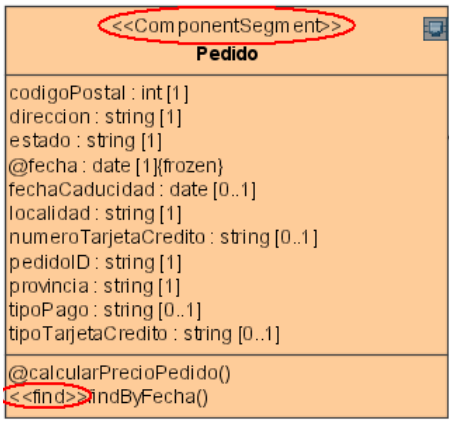


Figura 71. Clase *Pedido* del PIM (ArcStyler)

Otro aspecto negativo de ArcStyler es que no hay un PIM único que dirija todo el desarrollo de la aplicación. Podríamos decir que el modelo de clases que define los EJB es el PIM de la aplicación. No obstante, aunque el modelo de *Accessors* accede a las entidades del modelo de clases, se trata también de un modelo independiente de la plataforma, definido independientemente del modelo de clases. Como resultado tenemos dos PIMs independientes que interactúan entre sí. Esto contrasta con el PIM único de OptimalJ, centro de todo el desarrollo y mediante el cual se generan el resto de modelos.

Soporte para PSMs (P02)

El proceso típico de desarrollo con MDA contempla como pasos principales la transformación de PIM a PSM y de PSM a código, como se muestra en el esquema de la Figura 72.

Figura 72. Pasos en el desarrollo con MDA

OptimalJ sigue fielmente este esquema, generando a partir del PIM tres tipos de PSM que interactúan entre sí:

- PSM para la plataforma EJB.
- PSM para la tecnología Web JSP/servlets.
- PSM para la base de datos.

Aunque estos modelos no son muy expresivos y permiten una flexibilidad limitada, nos proporcionan un modelo intermedio entre PIM y el código donde podemos definir elementos específicos de una plataforma que no tienen sentido en el PIM, como métodos de búsqueda o métodos *home* para la plataforma EJB, roles de seguridad para la capa Web o vistas sobre las tablas del modelo de base de datos. Además, el PIM es el centro de todo el desarrollo, ya que de él derivan todos los PSMs.

ArcStyler, por contra, **carece de PSMs explícitos**. En lugar de PSMs, ArcStyler usa un PIM estereotipado y marcas para generar directamente el código de la aplicación para una plataforma determinada. Como vimos en el apartado anterior, esto provoca que algunos aspectos de la plataforma destino (p.e. métodos de búsqueda, métodos factoría o métodos *home* de la plataforma EJB) deban modelarse en el PIM.

Permitir varias implementaciones (P03)

Una de las claves de MDA consiste en permitir varias plataformas de implementación a partir de un mismo PIM. En este aspecto, existe una gran diferencia entre OptimalJ y ArcStyler.

OptimalJ permite generar tres tipos distintos de PSMs (modelo EJB, modelo de base de datos y modelo Web), relacionados entre sí y dirigidos a generar una aplicación para la **plataforma J2EE**. Ésta es la única plataforma de implementación disponible en OptimalJ. La única variación posible consiste en generar Servicios Web a partir del modelo EJB, e integrar la aplicación con aplicaciones externas usando CORBA o Servicios Web. Suponemos que este problema se irá solucionando en futuras versiones de la herramienta, añadiendo soporte para otras plataformas. En cualquier caso, se trata de una herramienta cerrada y el soporte para nuevas plataformas lo debe proporcionar el fabricante de la herramienta, la empresa *Compuware*.

En el polo opuesto tendríamos ArcStyler. De partida incorpora numerosos ***MDA-Cartridges***, cada uno dando soporte a una determinada plataforma (EJB, Java2, Servicios Web, .NET, JSP/Servlets). Además, mediante **CARAT** (CARtridge ArchiTecture), es posible crear nuevos *MDA-Cartridges*, extendiendo los existentes para adaptarlos a nuestros requisitos específicos o creando nuevos *cartridges* desde cero. Estos nuevos *MDA-Cartridges* se pueden instalar fácilmente como *plugins* en el entorno de ArcStyler para usarlos en nuestros proyectos.

Esta estructura abierta de ArcStyler hace que no dependamos exclusivamente del soporte suministrado por el fabricante (*Interactive Objects*), permitiendo a una empresa adaptarse a nuevas tecnologías con el paso de los años mediante la creación de nuevos *MDA-Cartridges*. Este aspecto supone en nuestra opinión un factor clave, que hace que ArcStyler esté en un nivel superior a OptimalJ como herramienta MDA.

Integración de modelos (P04)

Como ya hemos dicho, OptimalJ está orientado casi exclusivamente a crear aplicaciones para la plataforma J2EE usando los tres tipos de modelos ya vistos: modelo de base de datos, modelo EJB y modelo Web. Una ventaja de tener este esquema prefijado es la completa integración entre las distintas capas. La herramienta genera automáticamente los puentes adecuados para comunicar la capa Web con la capa EJB, y la capa EJB con la base de datos. Desde el primer momento obtenemos una aplicación totalmente operativa en la que los JSPs de la interfaz Web interactúan con los EJB del modelo del negocio y estos EJB interactúan con las tablas de la base de datos, de forma totalmente transparente al desarrollador.

Sin embargo, en ArcStyler esta integración es más complicada. Al poder elegir entre tantas implementaciones posibles, un *MDA-Cartridge* no siempre será capaz de generar los puentes adecuados para comunicarse con los elementos generados por otro *MDA-Cartridge*, puesto que puede que "no conozca" cómo interactuar con dichos elementos. Será responsabilidad de cada *cartridge* generar los puentes adecuados para comunicarse con elementos de otros *cartridges*.

Un ejemplo de esto lo tenemos en la comunicación entre el modelo *WebAccessor* y el modelo *EJB* para nuestro *Pet Store*. Mientras que en OptimalJ no es necesario añadir ningún tipo de código para comunicar la capa Web con la capa EJB, en ArcStyler necesitamos introducir código "pegamento" para que los datos de una instancia de EJB se muestren en un determinado *Representer* (interfaz de usuario). En este caso concreto, el *MDA-Cartridge* para los *WebAccessor* facilita en gran medida esta comunicación del siguiente modo:

- Generando automáticamente métodos auxiliares dentro del código fuente de un *Accessor* para acceder a la interfaz *Home* de los EJB usados por el *Accessor*, de manera que gracias a esta interfaz *Home* podamos recuperar, realizar búsquedas, modificar o crear nuevas instancias de EJBs.

