Universidad de Murcia
Facultad de Informática

CARTUCHOS MDA EN ARCSTYLER 4.0

Autor:
Joaquín Lasheras Velasco
jolave@um.es

Departamento de Informática y Sistemas
Becario FPI

Curso de Doctorado: Arquitectura del Software
Responsable: Jesús Joaquín García Molina
Programa: Matemática e Informática Aplicadas en Ciencias e Ingeniería
Bienio 2003/2005

1 Introducción

ArcStyler de Interactive Objects es una herramienta que implementa MDA, permitiendo desarrollar software de manera efectiva y eficiente mediante el uso de modelos. Para poder adaptar los distintos PIM a determinadas tecnologías se propone la utilización de Cartuchos (Cartridges) los cuales implantan las reglas de transformación, e incluso en algunos casos, de verificación de modelos.

En este trabajo vamos a explicar en que consisten estos Cartuchos, como pueden ser creados y modificados para ampliar su funcionalidad y para lo cual, pondremos el ejemplo del desarrollo de uno muy sencillo para generar código fuente Java. Para ello utilizaremos el desarrollo de Cartuchos dirigido por modelos, donde aplicaremos los principios de MDA para el desarrollo, mantenimiento y extensión de los Cartuchos-MDA de Arcstyler.

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, es decir, usar modelos para especificar reglas de transformación.

Para la realización de este trabajo hemos utilizado la versión de ArcStyler 4.0.90 y nos hemos basado como documentación principal en la ayuda proporcionada por la herramienta.

Los dos puntos de extensibilidad posibles que proporciona ArcStyler son:

- El ArcStyler MDA Cartridge Engine, a través del cual los Cartuchos pueden ser añadidos y por lo tanto contribuir con reglas de transformación de modelos y verificación, junto con la información del profile UML correspondiente, la definición de las marcas MDA y los tipos de datos requeridos por el cartucho. También existe la posibilidad de adaptar alguno de los cartuchos disponibles.

- Arcstyler Tool Adapter Standar (TAS), cuyo alcance queda fuera de este trabajo y es el API que ofrece ArcStyler para escribir plug-ins.

2 Cartuchos MDA

Con el "Editor de Arquitecturas de ArcStyler" y su IDE para Cartuchos MDA, 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. Por ello, va a permitir a una organización visualizar, desarrollar, testar y desplegar soporte MDA para mantener sus requisitos especificados, tanto si estos son extensiones para soporte MDA existente o para tipos de infraestructuras y arquitecturas completamente nuevas.

ArcStyler permite transformaciones de modelos (modelo a modelo o modelo a código) de acuerdo con la arquitectura dirigida por modelos de OMG (MDA). Las transformaciones de modelos toman un modelo de entrada y un conjunto de marcas y mapean este a uno o más modelos de salida, código de salida u otra infraestructura. Los modelos son almacenados en un repositorio el cual podría ser light-weight con implementación basada en XMI o un repositorio sofisticado multiusuario distribuido y transaccional (ver figura 1). Los metamodelos definidos

para el modelo fuente y el destino pueden diferir. En ArcStyler se utiliza el metamodelo de UML 1.4. Adicionalmente – para mantener la compatibilidad con Cartuchos anteriores – el metamodelo C-MOD está también soportado. Este es una variante del metamodelo de UML 1.3.

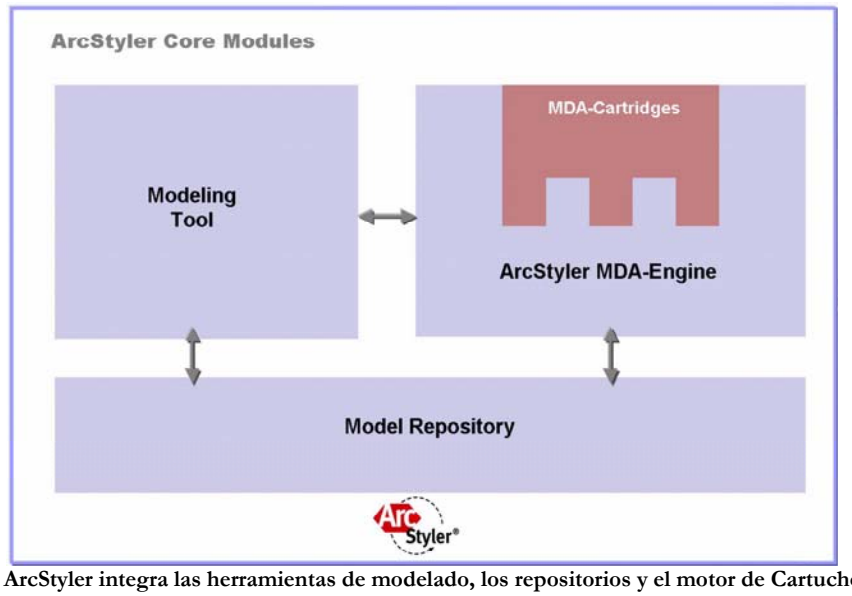

Figura 1. ArcStyler integra las herramientas de modelado, los repositorios y el motor de Cartuchos MDA

Una transformación de modelo (modelo a modelo o modelo a código) con ArcStyler es implementada en un MDA-Cartridge o Cartucho MDA. Además de las reglas de transformación, un Cartucho MDA puede proporcionar reglas de verificación, las cuales, cuando son aplicadas a un modelo de entrada, chequean el modelo para comprobar si es correcto con respecto a la transformación implementada por el cartucho MDA.

Un cartucho puede usar el repositorio como salida, pero también se puede utilizar los servicios del Motor MDA para la gestión de artefactos basados en texto. Esto incluye la gestión la gestión de cambios, control de versiones y la gestión de "areas protegidas" (secciones en el fichero texto donde los cambios manuales son preservados).

Un conjunto de Cartuchos MDA estándar están disponibles con ArcStyler. Estos Cartuchos cubren el soporte MDA completo para infraestructuras estándar ampliamente difundidas como son Java2, J2EE o .NET. Estos Cartuchos pueden ser libremente intercambiados usando el "MDA Cartridge Source Forge" en http://www.mda-at-work.com que también es accesible desde la propia herramienta ArcStyler. En la figura 2 podemos ver como también hacen uso de la herencia entre los cartuchos.

Además de la transformación y la verificación de modelos, un cartucho MDA puede incorporar asistentes o *wizards* que se integran en la herramienta de modelado de ArcStyler. Por ejemplo, el cartucho de EJB proporciona un asistente que facilita la creación de un elemento del modelo que representa a un componente EJB. Además cada uno de estos cartuchos suele incluir una ayuda para saber como utilizarlos o modificarlos.

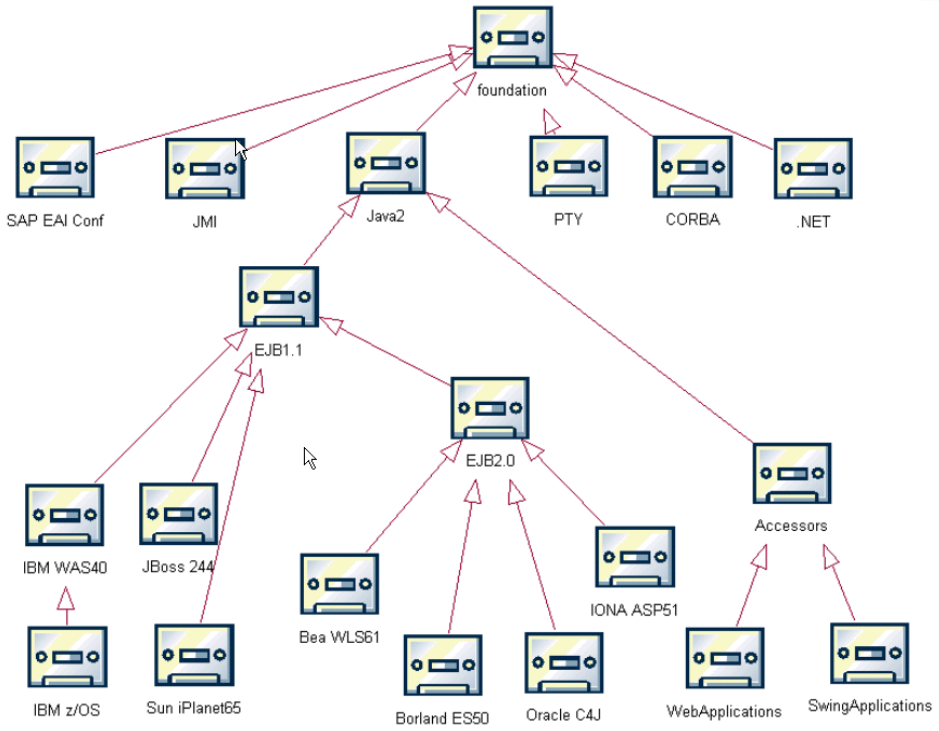

Figura 2. Jerarquía de los cartuchos MDA por defecto en ArcStyler. Herencia de Cartuchos.

Cada cartucho MDA define un estilo de modelado para especificar la estructura y precisión de los modelos de entrada para la transformación proporcionada por el cartucho MDA. Los verificadores en el cartucho MDA son usados para chequear los modelos existentes y ver si cumplen con el estilo de modelado. En muchos casos los objetivos son artefactos basados en texto, como nuestro ejemplo del código fuente de Java (transformaciones modelos a código). Sin embargo también se pueden desarrollar cartuchos para transformaciones modelo a modelo. Por ejemplo, los cartuchos Web Accessors de MDA aceptan modelos de componentes del negocio (*UML classifiers*) y los mapean a diagramas de estados UML constituyendo un modelo bien-formado para aplicaciones Web y servicios Web que sirven al componente del negocio seleccionado.

Las reglas de transformación implementadas en un cartucho MDA proporcionan soporte específico para tipos de modelos y estilos de arquitectura particulares y para su mapeo optimizado a infraestructuras o aplicaciones objetivo. Mientras que las Cartuchos MDA estándar en ArcStyler cubren un amplio rango de escenarios de desarrollo de aplicaciones y diseños, hay muchos escenarios en los cuales las reglas de transformación en un cartucho MDA pueden necesitar ser modificadas o extendidas y otros casos en los cuales un conjunto entero y nuevo de reglas de transformación necesitará ser aplicado para soportar el estilo arquitectónico cliente. Una de las virtudes de ArcStyler es su soporte para la creación y extensión de Cartuchos MDA como extensiones MDA usando el Editor de Arquitecturas y el IDE para Cartuchos MDA (La arquitectura CARAT the CARtridge ArchiTecture).