- Definiendo un modo de acceso sencillo y uniforme para todos los elementos de la interfaz. Por ejemplo, si queremos modificar desde el código fuente de un

Accessor el valor del campo *dni* del *representer VistaNuevoUsuario*, debemos escribir una sentencia similar a esta:

```
getVistaNuevoUsuarioRepresenter().getRoot().SetDni("nuevoValor");
```

El código "pegamento" consistiría entonces en recuperar o modificar instancias de EJBs a través de la interfaz *Home*, recuperar datos introducidos en la interfaz, "rellenar" campos de la interfaz con datos de los EJB, etc. Este código no sería necesario en OptimalJ, ya que la desde el primer momento la interfaz de la aplicación está comunicada con los objetos de negocio.

Interoperabilidad (P05)

Respecto a este criterio hay poco que comentar. Ambas herramientas permiten importar y exportar modelos vía ficheros XMI, facilitando así la interoperabilidad con otras herramientas de modelado.

Acceso a la definición de las transformaciones (P06)

Como ya comentamos en la evaluación de OptimalJ, la herramienta no permite acceder a las definiciones de las transformaciones, que en OptimalJ se incluyen en los llamados *patrones de transformación*. No obstante, podemos usar la edición *OptimalJ Arquitecture Edition* para crear nuevos patrones de transformación que permitan pasar de PIM a PSM (patrones de tecnología) o de PSM a código (patrones de implementación). Por ejemplo, podríamos diseñar un flujo de navegación para la capa Web distinto al proporcionado por el patrón *HTML Default*, para aplicarlo siempre que queramos a nuestros proyectos. Desconocemos el grado de dificultad que entraña la creación de un nuevo patrón de transformación, aunque por la documentación al respecto parece una tarea compleja que sólo será "rentable" si usamos el patrón creado en múltiples proyectos.

ArcStyler proporciona un mejor soporte para acceder a las definiciones de las transformaciones gracias a CARAT. Desde la misma herramienta podemos extender un *cartridge* existente y modificarlo para satisfacer nuestros requisitos específicos. Además, CARAT usa modelos para especificar reglas de transformación, siguiendo la filosofía MDA también para las definiciones de transformaciones. Crear un nuevo *cartridge* parece una tarea complicada, pero una vez construido obtendríamos soporte para una nueva plataforma y podríamos aprovecharlo para múltiples proyectos. Habría sido interesante profundizar en esta arquitectura de definición de *cartridges*, pero queda fuera del ámbito de este proyecto.

Verificador de modelos (P07)

Para aplicar una transformación a un determinado modelo, ya sea para generar otro modelo (PSM) o para producir código fuente, es necesario que el modelo origen cumpla una serie de reglas determinadas por la transformación que va a aplicarse. Estas reglas

aseguran que pueda ser aplicada la transformación sin producir efectos no deseados en la generación de PSMs o de código.

OptimalJ verifica la corrección de un modelo antes de aplicar cualquier transformación. Dispone para ello de un verificador para PIMs y otro para cada tipo de PSM, cuyas comprobaciones están bien documentadas en el manual de la herramienta [4]. Por ejemplo, el verificador para el modelo EJB realiza, entre otras, las siguientes comprobaciones:

- Un componente EJB sesión no puede tener un método de negocio *home*.
- La propiedad *domainElement* de un parámetro de búsqueda debe tener un valor.
- Un método de creación de EJB no puede tener tipo de retorno.
- Un componente EJB de tipo sesión no puede tener más de un método `ejbRemove()`.
- El método `ejbRemove()` de un componente EJB de tipo sesión debe tener *void* como tipo de retorno y no debe tener ningún parámetro.

ArcStyler también permite usar verificadores de modelos, pero no están incorporados en la herramienta, sino que cada *MDA-Cartridge* debe encargarse de implementar el verificador de sus modelos de entrada.

Cabe destacar que para los *MDA-Cartridges* usados en nuestra evaluación (*WebAccessors* y *WLS8*) , no hay documentación disponible sobre las comprobaciones que llevan a cabo sus respectivos verificadores. Además, muchos errores pasan desapercibidos para el verificador y se trasladan a la fase de generación de código, donde resulta complicado detectar el elemento del modelo que produjo el fallo. Por ejemplo, no definir el tipo de dato para un atributo no genera error en el verificador del *cartridge* WebAccessor, pero produce una excepción al generar el código para el modelo. Incluso en el caso de que el verificador detecte el error, en la mayoría de ocasiones proporciona escasa información acerca del motivo del error y del elemento que lo produjo.

Expresividad de los modelos (P08)

Es muy importante que los modelos del sistema sean lo más expresivos posibles. De esta manera, el desarrollador tendrá mayor flexibilidad y precisión a la hora de describir la arquitectura de las distintas partes del sistema.

La importancia de esta propiedad queda patente al comparar el modelo Web de ambas herramientas. OptimalJ define un lenguaje para modelar los PSMs de la capa Web poco expresivo, de muy alta granularidad, el cual da muy pocas opciones al desarrollador. En la Figura 73 puede verse un ejemplo de modelo Web en OptimalJ. Básicamente este modelo muestra componentes Web relacionados con componentes EJB. No permite especificar mediante modelos el flujo de control de la navegación Web, imprescindible para construir aplicaciones “a medida”.

¿Qué flujo de navegación genera entonces OptimalJ? La respuesta está en los **patrones de implementación**. La herramienta proporciona una navegación por defecto para cada

componente Web[14], proporcionada por el patrón *HTML Default*. Básicamente genera una interfaz que permite crear, recuperar, modificar y eliminar instancias de cada componente Web. Modificar la navegación generada por defecto es bastante complejo y requiere modificar el código fuente generado.

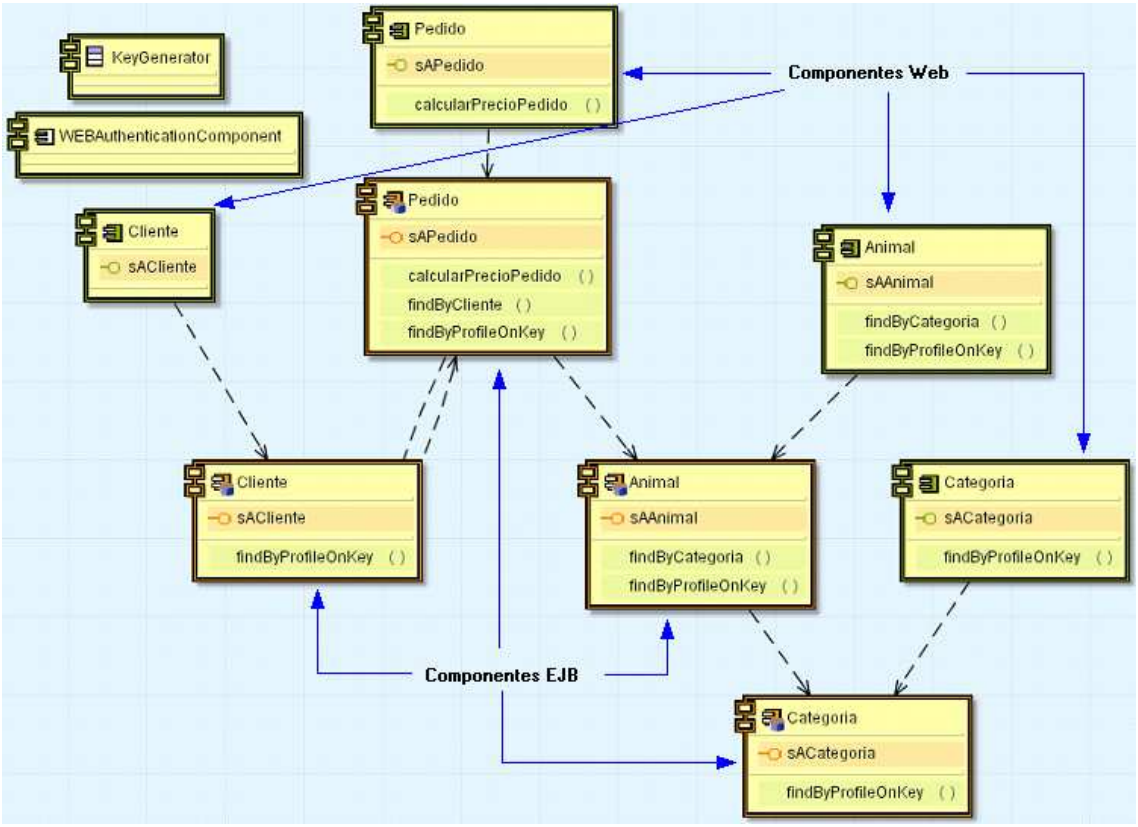


Figura 73. Modelo Web (OptimalJ)

Bien es cierto que mediante *OptimalJ Architecture Edition* podemos definir nuestros propios patrones. Aunque esto no se ha probado en el proyecto, por lo que hemos visto no parece ser algo sencillo, pero sería útil si vamos a usar el mismo esquema de navegación en todos nuestros proyectos.

Como conclusión, cambiar el flujo de navegación Web, aunque éste puede *refinarse* ligeramente, requiere modificar el código fuente (ficheros JSP, Web actions, etc). Al no usarse modelos para definir un aspecto tan importante como la navegación de la aplicación, se rompe ligeramente con la filosofía esencial del MDA: desarrollo dirigido íntegramente por modelos. Según los diseñadores de la herramienta, en futuras versiones se permitirá un control más flexible de la capa Web, pero por ahora debemos conformarnos con lo que la herramienta nos proporciona. Este aspecto supone, en nuestra opinión, una de las mayores deficiencias de OptimalJ.

En contraste tenemos el **modelo *Accessor*** de ArcStyler. Este modelo es muy expresivo y permite especificar de manera precisa el flujo de control de la interfaz de la aplicación mediante diagramas de estado, como puede verse en la Figura 74.

[14] OptimalJ crea un componente Web por cada clase del PIM que no sea una clase agregada.

El framework *Accessor* permite, por lo tanto, diseñar interfaces de usuario a medida, a costa, eso sí, de un mayor esfuerzo por parte del desarrollador.

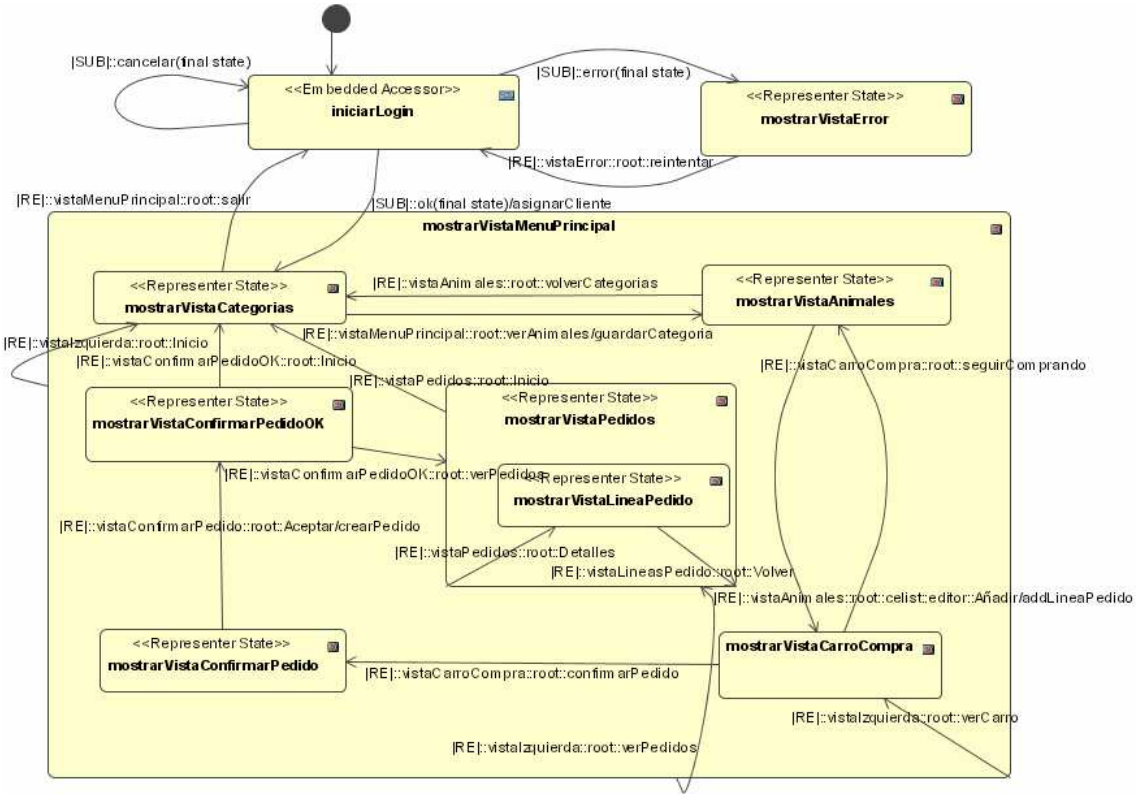


Figura 74. Diagrama de Estados para un *Accessor* (ArcStyler)

Como ya comentamos en el apartado Modelo Web (pag. 70) puede establecerse una analogía entre los *Accessor* y los **casos de uso** de una aplicación. Esta secuencia de interacciones descrita en un caso de uso puede perfectamente expresarse mediante el diagrama de estados asociado a un *Accessor*. Esta analogía resulta de gran utilidad, ya que podríamos usar el framework *Accessor* para modelar la capa de presentación en un desarrollo dirigido por casos de uso, diseñando un diagrama de estado para cada caso de uso del sistema.