Los Cartuchos MDA de ArcStyler usan Jphython como su lenguaje de script. ArcStyler utiliza actualmente Jpython versión 1.0.3. En este trabajo no queremos meternos a fondo con este lenguaje pero a modo de resumen debemos decir que Jphython es un lenguaje de

programación interpretado, interactivo y orientado a objetos. Es usualmente comparado a Tcl, Perl, Scheme or Java. Jpython maneja módulos, clases, excepciones, tipos de datos dinámicos de alto nivel y ligadura dinámica. Hay interfaces para muchas librerías y llamadas a sistema, al igual que para varios sistemas de ventanas (x11, Mac …). JPhyton es también utilizable como lenguaje de extensión para aplicaciones que necesitan una interfaz programable. Más información podemos ver en **http://www.python.org/doc**.

En las siguientes secciones vamos a poner como ejemplo el desarrollo de un cartucho simple MDA para ver las características de la Arquitectura de Cartuchos de ArcStyler (CARAT).

3 Creando un Cartucho MDA en ARCSTYLER

3.1 Transformaciones de Modelo y CARAT

Un Cartucho MDA para ArcStyler sirve para el propósito de transformar un modelo fuente a modelo objetivo. Modelos fuente basados en UML y modelos objetivo basados en texto son típicas combinaciones, por ejemplo cuando el código fuente es generado desde UML. Sin embargo hay otras posibles combinaciones.

Los modelos consisten de un conjunto de elementos del modelo. Por ejemplo, un modelo UML consiste de clases, operaciones, atributos, asociaciones, etc., mientras que un modelo basado en texto se compone de ficheros ASCII que consisten en una secuencia de caracteres ASCII los cuales pueden o no atenerse a una sintaxis que prescriba sus estructuras internas. Un modelo de transformación deberá especificar como estos elementos del modelo fuente afectan a la creación de los elementos del modelo objetivo. Proporcionar esta especificación es el propósito principal de un Cartucho MDA.

Por todo ello el desarrollo de modelos de transformación supone los mismos retos que el desarrollo de cualquier otra aplicación que no tenga una complejidad y tamaño trivial. Usando arquitecturas y diseños ad-hoc típicamente nos llevan a resultados difíciles de entender, duros de mantener y casi no adaptables, extensibles y reutilizables. Es por ello que, ArcStyler CARAT proporciona una bien definida arquitectura para el desarrollo de transformaciones de modelos. La arquitectura CARAT parte de los principios de la ingeniería orientada a objetos y toma estos conceptos al nivel de transformaciones de modelos. En consecuencia, MDA en sí mismo es aplicado al desarrollo de los modelos de transformación. Esto significa que las transformaciones de modelos son expresados como modelos en sí mismos – un cartucho específico (el cartucho CARAT) toma estos modelos y los traslada a programas ejecutables (El cartucho CARAT es generado usándose así mismo). Las distintas versiones de Arcstyler han pasado por diversas versiones de CARAT. Actualmente tenemos la versión 4 de ArcStyler y la 2.0 de CARAT. En ella los Cartuchos son modelados usando UML con un profile CARAT especial. Se generan usando el cartucho CARAT. En la figura 3 podemos ver un esquema de cómo quedaría.

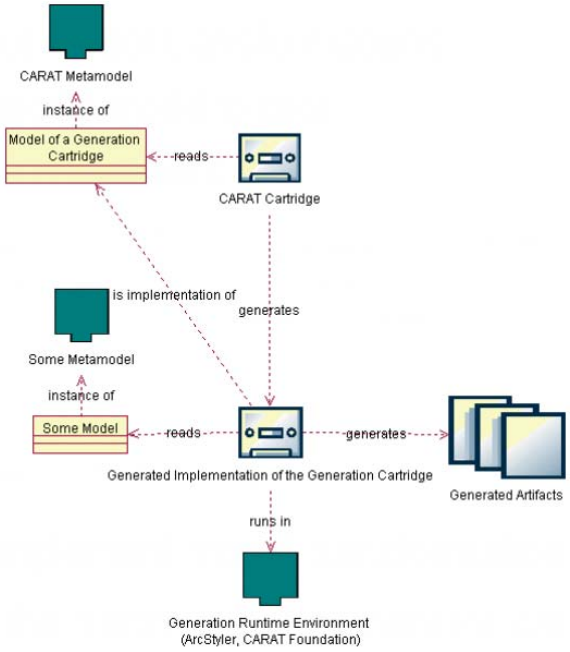

Figura 3. Generando Cartuchos con un cartucho

Vamos a introducir ahora los conceptos CARAT a partir de un ejemplo.

3.2 Un Cartucho Simple, un Fichero de Código Fuente Java

En el ejemplo elegido, un cartucho es desarrollado para tomar un modelo de entrada UML y generar clases y interfaces Java para cada clase UML. Las interfaces deberán tener el nombre de la clase UML correspondiente (asumiendo que estas tienen un identificador Java válido). Las interfaces deben estar localizadas en paquetes java que corresponden al *path* del paquete respectivo en el modelo UML. Para cada atributo en la clase UML, las declaraciones de los métodos *getter* y *setter* deben ser generadas en la interfaz. Las clases generadas deben estar localizadas en el mismo paquete java como las interfaces java, teniendo el nombre de la clase UML más un sufijo '*Impl*'. Las clases Java deben definir variables miembro para cada uno de sus atributos y las implementaciones apropiadas de los métodos *getter* y *setter* (figura 4).

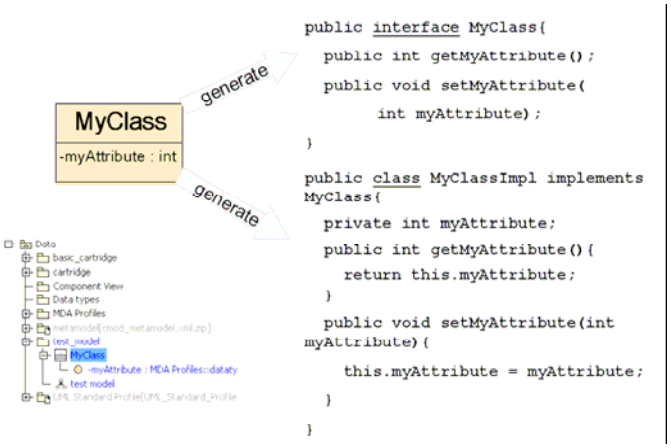

Figura 4. Entrada y salida del cartucho ejemplo

3.3 Principios de las Transformaciones

Hay algunos pasos intuitivos para la realización de la transformación requerida:

- Iterar sobre el modelo fuente y visitar cada clase. Esto significa que el modelo y todos los paquetes que este contiene necesitan ser recorridos recursivamente y cada clase será visitada.
- Para cada clase se generarán dos artefactos. Cuando hacemos referencia a su clase y estructura, todas las entidades generadas son llamadas artefactos. En el ejemplo, los artefactos son ficheros de código fuente Java pero también podrían haber sido elementos del modelo. Los ficheros fuente Java necesitan tener el nombre correcto asignado y necesitan estar situados en una estructura de directorios de acuerdo a la estructura del paquete en el modelo fuente.
- Cuando generamos el artefacto interfaz, hay que iterar sobre los atributos de la clase UML y generar las declaraciones de los métodos *getter* y *setter*.
- Cuando generamos el artefacto implementación, primero generamos la definición de las variables miembro y después las definiciones de los métodos de acceso. Por lo tanto, iterar sobre los atributos de la clase UML y generar una variable miembro para cada atributo. Iterar sobre los atributos de nuevo y generar un método *getter* y *setter* para cada uno de ellos.

Este intuitivo acercamiento revela los aspectos principales que una transformación de modelo tiene que manejar y de estos se pueden derivar los aspectos principales de CARAT.

- Operaciones de iteración y navegación son realizadas: La iteración es usada para buscar por los elementos del modelo que causan la generación de uno o mas artefactos en el espacio objetivo (por ejemplo, clases en el modelo UML causan la generación de artefactos de código fuente). Cuando generamos un artefacto, la navegación según el metamodelo puede ser utilizada para ir más lejos en la exploración del modelo (por ejemplo, recorriendo los atributos de la clase con objeto de generar los métodos *getter* y *setter* dentro de la generación de la clase artefacto).

- El espacio objetivo es estructurado: normalmente los artefactos están unidos conceptualmente (por ejemplo, los artefactos de clase e implementación Java van juntos y son generados a la vez para el mismo elemento del modelo). Además, los artefactos están estructurados internamente (por ejemplo el artefacto de la clase Java consiste de una cláusula paquete, una declaración de clase, una sección de variables miembro y una sección de operaciones).

- Los elementos del modelo son interpretados de forma diferente dependiendo del contexto: dependiendo del contexto de la generación, el mismo elemento del modelo puede aparecer en diferentes formas. Por ejemplo, una clase y particularmente su nombre en el modelo UML aparece como el nombre de la interfaz y como el nombre del artefacto clase implementación. Lo mismo se aplica para los atributos y sus nombres los cuales aparecen como miembros variables, y como nombres de métodos *getter* y *setter*.

3.4 Conceptos de CARAT ArcStyler

Las consideraciones realizadas anteriormente llevan a manejar las estructuras típicas para un Cartucho MDA que consisten de 4 conceptos clave:

- **Artefactos CARAT**: son elementos del modelo objetivo que son creados durante la transformación. En el caso de modelos objetivo basados en texto, este artefacto sería un fichero, mientras que para modelos objetivo basados en UML este podría ser un *classifier*.

- **Secciones de artefactos CARAT** (también llamados "CARAT Generatables"): son elementos del modelo objetivo que están subordinados a los artefactos y no pueden existir solos. En el caso de modelos objetivo basados en texto, una sección del artefacto identifica una región en un fichero. En un modelo objetivo basado en UML, el conjunto de atributos que un *classifier* tiene podría ser descrito como una sección del artefacto. Las secciones de los artefactos pueden ser anidadas y por esto describe una estructura jerárquica del modelo objetivo. La estructura definida por el artefacto y las secciones del artefacto usualmente refleja el metamodelo objetivo de la transformación.

- **Conjuntos de artefactos CARAT**: son grupos de artefactos que son siempre creados juntos, *triggered* por alguna condición. Esta condición esta basada en el modelo fuente y opcionalmente en información de configuración adicional. Por ejemplo, si un Cartucho MDA implementa una transformación desde un modelo UML describiendo un componente EJB con un *classifier* simple a un conjunto de ficheros fuente Java y *deployment descriptors* entonces habría que tener un conjunto de artefactos que son *triggered* por un *classifier* de esta clase que representa un componente EJB. Este conjunto de artefactos agruparía los artefactos responsables para la creación de los fuentes java de la interfaz home, la interfaz remota, la clase *key*, la clase *bean* y el *deployment descriptor*.