Uso de patrones (P09)

Un desarrollo completo con MDA debería permitir al desarrollador aplicar patrones de diseño en la construcción de los distintos modelos o incluso en la generación de código, de manera que puedan aplicarse repetidamente y sin esfuerzo las mejores soluciones encontradas para determinados problemas.

OptimalJ utiliza patrones en todos los niveles de abstracción, clasificándolos en varios tipos según su propósito. A nivel de PIM, permite reutilizar modelos del dominio mediante **patrones de dominio** definidos por el usuario, y lo mismo sucede a nivel de PSM con los **patrones de aplicación**.

Mediante los **patrones de tecnología** se transforma el modelo del dominio (PIM) en el modelo de la aplicación (PSMs), y con los **patrones de implementación**

transformamos los PSMs en código. Estos patrones contienen las definiciones de transformación para dar soporte a la plataforma J2EE.

A nivel de código, pueden aplicarse **patrones de código**, patrones de bajo nivel que se aplican al código generado, como los patrones GoF [10]. De los patrones de código utilizados por OptimalJ destacamos el patrón *Front Controller* para procesar las peticiones del usuario en la capa de presentación, y el patrón *Business Façade* para acceder a la capa de negocio (EJB) desde la capa de presentación (Web).

Todos estos patrones se crean con la versión *OptimalJ Architecture Edition*, excepto los patrones de dominio, que como vimos en el apartado La ventaja principal de este modelo es que permite generar automáticamente *Beans* de tipo *Session* en el modelo de EJB, aunque la realidad es que no ofrece mucha flexibilidad a la hora de definir las vistas.
Patrones de Dominio (pág. 38) pueden crearse fácilmente desde la misma herramienta. La Figura 75 muestra los tipos de patrones existentes en OptimalJ y los niveles en los que se aplican.

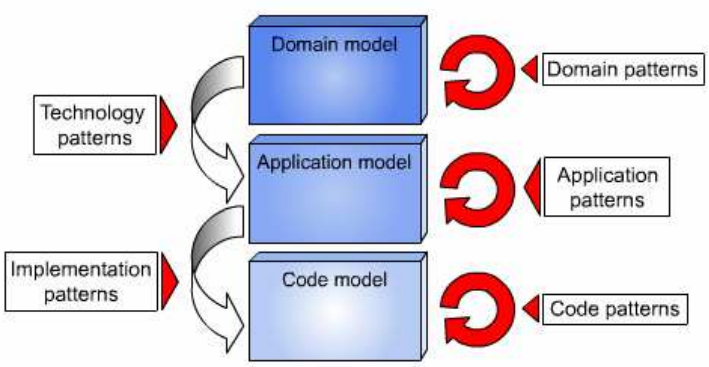


Figura 75. Patrones en OptimalJ, extraído de [6]

ArcStyler no pone tanto énfasis en los patrones como OptimalJ. Cada *MDA-Cartridge* puede implementar los patrones de diseño que desee, pero en la documentación de los *cartridges* utilizados [16, 17] apenas se mencionan unos pocos patrones, como el *Front Controller* para los *Accessor* o el *Compact Bean* para los EJB.

Soporte para la regeneración de modelos (P10)

OptimalJ gestiona de manera excelente la regeneración de modelos. Además de generar los PSMs a partir del PIM, permite realizar el paso inverso con el PSM de base de datos, es decir, generar el PIM a partir de un PSM de base de datos.

Los cambios realizados manualmente tanto en los PSMs como en el código se conservan aunque se regenere toda la aplicación. A nivel de código, este aspecto se controla mediante la distinción entre **bloques libres** y **bloques protegidos**. Todo lo que el desarrollador escriba en los bloques libres se conserva en sucesivas regeneraciones de código, mientras que los bloques protegidos no deben ser modificados. En la Figura 76 vemos un fragmento de fichero generado por OptimalJ, con bloques libres y protegidos.

ArcStyler no dispone de PSMs, así que este criterio (P10) sólo es aplicable a la regeneración de código. Utiliza un mecanismo similar al de OptimalJ para asegurar que se conserven los cambios realizados en el código, sólo que lo que en OptimalJ se llamaban bloques libres aquí curiosamente tienen el nombre de **áreas protegidas**. En estas áreas protegidas el desarrollador puede extender el código genereado asegurándose de que los cambios se conservarán en sucesivas regeneraciones. En la Figura 77 se muestra un fragmento de fichero generado por ArcStyler, editado con la herramienta *JBuider*, donde puede verse un área protegida.

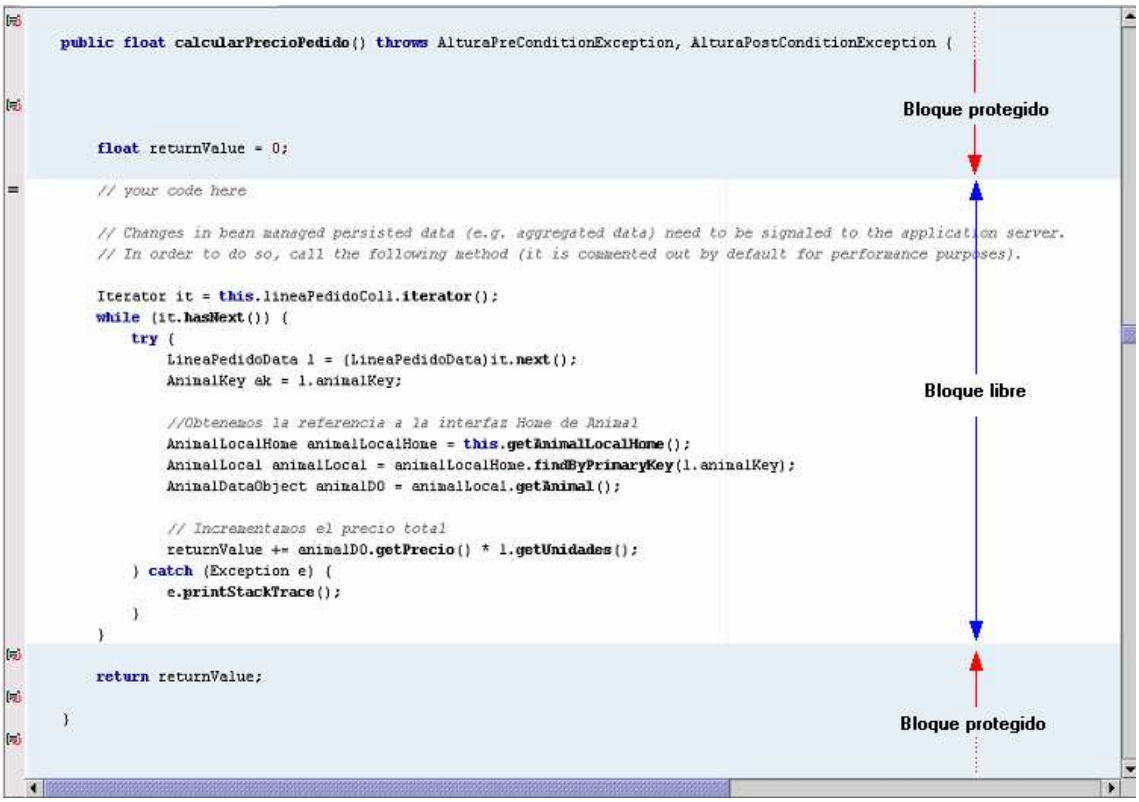


Figura 76. Bloques libres y bloques protegidos en OptimalJ

Transformaciones intra-modelo (P11)

OptimalJ utiliza transformaciones intra-modelo a nivel de PSM para trasladar algunos cambios realizados en un PSM a otros. Por ejemplo, un cambio realizado en el modelo de base de datos se refleja también en los modelos EJB y Web. No obstante, hemos observado que en la mayoría de ocasiones los cambios realizados en un PSM no se trasladan al resto.

En ArcStyler, la única transformación intra-modelo encontrada consiste en generar un modelo *Accessor* genérico a partir del modelo de clases. Esto es posible gracias los *Generic Accessors*, pertenecientes al framework *Accessor* [16]. Esta funcionalidad no ha sido probada con detalle, aunque puede resultar de gran utilidad para generar en poco tiempo una interfaz básica para la aplicación, que el desarrollador puede extender o modificar para adaptarla a sus requisitos específicos.

```
429   *
430   * @return short
431   *
432   */
433  public short calcularPrecioPedido()
434  {
435    /* START OF PROTECTED AREA <<calcularPrecioPedido>> */
436    // insert custom code here
437    /**@todo: Implement logic for calcularPrecioPedido() */
438    short total = 0;
439    try {
440      java.util.Iterator it = this.getLineasPedido().iterator();
441      while (it.hasNext()) {
442        LineaPedido lp = (LineaPedido) it.next();
443        total += lp.getUnidades() * lp.getAnimal().getPrecio();
444      }
445    } catch (Exception e) {
446      e.printStackTrace();
447    }
448    return total;
449    /* END OF PROTECTED AREA dec72e2100000073(C) */
450  }
451
452  /**
453   *
454   * Getter for association end <tt>cliente</tt>
```

Área protegida

Figura 77. Áreas protegidas en ArcStyler

Trazabilidad (P12)

Conocer el "camino" que ha seguido un elemento del modelo desde su origen en el PIM hasta su implementación en código resulta muy útil, por ejemplo, a la hora corregir errores o de evaluar el impacto de un cambio en el PIM o en los PSMs.

OptimalJ registra las transformaciones realizadas por un patrón de transformación en un paquete especial con el nombre del patrón. Gracias a este registro, podemos conocer para cada elemento del PSM su origen en el PIM (clase, atributo, etc.) y su destino en el código (fichero de código, paquete, script de base de datos, etc). La Figura 78 muestra el origen (*source*) y destino (*target*) del componente EJB *Cliente*.

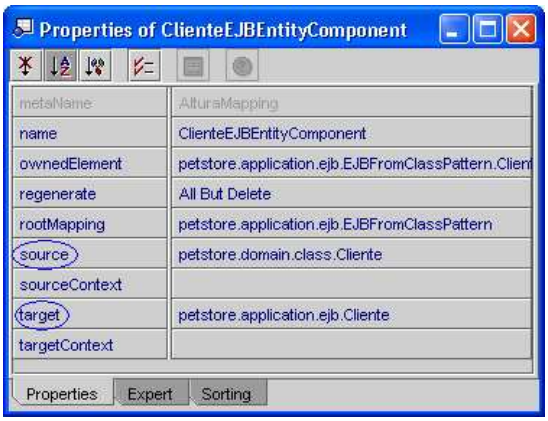


Figura 78. Origen y destino de un componente EJB (OptimalJ)

Las opciones de trazabilidad permitidas al usuario en ArcStyler son aún más limitadas, y se reducen a conocer todos los diagramas que contienen un determinado elemento del modelo. No permite saber, por ejemplo, el destino en el código de los elementos del

modelo. La Figura 79 muestra un ejemplo de esta opción, donde se muestran todos los diagramas en los que aparece la clase *Animal*.

Figura 79. Apariciones de un elemento del modelo en otros diagramas (ArcStyler)

Ciclo de vida (P13)

OptimalJ permite controlar prácticamente todas las fases del ciclo de vida sin necesidad de utilizar otras herramientas: análisis, diseño, codificación, depuración, prueba, despliegue y mantenimiento.

ArcStyler también abarca casi todas las fases del desarrollo, excepto la fase de codificación, ya que no dispone de un editor de código propio. No obstante, gracias al mecanismo de extensibilidad de ArcStyler podemos integrar en el entorno de la herramienta editores de código externos como *JBuilder*. Aunque no lo hemos probado, ArcStyler incorpora herramientas para el modelado de negocio y el modelado de requisitos como pasos previos a la construcción del PIM, algo que no es contemplado en OptimalJ, donde se parte de la fase de análisis elaborando el PIM.

Estandarización (P14)

Tanto OptimalJ como ArcStyler utilizan los principales estándares involucrados en MDA:

- UML como lenguaje de modelado.
- Repositorios MOF para el almacenamiento de modelos.
- XMI para el intercambio de modelos.

ArcStyler utiliza también el estándar JMI para definir el acceso a los modelos usando Java (véase JMI en pág. 60).

Control y refinamiento de las transformaciones (P15)

El control de las transformaciones en OptimalJ es bastante limitado, y se reduce a la modificación algunas de las propiedades disponibles en el cuadro de propiedades de determinados elementos del modelo, que pueden considerarse parámetros de transformación. Un ejemplo de estos parámetros de transformación lo tenemos en los

atributos de clases del PIM, cuyo cuadro de propiedades puede verse en la Figura 80. En este cuadro donde podemos establecer la longitud que tendrá el dato en la base de datos (*length*) o especificar una expresión en Java para el valor inicial (*initialValue*). El resto de propiedades en su mayoría no pueden considerarse parámetros de transformación, sino que son propiedades normales de cualquier atributo, como su multiplicidad (*multiplicity*), su nombre (*name*) o su tipo (*type*).