- **CARAT Bluprints** son una aplicación del patrón Decorator GoF. (Gamma, Helm, Johnson, Vlissides: "Design Patterns", Addison-Wesley, 1995). Los Blueprints implementan el conocimiento de como las instancias de un elemento del metamodelo fuente en particular (por ejemplo, una operación o un atributo) son representados en los diferentes contextos definidos por los artefactos y las secciones de artefactos. Como una consecuencia de que los blueprints son decoradores para los elementos del modelo en un cartucho, hay usualmente un blueprint por elemento "tipo" del modelo (por ejemplo, por elemento en el metamodelo). En nuestro ejemplo, habría un Blueprint para el elemento del metamodelo UML Clase (llevando la representación de una instancia de clase UML en varios contextos, como en el contexto de generación de clases Java e interfaces java) y habría un blueprint para el elemento del metamodelo UML atributo (llevando la representación de un atributo UML en varios contextos, como la generación de los métodos *getter* o *setter*)

3.4.1 Implementando un Cartucho Simple

Implementar un cartucho simple implica dos pasos principales: el primero, basándose en el estilo de modelado CARAT, la estructura del cartucho debe ser definida, la cual es alineada con

la estructura de los artefactos objetivo. Después, una vez generado el cartucho simple usando el cartucho CARAT la implementación del cartucho es refinada manualmente usando el IDE (*Integrated Development Environment*) de cartuchos.

Configuración del Cartucho MDA

En primer lugar, un nuevo proyecto MDA debe ser creado usando la plantilla de proyecto CARAT. Usando la plantilla del proyecto nos aseguramos de que es cargado el soporte necesario para el profile UML y el cartucho CARAT. Para el cartucho MDA es creado un paquete, de tal forma que este puede ser almacenado separadamente y reusado por subcartuchos. La relación de herencia con el cartucho MDA "**Foundation**" esta especificada. Como las clases en Java, los Cartuchos CARAT heredan de otros cartuchos. El cartucho base usado en el ejemplo es el cartucho base por defecto CARAT "**Foundation**" (el cual es comparable a *java.lang.Object*). En la invocación de un cartucho, el proceso de iteración del modelo tiene que ser comenzado. Por ello, un "**RootIterator**" es asociado a través de la asociación "**FeatureRoot**". Además, la clase *utility* "**BasicUtil**" es creada en el modelo y asociada con el cartucho MDA. Como este es el nombre de clase *utility* por defecto, no es necesaria especificación adicional en la hoja de propiedades del *classifier* en el cartucho MDA. El modelo resultante es mostrado en la figura 5.

Estructura del Artefacto

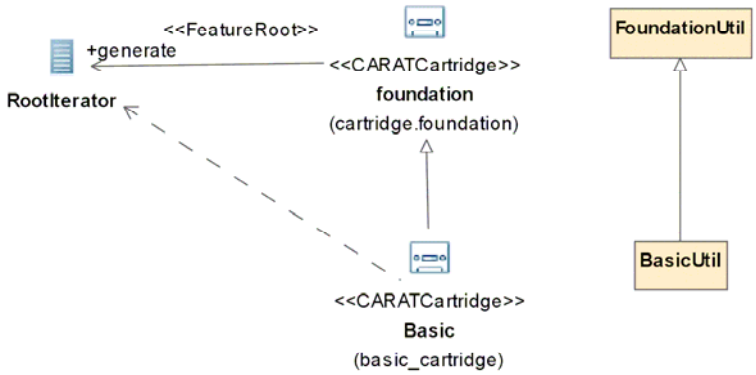

Figura 5. Configuración de un cartucho básico

El cartucho MDA especificado por el modelo generará dos ficheros fuente java (los artefactos clase e interfaz) para cada elemento clase de UML examinado. Esto es, un **Conjunto de Artefactos** simples es modelado de tal forma que contendrá (por ahora) dos artefactos - un **artefacto** para la generación del fichero fuente de la interfaz java para cada *classifier* y el otro correspondiente para el fichero de clase Java. El conjunto de artefactos especifica una expresión **"handlesElement**" que filtra los objetos clase de UML con un estereotipo. Las marcas (que se verán después) del artefacto proporcionan **InterfaceArtifact.tpl** y **ClassArtifact.tpl** respectivamente como el nombre de los ficheros plantillas usados para generar los artefactos (figura 6).

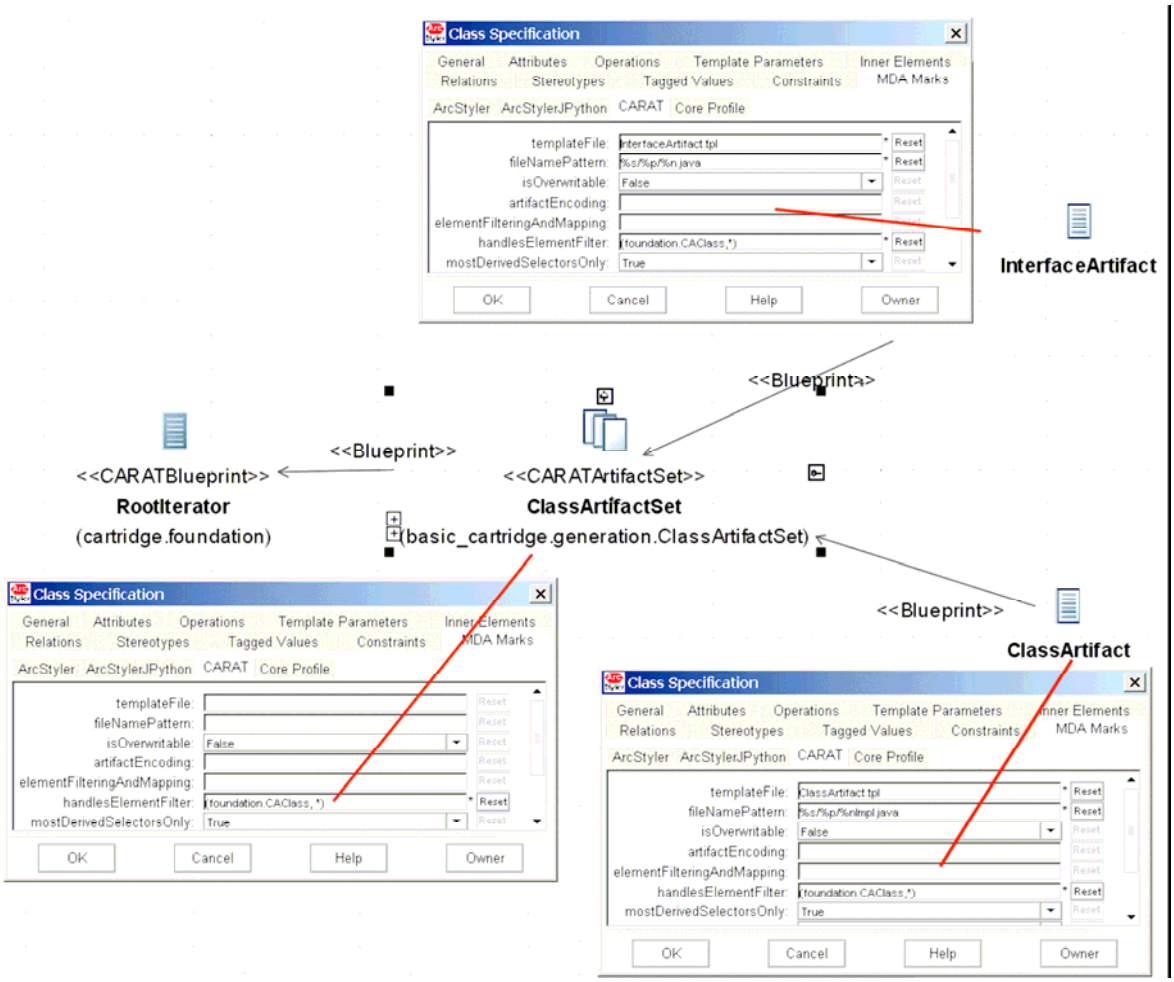

Figura 6. Artefactos de un cartucho básico

Hay que destacar los estereotipos de las asociaciones. Los artefactos **InterfaceArtifact** y **ClassArtifact** están registrados con el **ArtifactSet** usando una asociación **<<blueprint>>**.

A continuación, la estructura del **InterfaceArtifact** y del **ClassArtifact** necesita ser definida. Las **secciones del artefacto** son usadas para estructurar los artefactos. El **InterfaceArtifact** consta de una sección que contiene la declaración de la interfaz (**InterfaceDeclaration**) y otra sección (**InterfaceMemberAccess**) que contiene las declaraciones de los métodos de acceso (figura 7).

La estructura de **ClassArtifact** es similar a la estructura de **InterfaceArtifact** – contiene una sección para la declaración de la clase, una sección para la implementación de los métodos de acceso miembros y una sección adicional que contiene las definiciones de las variables miembro (figura 8).

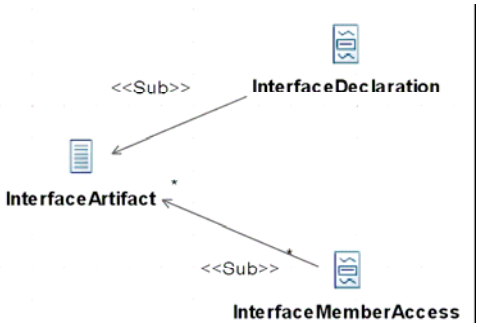

Figura 7. Estructura del artefacto intefaz

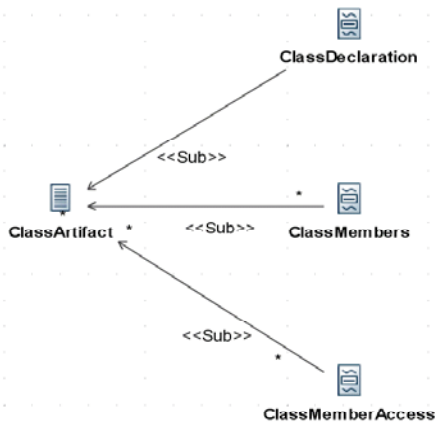

Figura 8. Estructura del artefacto clase

Dar Contenido a la Estructura: Los Blueprints

El modelo del cartucho desarrollado no solo describe la estructura compuesta de un cartucho (por ejemplo, una composición de un iterador, conjuntos de artefactos, artefactos y secciones de artefactos). También describe el flujo de control de la transformación en tiempo de ejecución, una vez generado: Para cada elemento del modelo recorrido por el iterador, el control y el elemento del modelo actual (por ejemplo, el sujeto de la transformación) es pasado secuencialmente a los siguientes sub-compuestos. Durante este proceso de delegación, los sub-compuestos pueden ser filtrados por ciertos elementos (por ejemplo, evitar los otros elementos), pueden expandirse elementos (por ejemplo, mapear un elemento a muchos elementos como si esto fuera realizado por el *root iterator*) o la navegación del metamodelo puede ser realizada antes de delegar a los siguientes sub-compuestos.

Los **blueprints** son usados para encapsular y realizar la interpretación exacta del elemento del modelo fuente en el espacio objetivo en diferentes contextos. Por lo tanto, para cada "tipo" del elemento del modelo fuente (por ejemplo, para cada elemento del metamodelo) un **blueprint** es construido. Este **blueprint** actúa como un decorador, encapsulando el "conocimiento" de cómo representar un "tipo" del elemento del modelo en diferentes contextos.