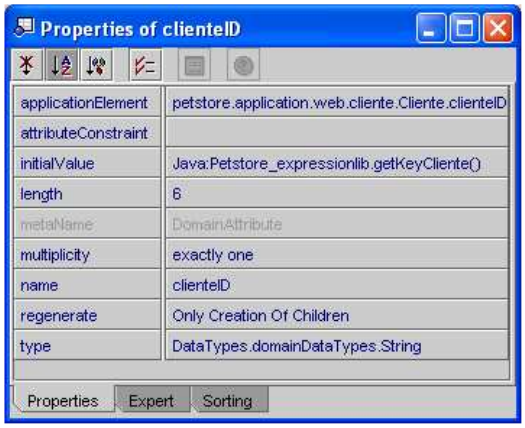


Figura 80. Cuadro de propiedades de un atributo del PIM (OptimalJ)

En ArcStyler, sin embargo, sí existe un soporte más completo para controlar y ajustar las transformaciones, gracias a las **marcas** asociadas a cada elemento del modelo. Estas marcas permiten controlar numerosos aspectos de la futura transformación, tantos como se definan en el *MDA-Cartridge* que se esté usando. Fijémonos por ejemplo en el completo conjunto de marcas para la tecnología EJB disponibles para una clase, mostrado en la Figura 81. Todas estas marcas definen distintos aspectos de la transformación de la clase a la plataforma EJB, como por ejemplo:

- *BeanType*: permite establecer el tipo de *bean* (*Session* o *Entity*).
- *GenLocalInterfaces*: indica si se generarán también las interfaces locales, además de las remotas.
- *PersistenceManagement*: establece el tipo de persistencia (CMP, BMP o la que proporciones por defecto el MDA-Cartridge).
- *StateManagement*: indica si el *bean* va será con o sin estado (*Stateful* o *Stateless*).

Esta gran variedad de marcas proporciona al desarrollador una gran versatilidad a la hora de dirigir el proceso de transformación, algo que en OptimalJ hoy por hoy no es posible.

El uso de marcas corresponde al enfoque de *transformación de instancias* que vimos en el apartado Definiciones de Transformación (pág. 23). ArcStyler usa ambos enfoque de *transformación de instancias* y *transformación de tipos* para generar el código de un modelo. OptimalJ, sin embargo, utiliza casi exclusivamente la *transformación de tipos*.

Calidad del código generado (P16)

Ambas herramientas generan código de calidad y bien documentado. No obstante, resulta más difícil modificar o extender los ficheros generados por OptimalJ, ya que se

utilizan numerosos patrones en el código que dificultan la legibilidad y que es necesario conocer antes de realizar modificaciones.

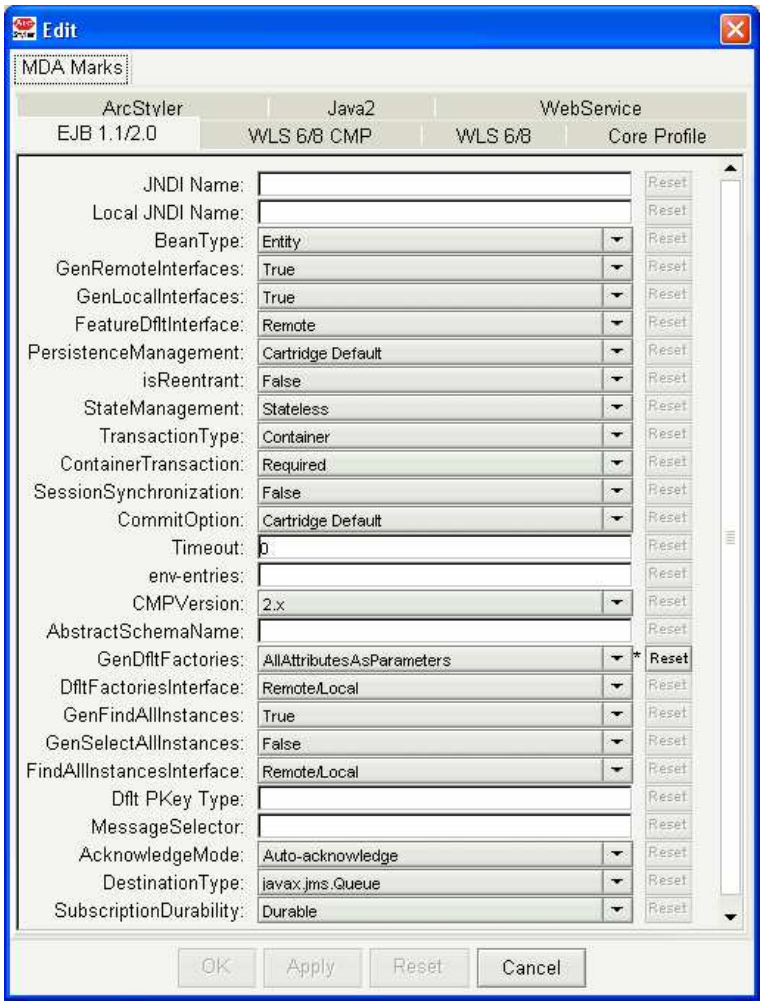


Figura 81. Panel de marcas de una clase para la tecnología EJB (ArcStyler)

Herramientas de soporte (P17)

Como ya dijimos, OptimalJ incorpora un editor de modelos, un editor de código, un servidor de bases de datos (*SolidDB*), un servidor de EJB (*JBoss*) y un servidor de JSP/servlets (*Tomcat*). No necesitamos herramientas de terceros en ninguna fase del desarrollo.

ArcStyler, por contra, no dispone de un editor de código propio, aunque ya comentamos que pueden integrarse otras herramientas en el entorno de ArcStyler. Tampoco dispone de un servidor de EJB ni de bases de datos, que han tenido que ser instalados de manera independiente. Aun así, mediante la herramienta ANT podemos iniciar o detener ambos servidores desde el entorno de ArcStyler.

8.2 Otros aspectos

Al margen de los aspectos directamente relacionados con MDA, existen otros aspectos que también influyen en la calidad de la herramienta. Comentaremos estos otros aspectos con detalle en los próximos apartados.

Rendimiento y estabilidad

Tanto OptimalJ como ArcStyler poseen grandes **requisitos de memoria**, necesitando un potente ordenador para realizar el desarrollo y la prueba del software. Debemos destacar que la evaluación de las herramientas se ha llevado a cabo en un ordenador con procesador PentiumIV a 2400 MHz y con 512 MB de RAM.