Hay varios contextos de transformación en los que un **blueprint** puede ser requerido. Estos son expresados por las diferentes secciones del artefacto que tienen un **blueprint** asociado. Las figuras 9, 10 y 11 muestran los **blueprints** que el cartucho ejemplo define y como ellos son usados. Por su estructura básica solo hay dos "tipos" de los elementos del modelo que necesitan ser interpretados en el espacio objetivo – clases y atributos. Las figuras muestran los **blueprints** asociados con las secciones de artefacto apropiadas.

Cabe destacar la navegación del metamodelo expresada por el clave cualificadora en el registro del blueprint del atributo **BPAttribute**. Cuando delegamos a través de la estructura compuesta del cartucho, el sujeto actual de la generación es propagado. Esto es apropiado para el ejemplo, cuando generamos la declaración de la interfaz o de la clase. Sin embargo, cuando generamos para cada atributo de la clase, antes que delegar a los blueprint sub-compuestos, la clase necesita ser travesada para que el atributo de la clase sea propagado en vez de la clase en sí

misma. Esta navegación es modelada usando la clave cualificadora, la cual es una expresión de los elementos CARAT del lenguaje de expresión de mapeo y filtrado (Podemos ver más información sobre las reglas de mapeo y filtrado en la ayuda de la herramienta).

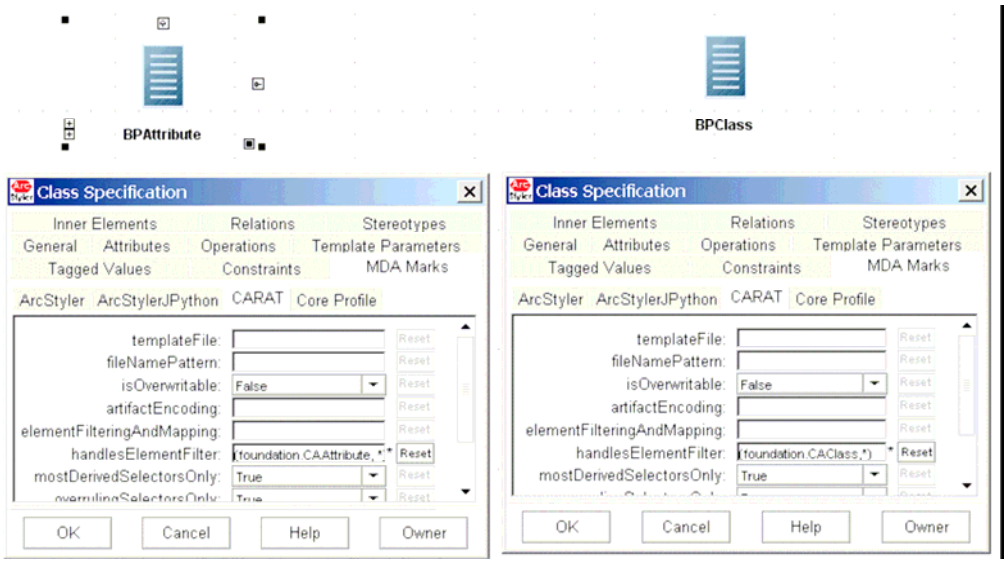

Figura 9. Blueprint de los elementos del modelo y filtro "handlesElement"

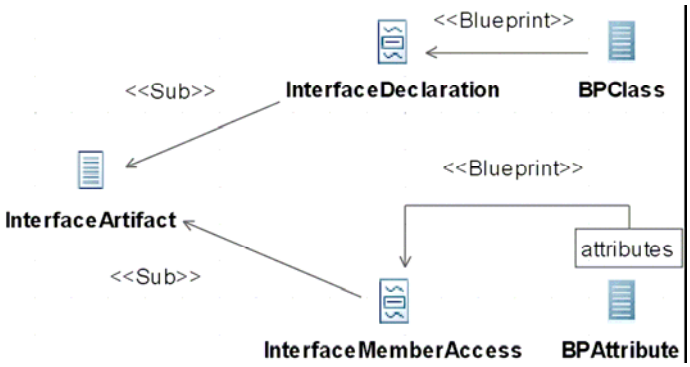

Figura 10. Añadiendo blueprints a las secciones del artefacto interfaz

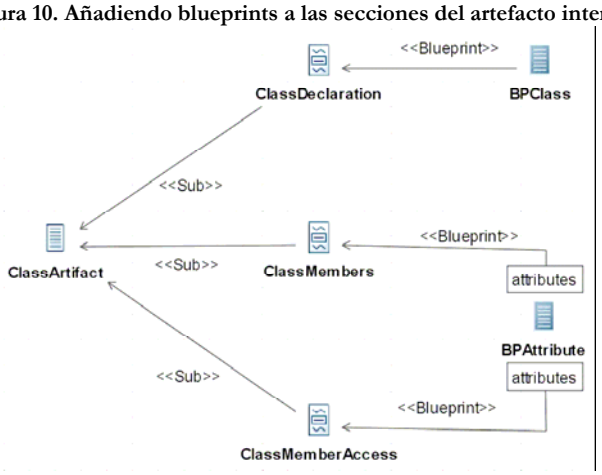

Figura 11. Añadiendo blueprints a las secciones del artefacto clase

Generando el cartucho MDA

Ahora el modelo es suficientemente completo para que a partir del cartucho MDA CARAT generar la implementación del cartucho MDA ejemplo por primera vez. Configurando el cartucho MDA CARAT como el que es usado para su generación y después seleccionado el paquete en más alto nivel del cartucho simple y generando el modelo tendremos como resultado la primera implementación del esqueleto Jpython para el cartucho MDA modelado. Esta implementación del esqueleto es un cartucho valido ya. Puede ser cargado en el motor MDA de Arcstyler y puede ser ejecutado- sin embargo, este cartucho todavía necesita dos refinamientos:

- Una implementación por defecto del fichero plantilla del artefacto ha sido generada. Estos ficheros necesitan ser adaptados de tal forma que ellos reflejen la estructura de un fichero de clases java o de interfaz java respectivamente.

- La "lógica del negocio" de la transformación necesita ser añadida. La implementación generada ya contiene "slots" donde la lógica del negocio puede ser insertada. En particular, esta es la lógica que trata con la interpretación de los elementos del modelo en diferentes contextos como el contexto de una declaración de una clase o de una interfaz.

Adaptando el Fichero Plantilla del Artefacto

El fichero **.tpl** para los artefactos **ClassArtifact** y **InterfaceArtifact** están entre los ficheros generados. Como un fichero plantilla, este contiene texto estático que es emitido al flujo de salida directamente y un número de invocaciones delegadas de acuerdo a la estructura del cartucho. Se puede editar el fichero plantilla seleccionando el artefacto en el modelo y seleccionado la opción del menú contextual (*MDA Cartridge Wizards - CARAT - Jump To Implementation*). Con esto pasaríamos a la perspectiva de Cartuchos y cargaríamos y mostraríamos el fichero plantilla para el artefacto seleccionado. La figura 12 muestra el fichero plantilla adaptado del artefacto interfaz.

Después de seleccionar el sujeto de la generación, este primero delega a la sección del artefacto llamada **InterfaceDeclaration** que ha sido registrada con este artefacto en el modelo del cartucho. Cabe destacar que el sujeto actual de la generación (*rootElement*) es pasado cuando se delega a sub-compuestos como una sección del artefacto. Esta sección del artefacto se supone que contribuye a la declaración de la interfaz. Después, un corchete abierto es escrito y el entorno del generador en tiempo de ejecución es requerido para incrementar el nivel de sangrado para un bien formateado *layout*. Posteriormente, el control es de nuevo dado a una de los sub-compuestos por la delegación a la sección del artefacto **InterfaceMemberAccess**. Después de que la llamada finalice, el nivel de sangrado es decrecido y el corchete de cierre es escrito. La plantilla del artefacto de clase ha sido adaptada de forma similar, ver la figura 13.

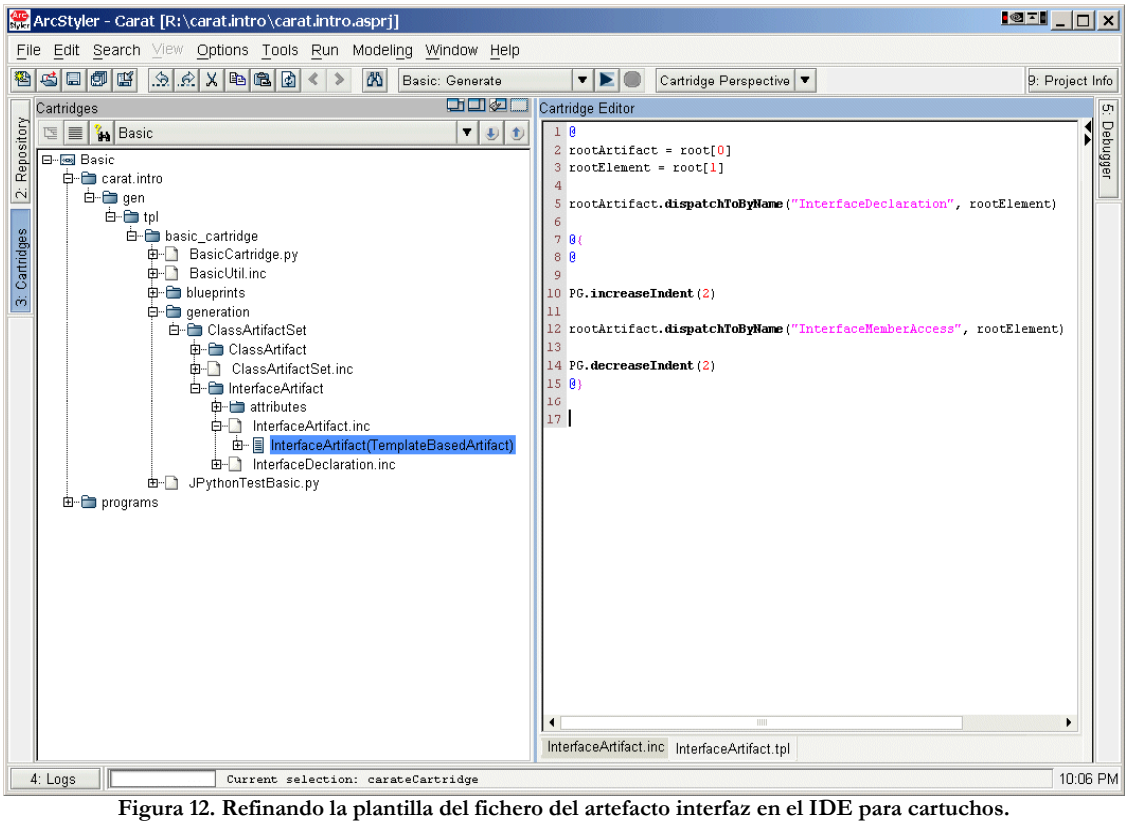

Figura 12. Refinando la plantilla del fichero del artefacto interfaz en el IDE para cartuchos.

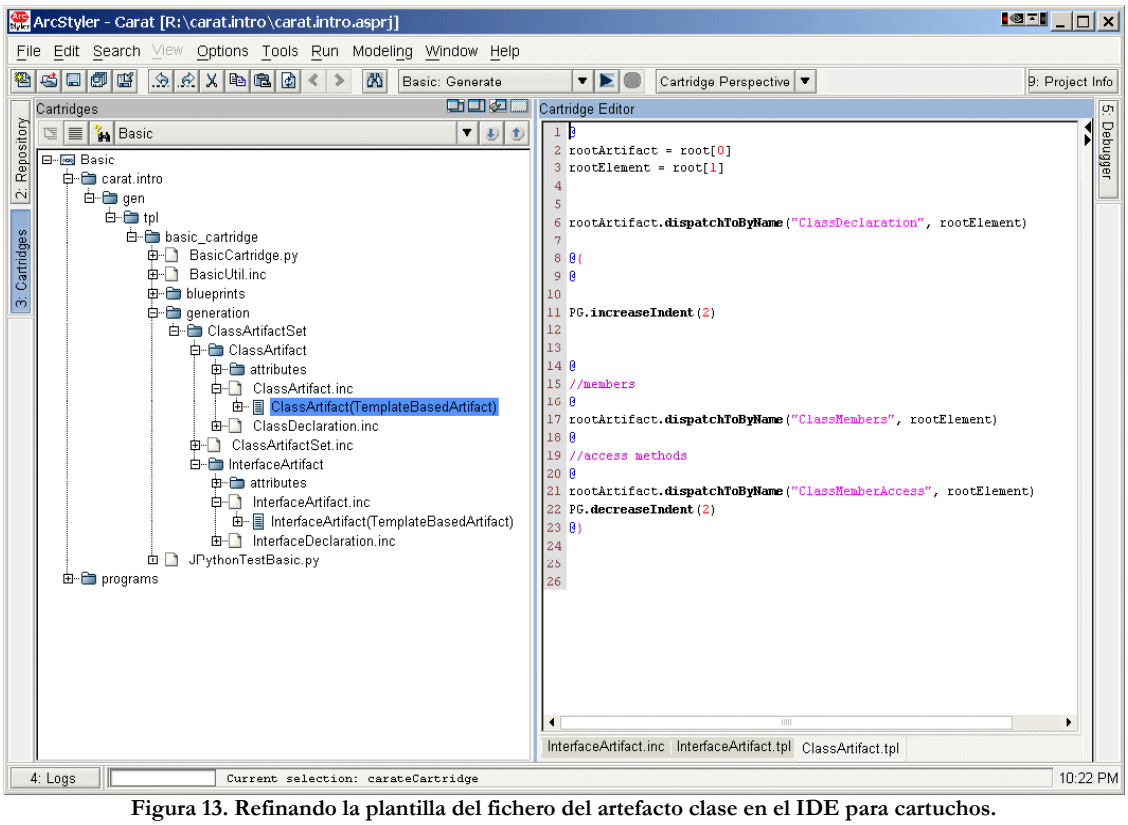

Figura 13. Refinando la plantilla del fichero del artefacto clase en el IDE para cartuchos.

Implementando las Contribuciones Especificas del Contexto del Blueprint

La siguiente tarea es la definición de cómo los elementos del modelo fuente deben ser interpretados en los diferentes contextos generados (por ejemplo, en las diferentes secciones del artefacto). Las operaciones generateFor...() son generadas para todos los blueprints para estos contextos en el cual éstos están registrados. Se puede navegar de las contribuciones a una sección del artefacto seleccionando esta en el modelo de cartucho y eligiendo la opción del menú contextual *MDA Cartridge Wizards - CARAT - Jump To Contributors*. En el caso de múltiples blueprints contribuyentes, estos serán listados en una caja diálogo y se podrá seleccionar entre ellos.

Cuando nos fijamos en la operación *generateForXYZ* (XYZ denota el nombre de la sección del artefacto que proporciona el contexto de interpretación) como la mostrada en la figura 14, hay un espacio de código generado que está entre la declaración del método y el comienzo del área protegida. Siempre es deseable que los usuarios pueden cambiar el código generado manualmente, por ello el área *protected area* esta creada con este propósito. Los usuarios pueden reconocer un área protegida por los comentarios que la rodean. El generador puede reconocer y proteger cambios manuales. La implementación por defecto del método *generateFor* consiste de un comentario "*Implement your contribution here*" que necesita ser refinado de forma manual. En el caso de este ejemplo, el método ya ha sido refinado y delega a otro método que puede ser usado desde diferentes lugares y por lo tanto ha sido factorizado. Este método ha sido modelado en el **blueprint** de la clase **BPClass** explícitamente y es implementado como se muestra en la figura 15.

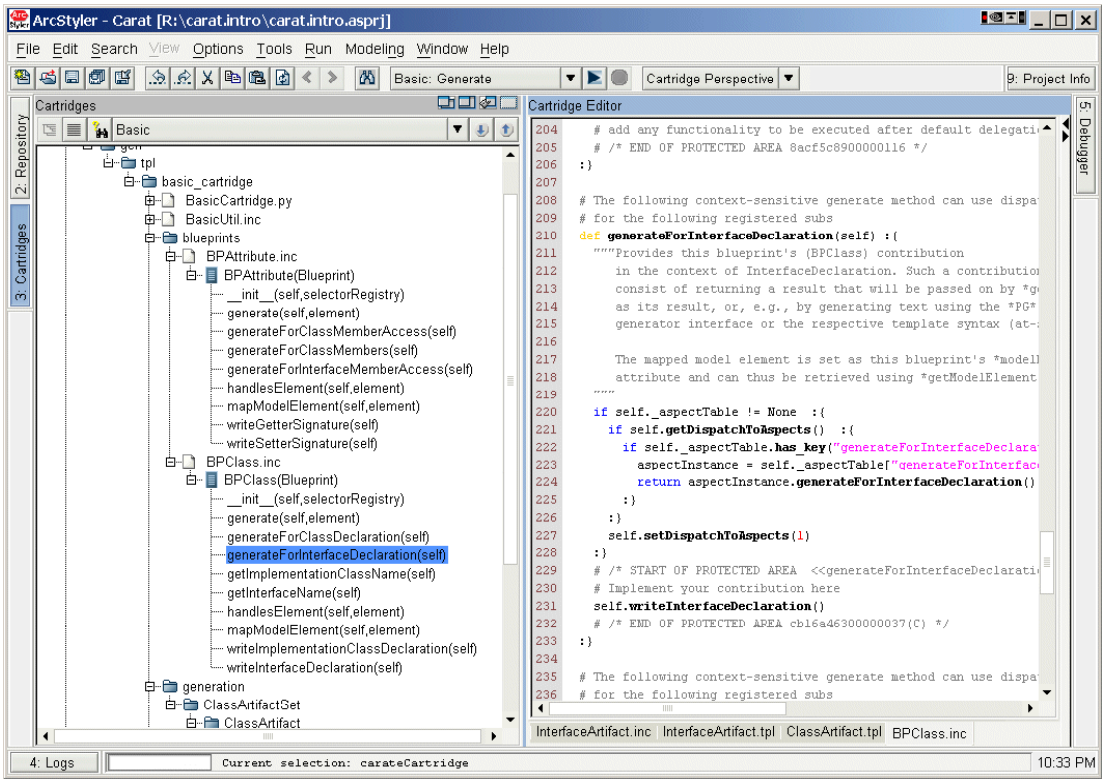

Figura 14. Refinando las implementaciones del método del blueprint Clase en el IDE de cartuchos

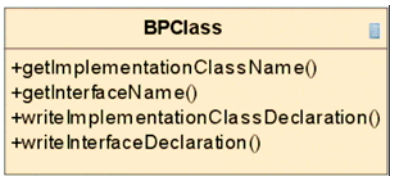

Figura 15. Factorizando la lógica a través de métodos.

La figura 16 muestra la implementación de writeInterfaceDeclaration() la cual es de nuevo una mezcla de texto estático y computación dinámica.

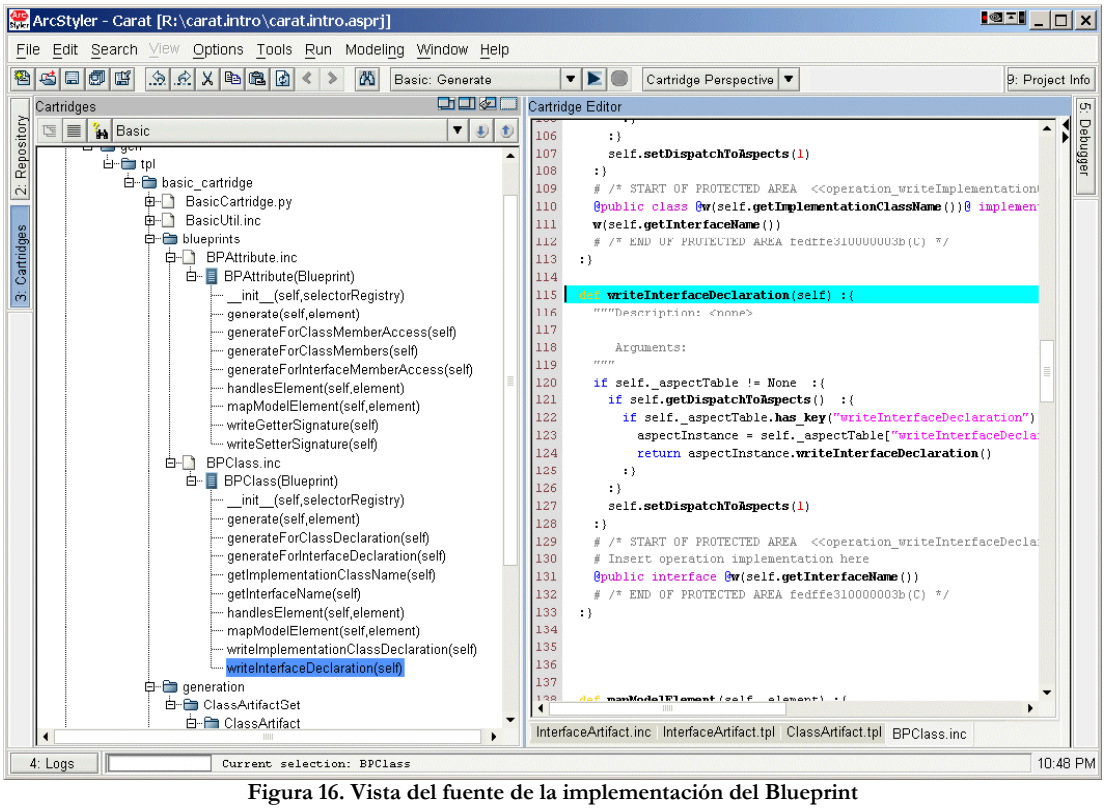

Figura 16. Vista del fuente de la implementación del Blueprint

Finalmente, la implementación de getInterfaceName() accede al elemento del modelo el cual es "decorado" por el brueprint y devuelve su nombre.

```
def getInterfaceName(self) :{
# /* START OF PROTECTED AREA ...*/
# Insert operation implementation here
return self.getModelElement().getName()
# /* END OF PROTECTED AREA fedffe310000003b(C) */
```

Ahora, el ejemplo del cartucho esta completado. Un modelo de test pues ser creado en un paquete separado en el modelo UML, el cartucho puede ser cargado en ArcStyler y su característica **generar** puede ser ejecutada.