Aun con estas buenas prestaciones, en la fase de prueba los 512 MB de RAM resultan insuficientes en muchos casos, sobre todo con ArcStyler. En esta fase de prueba es donde realmente se consume más memoria, puesto que debemos tener en funcionamiento simultáneamente numerosas aplicaciones:

- La herramienta MDA (OptimalJ o ArcStyler) para realizar cambios en los modelos.
- Un servidor EJB para desplegar los componentes EJB.
- Un servidor de bases de datos para almacenar y recuperar los datos de la aplicación.
- Un servidor de JSPs/servlets para probar la interfaz de usuario.
- Para ArcStyler, necesitamos también una herramienta IDE como *JBuilder*, ya que ArcStyler no dispone de un editor de código integrado en la herramienta.

Para el caso de ArcStyler, la falta de memoria hace que cada prueba tarde varios minutos, ya que en muchas ocasiones es necesario reiniciar los servidores de JSPs y de EJB. Esto enlentece el desarrollo de la aplicación y la corrección de errores.

Por otro lado, el gran consumo de memoria origina que en ocasiones las herramientas se vuelvan **inestables** y no respondan a las acciones del usuario, aunque generalmente se recuperan tras varios segundos, sin que se produzca pérdida de información.

Facilidad de uso

Si algo distingue a OptimalJ de ArcStyler es la **sencillez** en el manejo de la aplicación. Usar la herramienta OptimalJ no es muy diferente de usar cualquier herramienta de modelado software, incluso podríamos decir que en la mayoría de ocasiones es más sencillo. No dispone de demasiadas opciones, los menús son bastante intuitivos y en poco tiempo podemos familiarizarnos con la herramienta.

Para facilitar la creación de la mayoría de elementos del modelo, tanto a nivel de PIM como de PSM, OptimalJ posee asistentes (*wizards*), que nos guían mediante una serie de pasos en el proceso de creación del elemento. En la Figura 82 podemos ver uno de estos asistentes.

Otra ventaja de OptimalJ es que resulta extremadamente sencillo realizar una aplicación básica. En cuestión de horas podemos conseguir una aplicación de mantenimiento (creación, recuperación, modificación y eliminación de instancias) totalmente operativa a partir de un simple PIM. Bien es cierto que esta sencillez es a menudo un obstáculo, ya que resulta muy complicado realizar modificaciones sobre la aplicación generada, debiendo alterar en la mayoría de ocasiones el código fuente.

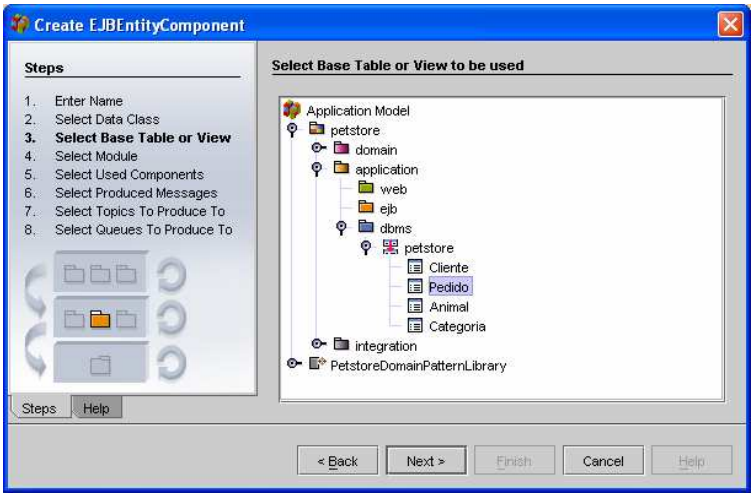


Figura 82. Asistente para la creación de un componente EJB de tipo entidad (OptimalJ)

En el extremo opuesto tendríamos ArcStyler. Familiarizarse con el uso de esta herramienta es realmente complicado, no solo por las múltiples opciones que ofrece, sino porque hace uso de otras herramientas y frameworks que también debemos conocer, como la herramienta ANT para construir la aplicación generada, el framework JUnit para realizar pruebas unitarias o el framework *Accessor* para construir interfaces de usuario.

Construir aplicaciones en ArcStyler resulta mucho más complejo que en OptimalJ, entre otras cosas porque recae más peso en el desarrollador, que debe definir muchos más aspectos de la/s plataforma/s de destino. Un ejemplo lo tenemos en el modelo Web: mientras en OptimalJ este modelo se genera automáticamente a partir del PIM y desde el primer momento genera código operativo, en ArcStyler debemos definir completamente el modelo Web, y a un nivel de detalle mucho mayor que el de OptimalJ.

Una gran deficiencia encontrada en el manejo de OptimalJ es que carece de la **opción *deshacer*** (*undo*), algo que no ocurre en ArcStyler. La ausencia de esta opción dificulta seriamente la labor del desarrollador, ya que cada cambio equivocado hay que deshacerlo manualmente, y en ocasiones no podemos volver exactamente al estado anterior. Por ello, se hace necesario realizar copias de seguridad del proyecto antes de cualquier cambio importante.

Documentación

Además del manual oficial de la herramienta, OptimalJ pone a disposición de los usuarios otra guía con tutoriales sencillos y explicados paso a paso de cómo realizar

tareas muy comunes, que facilitan y amenizan sin duda el aprendizaje de la herramienta. También es cierto que el manual oficial resulta insuficiente a la hora de realizar tareas complejas. Para solucionar este problema la empresa *Compuware* ha elaborado algunos **artículos técnicos** (disponibles en la página Web de la compañía) que resuelven problemas complejos planteados por los usuarios.

Por otro lado, *Compuware* pone a disposición de los usuarios un **foro de discusión** muy activo en su página Web, donde pueden plantearse dudas concretas sobre el uso de la herramienta. Este foro nos ha sido de gran utilidad, ya que muchas de las dudas que se nos han planteado estaban ya resueltas en dicho foro.

ArcStyler dispone de numerosos manuales, tanto para el manejo de la aplicación como para el uso de cada *MDA-Cartridge*. Tanta documentación llega a ser desconcertante en un principio, pues no está muy claro por donde debe empezar un usuario "novato". En cualquier caso, una vez familiarizados con la herramienta estos manuales resultan de gran utilidad. También existen tutoriales para algunos *MDA-Cartridges*, como los de EJB y *Accessor*.

Aunque los manuales son, por lo general, bastante completos, sería muy útil disponer de un foro para realizar consultas específicas sobre la herramienta.

9 Conclusiones y Trabajo Futuro

De nada sirve todo lo propuesto por MDA sin herramientas que den soporte a este nuevo framework. En este proyecto hemos evaluado dos de estas herramientas, obteniendo muy buenos resultados.