3.5 Extendiendo el Cartucho MDA

Los conceptos de extensión de un cartucho CARAT son similares a extender un framework orientado a objetos: el framework será extendido sin ser cambiado internamente y por lo tanto respetando las aplicaciones existentes. Para ello, el lenguaje de programación orientado a objetos proporciona mecanismos de extensión como herencia y polimorfismo y el framework define puntos de extensión que permiten la extensión del framework de forma anticipada. Para los cartuchos CARAT de ArcStyler, la arquitectura CARAT jugaría el rol del lenguaje de programación orientado a objetos y proporcionaría los mecanismos de extensión. El cartucho jugaría el rol del framework orientado a objetos y proporcionará los puntos de extensión.

A groso modo, podemos decir que hay 3 formas para extender un cartucho:

- Añadiendo artefactos para que sean generados. En este ejemplo, este podría ser extender el cartucho ejemplo desarrollado para tener la capacidad de generar un fichero constructor ANT que se encargaría de compilar y empaquetar las clases Java generadas fácilmente.
- Cambiar la forma en que los artefactos son generados. En este ejemplo, este podría ser que todas las *interfaces* (de nuestro modelo objetivo en el cartucho ejemplo) podrían requerir ser extendidas de *java.io.Serializable*.
- Suprimir la generación de algún artefacto.

Debido a la estructura compuesta de un cartucho existen algunos mecanismos de extensión naturales proporcionados por la arquitectura CARAT que permite puntos de extensión naturales en el cartucho generado. Por ejemplo, el **root iterator** y el **conjunto de artefactos** son puntos de extensión natural ya que pueden fácilmente ser añadidos conjuntos de artefactos o simples artefactos en un sub-cartucho. CARAT **SelectorRegistries** es otro concepto que proporciona la capacidad de sobrescribir un blueprint de un cartucho con una redefinición.

La siguiente sección muestra como extender el cartucho ejemplo por un artefacto para el fichero constructor ANT y como cambiar la generación de comportamiento tal que todas las *interfaces* extiendan de *java.io.Serializable*.

3.5.1 Sobrescribiendo el Comportamiento de un Super-Cartucho

Basándonos en el cartucho ejemplo, vamos a implementar un cartucho derivado, el cual genera las declaraciones de las interfaces Java de tal forma que extiendan de *java.io.Serializable*. Como primer paso, un nuevo cartucho tiene que ser creado tal que derive del cartucho ejemplo desarrollado. El enfoque es similar que para el cartucho ejemplo. Un paquete es creado en el modelo UML y un cartucho es situado allí. En vez de heredar del cartucho CARAT **foundation** este extenderá el cartucho ejemplo (figura 17).

Cuando generamos el nuevo cartucho usando el cartucho CARAT, un cartucho es generado, que es totalmente funcional y se comporta exactamente igual que el cartucho padre. La clave para cambiar el comportamiento es la clase Blueprint **BPClass.** En este, el método *writeInterfaceDeclaration ()* ha sido implementado. Como heredado del cartucho ejemplo este método genera la sentencia *interface* y el nombre de la interfaz. La implementación tiene que ser sobrescrita con objetivo de generar la cláusula *extends* deseada. Por lo tanto, el blueprint derivado es modelado de tal forma que *writeInterfaceDeclaration()* es sobrescrito (figura 18).

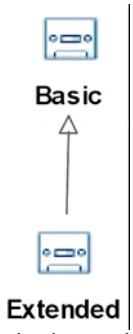

Figure 17. Extendiendo el cartucho ejemplo

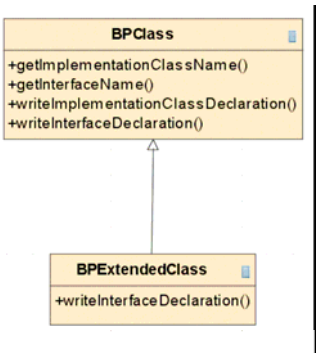

Figura 18. Extendiendo el blueprint sobrescribiendo los métodos heredados

La implementación del método sobrescrito se muestra en la figura 19.

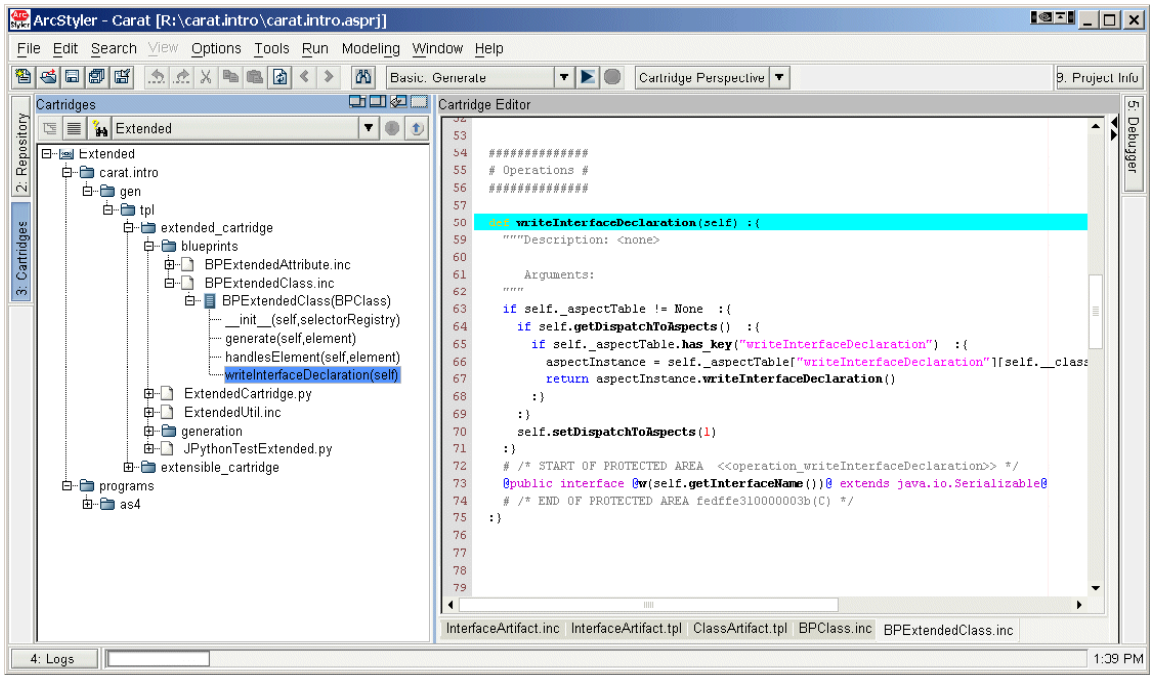

Figura 19. Implementación del método sobrescrito.

El único aspecto importante que queda ahora es como asociar el blueprint derivado con el cartucho de tal forma que en tiempo de ejecución este sea usado en vez de la implementación base. Puesto que al crear el cartucho ejemplo no se tuvo en cuenta la extensibilidad, hay una falta de puntos de extensión. La figura 20 muestra, como el cartucho base necesita ser modificado con objeto de proporcionar el deseado punto de extensión. En vez de registrar la sección del artefacto **InterfaceDeclaration** con el blueprint de la clase **BPClass** directamente, el registro es realizado indirectamente vía el llamado **SelectorRegistry.** En tiempo de ejecución, cuando generemos desde la sección del artefacto **InterfaceDeclaration**, antes de delegar al blueprint de la clase, el más específico de los blueprint es seleccionado por el **SelectorRegistry.** Para el cartucho base, este será siempre **BPClass.** Cuando modelamos el sub-cartucho, el blueprint extendido de la clase **BPExtendedClass** es registrado con el selector de registro proporcionado por el cartucho base. Esta asociación no tiene impacto en el cartucho base pero causa que **BPExtendedClass** sea devuelto por el registro selector en

tiempo de ejecución en vez de **BPClass.** Por lo tanto será una buena practica siempre asociar blueprints con las secciones de los artefactos vía registros selectores y por lo tanto permitiendo la reutilización de cartuchos.

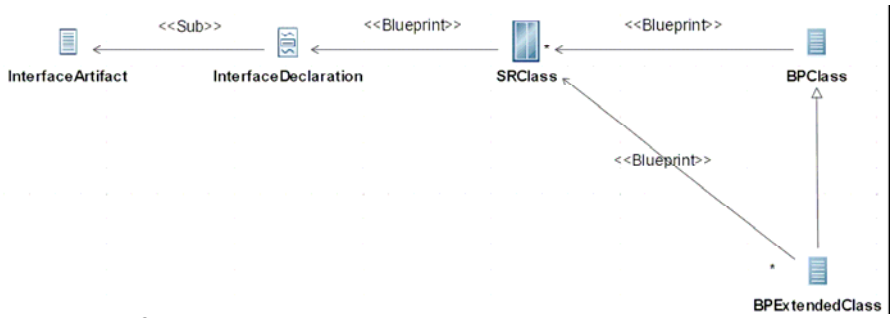

Figura 20. Sobrescribiendo el comportamiento de un blueprint en un sub-cartucho

Ahora, el cartucho extendido puede ser generado usando el cartucho CARAT y cuando se ejecute, este producirá la declaración de interfaz modificada que ahora incluirá la cláusula *extends*.

3.5.2 Añadiendo un Artefacto

Como otra extensión del cartucho ejemplo, el sub-cartucho podría generar un artefacto adicional que soportara la compilación y empaquetado de los artefactos clase e interfaz generados. Aquí, ANT es utilizado y un fichero constructor apropiado es generado. En el modelo, debería ser posible agrupar las clases en unidades de despliegue asignándolas a componentes físicos. Cuando generamos el componente físico, un fichero constructor ANT es generado que principalmente consiste de texto estático, excepto la lista de ficheros java. la cual esta basada en los residentes asignados a un componente físico particular.

La figura 21 muestra como el conjunto artefacto adicional es añadido al sub-cartucho registrando este con el *root iterator* del cartucho base. Desde aquí, el procedimiento es ya familiar: una estructura compuesta es creada consistente de un conjunto de artefactos, un artefacto el cual esta basado en una plantilla (figura 22) y una sección de artefacto simple. Un fichero constructor es generado para cada componente físico en el modelo, tal que el **BuildScripts handlesElementFilter** es puesto a (foundation.CAComponent,*). Cabe destacar la navegación del metamodelo antes de delegar al blueprint de la clase **BPClass.** Además, cuando delegamos a través de la jerarquía compuesta, el componente físico tiene que ser pasado como el sujeto de la generación. Es más, para la lista de ficheros que necesita ser compilada, una entrada por residente es generada. La navegación es modelada, por lo tanto, usando la clave cualificadora *residents*.