OptimalJ es una herramienta sorprendente en un principio, pues en pocas horas permite crear de forma sencilla una aplicación básica para la plataforma J2EE a partir de un simple diagrama de clases, generando código de calidad y aplicando los mejores patrones de diseño. Sin embargo, tiene serias carencias que todavía necesita mejorar para permitir la construcción de aplicaciones a medida, sobre todo en lo referente al modelado de interfaces de usuario.

ArcStyler, al igual que OptimalJ, permite un desarrollo basado en modelos. No obstante, creemos que OptimalJ refleja más fielmente el proceso MDA, con una clara separación entre el PIM y los PSM que ArcStyler no soporta. En ArcStyler, el PIM se transforma directamente en código sin que exista un PSM que pueda ser modificado por el desarrollador. Aun así, ArcStyler permite numerosas implementaciones mediante los *MDA-Cartridges*, y el diseño de interfaces de usuario es excelente gracias al framework *Accessor*. Hoy por hoy se trata de una **herramienta más madura y completa** que OptimalJ, ya que da al desarrollador una mayor versatilidad a la hora de describir los modelos, dirigir las transformaciones y elegir distintas implementaciones. Además, su estructura abierta permite extender la aplicación con nueva funcionalidad, por ejemplo, integrando herramientas externas en el entorno de desarrollo o añadiendo nuevos *MDA-Cartridges* a los disponibles.

En definitiva, en OptimalJ se define un proceso MDA completo para la plataforma J2EE, estableciendo perfectamente los puentes entre los tres PSMs generados y generando desde el primer momento una aplicación operativa. Por contra, ArcStyler sigue una filosofía de herramienta MDA "universal", abierta a la incorporación de cualquier tecnología y con mucha mayor flexibilidad a la hora de describir el sistema, a cambio de exigir más trabajo del desarrollador para construir la aplicación final.

MDA está en su infancia y todavía queda mucho por mejorar, pero podemos afirmar que **MDA ya es una realidad**. Tanto OptimalJ como ArcStyler posibilitan un desarrollo de software centrado en modelos de alto nivel. Estas herramientas ahorran la mayor parte del proceso de codificación, generando más del 90% del código de la aplicación, de modo que el desarrollador sólo tiene que añadir al código aquella funcionalidad que no puede expresarse mediante modelos. Es más, en un futuro cercano, esta fase de codificación podría llegar a desaparecer, dejando todo el peso de la codificación a las herramientas MDA.

En futuros trabajos sería interesante realizar un estudio más completo de la herramienta ArcStyler, profundizando en los siguientes aspectos:

- Estudio de los modelos relacionados con el **modelado de negocio** y el **modelado de requisitos**.
- Estudio de la arquitectura de cartuchos de ArcStyler (***CARAT***) y creación de un cartucho.

- Estudio más profundo **framework** ***Accessor***, de cara a integrar este framework en un proceso de desarrollo dirigido por casos de uso.

Al margen del estudio de ArcStyler, otros trabajos interesantes podrían ser:

- Profundizar en el estándar **QVT** y en los lenguajes de transformación de modelos.
- Estudio de **UML 2.0** para analizar las extensiones orientadas a favorecer MDA: definiciones de *profiles* y aspectos relacionados con la capacidad expresiva de los diagramas de interacción.
- Estudio del lenguaje de definición de patrones en **OptimalJ** y creación de nuevos patrones de implementación.

Bibliografía

1 Berenguer, D.P., *Desarrollo de Aplicaciones Web con Struts*. 2002, Murcia.

2 Booch, G., A. Brown, and J. Rumbaugh, *An MDA Manifesto.* 2004, IMB Rational Software.

3 Compuware, *OptimalJ 3.0*. 2004, http://javacentral.compuware.com/

4 Compuware, *Using OptimalJ 3.0*. 2003.

5 Compuware, *Using OptimalJ 3.0: Tutorials*. 2003.

6 Compuware, *Model Driven Architecture in OptimalJ*. 2003. http://javacentral.compuware.com/demos/mda.html

7 DeMichiel, L.G., L.Ü. Yalçinalp, and S. Krishnan, *Enterprise JavaBeans Specification, Version 2.0*. 2001.

8 Frankel, D., *Model Driven Architecture. Applying MDA to Enterprise Computing*, 2003, Wiley.

9 Fuentes, L. and A. Vallecillo, *Una Introducción a los Perfiles UML*. 2004. http://www.lcc.uma.es/~av/Publicaciones/04/UMLProfiles-Novatica04.pdf

10 Gamma, E., et al., *Design Patterns*. 1994, Addison-Wesley.

11 Hubert, R., *Convergent Architecture*. 2002, Wiley Computer Publishing. http://convergentarchitecture.com

12 Interactive Objects, *ArcStyler 4.0.90*. 2004. http://www.arcstyler.com/

13 Interactive Objects, *ArcStyler Platform Guide For ArcStyler Version 4.0*. 2003.

14 Interactive Objects, *ArcStyler Modeling Style and User's Guide for ArcStyler Version 4.0*. 2003.

15 Interactive Objects, *ArcStyler Accessor Tutorial for ArcStyler Version 4.0*. 2003.

16 Interactive Objects, *ArcStyler Accessor Guide for ArcStyler Version 4.0*. 2003.

17 Interactive Objects, *ArcStyler MDA-Cartridge Guide for BEA Weblogic Server 8.1 for ArcStyler Version 4.0*. 2003.

18 Interactive Objects, *MDA Cartridge Development and Extensibility Guide for ArcStyler Version 4.0*. 2003.

19 Kleppe, A., J. Warmer, and W. Bast, *MDA Explained*. 2003, Addison-Wesley.

20 King's College London, *An Evaluation of Compuware OptimalJ Professional Edition as an MDA Tool*. 2003.

21 Miller, J. and J. Mukerji, *MDA Guide Version 1.0.1*. 2003.

22 Miller, J. and J. Mukerji, *Model Driven Architecture (MDA)*. 2001.

23 Object Management Group, *MetaObjectFacility (MOF) Specification, Version 1.4*. 2002.

24 Object Management Group, *Model Driven Architecture. A Technical Perspective*. 2001.

25 Object Management Group, *UML 2.0 OCL Specification*. 2003.

26 Poole, J.D., *Model-Drivern Architecture: Vision, Standards And Emerging Technologies*. 2001, Hyperion Solutions Corporation.

27 QVT-Partners, *Revised submission for MOF 2.0 Query / Views / Transformations RFP*. 2003.

28 Soley, R., *Model Driven Architecture*. 2000, OMG.

29 SUN Microystems, *Java Pet Store*. 2004. http://java.sun.com/developer/releases/petstore/

30 SUN Microystems, *Java™ 2 Platform Enterprise Edition Specification, Versión 1.4*. 2003.

31 The Middleware Company, *Model Driven Develpment for J2EE Utilizing a Model Driven Architecture (MDA) Approach*. 2003.