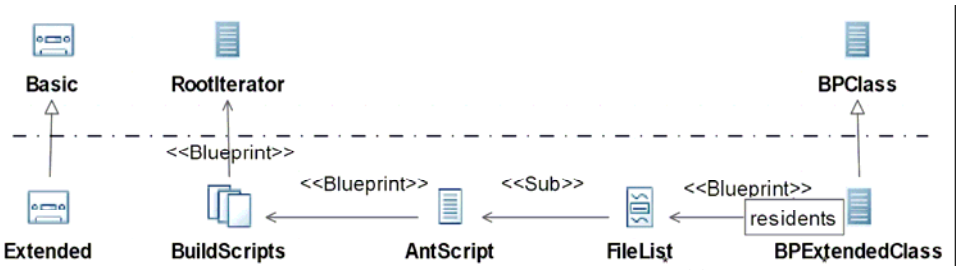

Figura 21. Añadiendo un artefacto de fichero constructor ANT al sub-cartucho

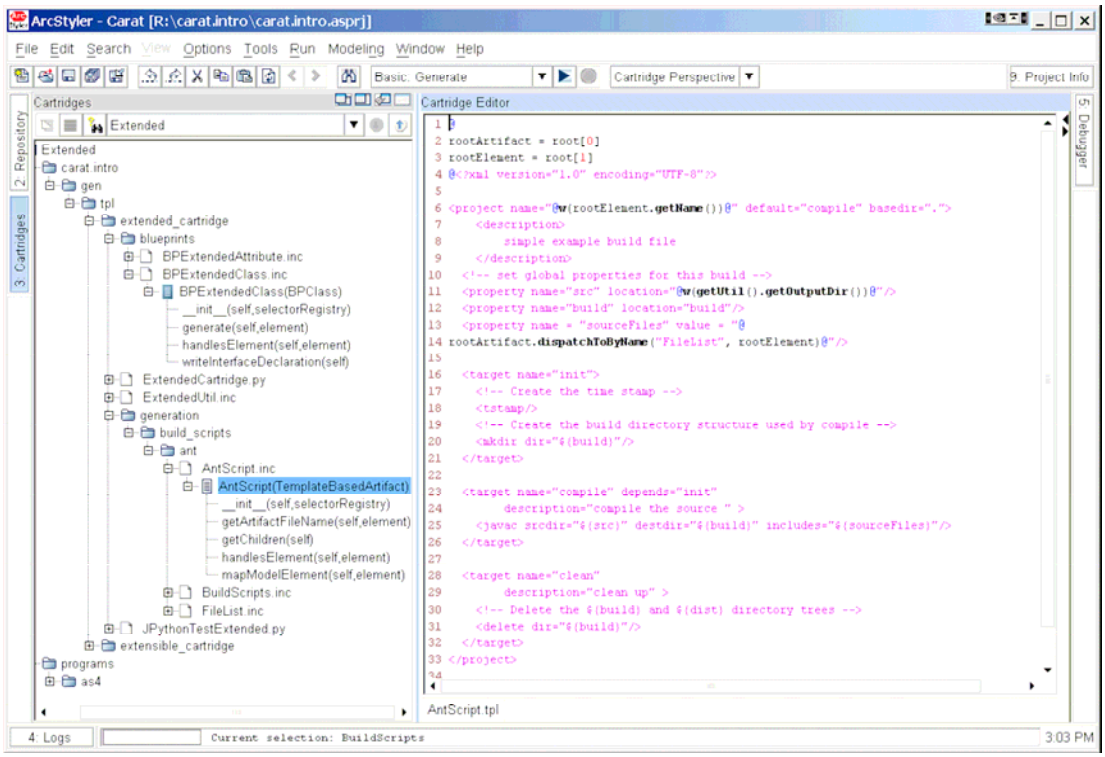

Figura 22. Plantilla del fichero constructor ANT

La implementación de cómo las clases son interpretadas como las entradas de la lista del fichero ANT está localizado en el método generado *generateForFileList()* y ha sido refinado manualmente como se muestra en la figura 23.

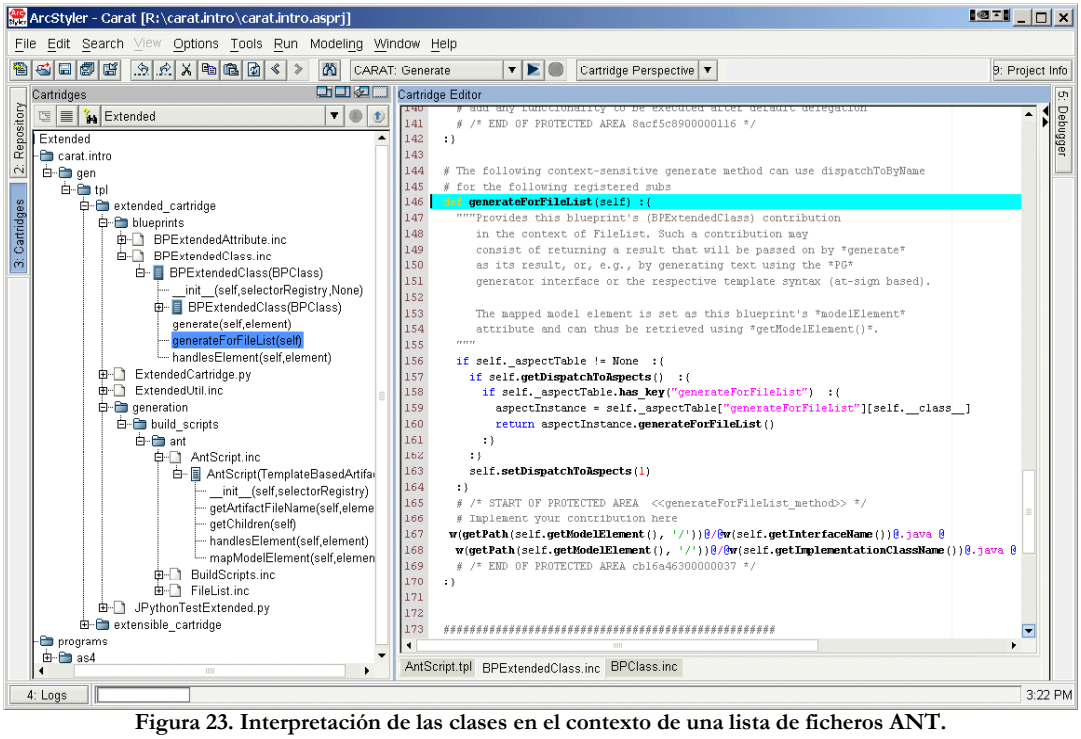

Figura 23. Interpretación de las clases en el contexto de una lista de ficheros ANT.

3.5.3 Testeando el Cartucho MDA

Finalmente, un paquete de test puede ser incluido en el modelo del cartucho MDA. Este puede ser usado para aplicar el cartucho MDA a algunos elementos del modelo para testear si la generación trabaja adecuadamente. Es razonable poner el flag *Ignore* del paquete de más alto nivel en la hoja de propiedades de la parte de test del modelo a *True* con objeto de evitar la generación accidental del modelo de test con el cartucho MDA CARAT.

4 Otros Aspectos de CARAT

4.1 Las marcas (marks)

Un cartucho lee un modelo de entrada y lo mapea a varias salidas. Uno de los beneficios más importantes de MDA es que el modelado puede ser portable para diferentes plataformas objetivos. Sin embargo, a veces el modelo portable de entrada no contiene toda la información requerida para tomar todas las decisiones para mapear a una plataforma objetivo especifica. Esta información puede ser adjuntada al modelo usando las marcas. En la figura 24 podemos ver un ejemplo de un conjunto de marcas usadas para el cartucho Java2 en una clase.

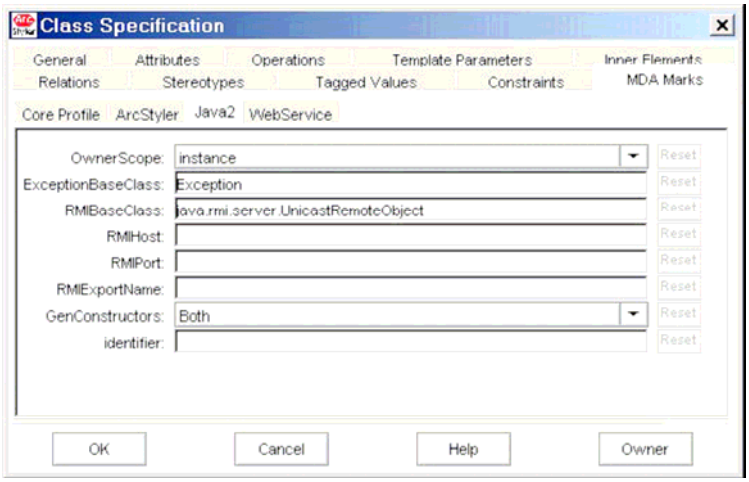

Figura 24. Marcas Java2 en una clase

Otra forma de considerar las marcas es usándolas para mantener valores de meta-atributos definidos por el usuario. Dicho de otra forma, si el actual metamodelo (por ejemplo, UML) no es suficiente para mantener todos los atributos que querría tener disponibles asignados a los elementos del modelo, se pueden usar las marcas para guardar esta información adicional. Es conveniente proporcionar métodos de acceso en los blueprints de los cartuchos para encapsular la obtención de los valores de las marcas.

En un cartucho queda definido que marcas son esperadas según el elemento del modelo. Esto es hecho en el modelo del cartucho, donde la definición del conjunto de marcas es especificada usando UML. Un cartucho puede usar cualquier número de definiciones de conjuntos de marcas, cada uno de los cuales es representado por una clase con estereotipo **MarkSetDefinition** que esta conectado al cartucho por una asociación con estereotipo

MarkSetDefinition. La figura 25 muestra el modelo del cartucho Java2 y como este declara dos definiciones de conjunto de marcas.

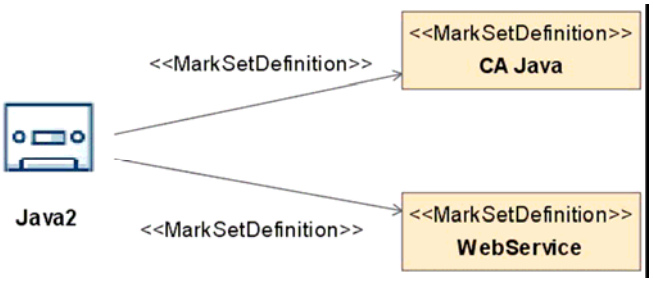

Figura 25. Declarando las definiciones del conjunto de marcas para un cartucho

La figura 26 muestra un extracto de las definiciones del conjunto de marcas del modelo para el cartucho Java2 que se corresponden con las de la figura 24. Se puede ver que la definición del conjunto de marcas es modelada usando una clase con estereotipo **MarkSetDefinition** (clase "CA Java" en la figura). Éste tiene una asociación directa a la clase "**TaggedValueSet**" ("default" en la figura) la cual es usada para propósitos de agrupamiento.

La clase de conjunto de valores etiquetados referencia a una o más clases con el estereotipo **MetamodelElementSpecificTaggedValueSet.** Como se muestra en la figura 26, se pueden usar los atributos de estas clases para definir la marcas que quieres tener disponibles para editar en un diálogo como en la figura 24 y acceder por lo tanto a las marcas dentro del cartucho

Las marcas son mucho más que valores etiquetados. Se puede especificar un nombre, un tipo y un valor por defecto. Los tipos pueden ser:
• StringMark
• IntegerMark
• BooleanMark
• MultilineStringMark
• Un tipo definido por el usuario: *enumeration*.

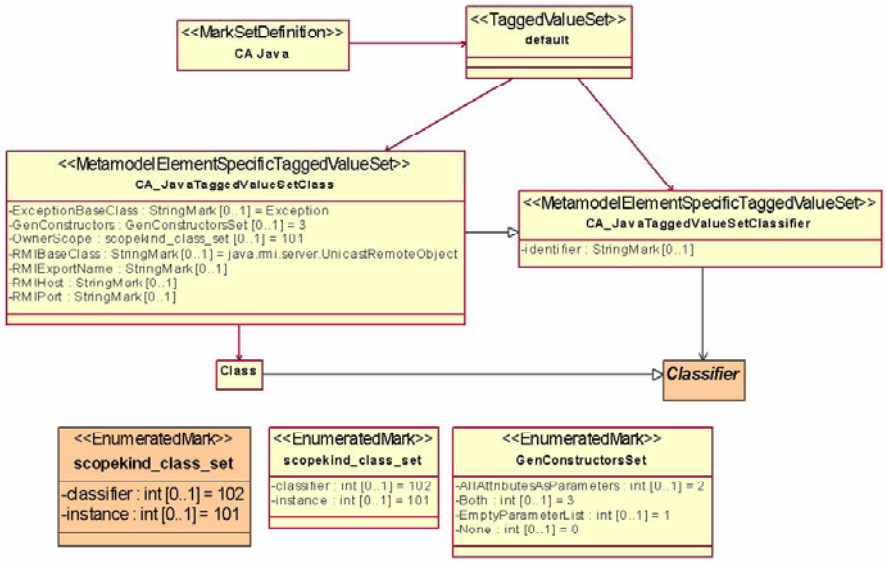

Figura 26. Extracto de la definición del conjunto de marcas en el modelo java2.

La clase de definición de conjunto de marcas agrupa definiciones de marcas. Cada conjunto se mostrará al usuario como una hoja de propiedades en el editor de marcas de ArcStyler. La figura 27 muestra la correspondencia entre la definición del conjunto de marcas y como estas son mostradas en el editor de marcas.

Se tiene que declarar a que tipo de los elementos del modelo las marcas se aplican. En el ejemplo mostrado la *CA_JavaTaggedValueSetClass* se aplica a los elementos del modelo de tipo Clase, expresado esto por la asociación desde la clase de definición del conjunto de marcas al metatipo **Class.** Esta relación podría también ser configurada para considerar solo elementos que tengan un específico estereotipo.

Las marcas pueden heredar de otros conjuntos de marcas. El conjunto de marcas derivado hereda todos los valores definidos. La figura 26 muestra esto para el ejemplo del *classifier* y las clases. La marca **identifier** es definida para el *classfier*. Puesto que clases es una clase especial de *classfier*, este hereda la definición de marcas. Por lo tanto, la definición del conjunto de marcas para las clases (**CA_JavaTaggedValueSetClass**) es modelado de tal forma que hereda de la definición del conjunto de marcas del *classfier* (**CA_JavaTaggedValueSetClassifier).**

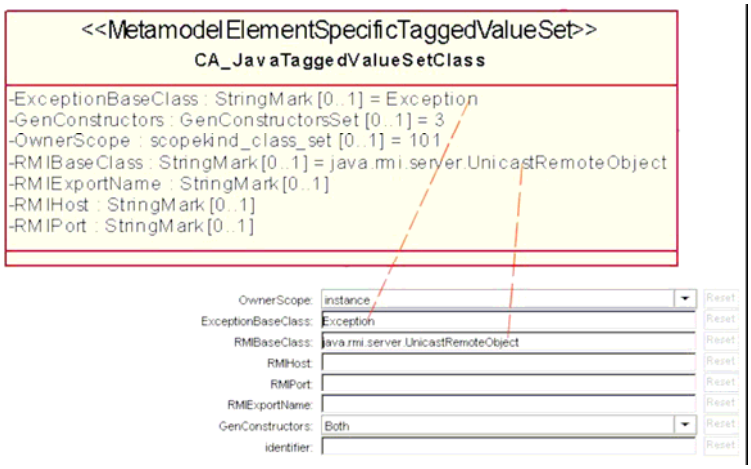

Figura 27. Como las definiciones del conjunto de marcas son mostradas en el Editor de marcas

Accediendo a las Marcas en un Cartucho

Un cartucho puede leer y escribir marcas usando las dos funciones JPhython **getTV** y **setTV**. Asumiendo que la variable **c** se corresponde con el elemento del modelo de tipo **Class**. Usando las definiciones del conjunto de marcas de la figura 26, el código fuente del cartucho que obtiene el valor de la marca **identifier** de la clase asignada a **c** sería como sigue:

```
identifierMarkValue = getTV(c, "identifier", "CA Java")
```

El valor "CA Java" se corresponde con el nombre de la clase **<<MarkSetDefinition>>** a la cual la marca pertenece. Si el usuario no ha asignado un valor a la marca explícitamente, el valor por defecto será devuelto. Hay que tener en cuenta que las marcas siempre devuelven los valores como *strings*. Para poner el valor de la marca, por ejemplo a "NewIdentifier" sería como sigue:

```
setTV(c, "NewIdentifier", "identifier", "CA Java")
```

4.2 Tipos de Datos

Normalmente el modelo de entrada de una transformación puede necesitar referenciar a tipos desde el entorno objetivo que el usuario no tendría que crear por si mismo una y otra vez para cada modelo que ellos quisieran utilizar en el cartucho. Por ejemplo, cuando usamos el cartucho Java2, los tipos más usados por los usuarios desde el Java Development Kit deberían estar disponibles de tal forma que estos pudieran ser utilizadazos por ejemplo como atributo o tipos de parámetro.

Por ello, un cartucho puede especificar un número de paquetes como paquetes especiales de *tipos de datos*. Cuando generamos la implementación del cartucho usando el cartucho CARAT, un fichero XMI será producido para cada uno de estos paquetes, manteniendo el contendido de los paquetes. Estos ficheros XMI automáticamente serán cargados en el modelo que usa el cartucho, de tal forma que dejara disponibles los tipos de datos para modelar.

La figura 28 muestra, como ejemplo, los paquetes de tipos de datos que el cartucho Java2 define. Estos son paquetes que contienen los tipos frecuentemente usados por los usuarios de Java Development Kit (paquete **java2**) y de los tipos de Java2 Mobile Edition (paquete **j2me**). Los paquetes de tipos de datos son referenciados por el cartucho usando la dependencia con el estereotipo **datatypes**.

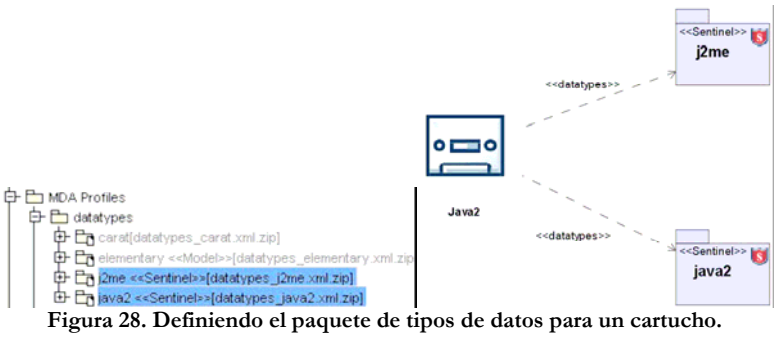

Figura 28. Definiendo el paquete de tipos de datos para un cartucho.

4.3 Tareas Típicas

Por otra parte en la ayuda que proporciona la herramienta, tras la experiencia acumulada con otros desarrollos de cartuchos MDA, se nos es presentado un conjunto de típicas tareas y patrones que han sido identificados en el uso del estilo de modelado CARAT y para el cual han implementado algunos *wizards* (tareas típicas, a parte de las ya comentadas en la sección de *Extendiendo el cartucho MDA*). Algunas de ellas, de las cuales se pueden ver más detalles en la ayuda, son:

- Crear un artefacto para generar automáticamente un fichero existente utilizando ArcStyler.

- Preservar los cambios manuales en la salida, evitando que si es modificado el cartucho el texto ya manualmente insertado sea sobrescrito.

- Evitar la generación de artefactos y conjuntos de artefactos heredados.

- Añadir propiedades a los cartuchos, es decir información adicional de configuración.

- Crear nuevas *Features* para un cartucho, *features* que son heredadas de los cartuchos base, en particular la característica de **generar** que es heredada del cartucho **foundation.**

- Patrón "Collector" el cual nos sirve para diferenciar cual de las contribuciones, es decir blueprints asociados a una sección del artefacto, debe ser elegida.

- Elemento del modelo "wrapping", relacionado con una mejora para expresar el mapeo y filtrado entre asociaciones mediante claves cualificadas de Composiciones CARAT.

5 Conclusiones

En este trabajo hemos pretendido dar una visión de cómo se debe trabajar para conseguir desarrollar un cartucho MDA para que sea incluido en la herramienta ArcStyler y con ello conseguir la especificación de una transformación MDA entre modelos. Con el manejo de la herramienta nos hemos dado cuenta que utilizándola (y sobre todo usando los cartuchos que ya tiene definidos para distintos entornos de trabajo) se puede consigue llegar desde un modelo a la generación de código, haciendo una serie de modificaciones, pero en definitiva cumpliendo lo expresado en la metodología MDA. Sin embargo hemos de decir que tras el trabajo realizado consideramos que la tarea de desarrollar un cartucho para una nueva tecnología más o menos compleja puede resultar bastante compleja, puesto que uno se enfrenta a un entorno de desarrollo relativamente nuevo y un lenguaje de programación no muy extendido como es Jphyton.

También debemos añadir que este trabajo es sólo un leve acercamiento a la arquitectura CARAT de ArcStyler y que en la documentación de la herramienta podemos encontrar en mucho más detalles toda la descripción de la arquitectura CARAT de ArcStyler.

6 Bibliografía

Interactive Objects, ArcStyler 4.0.90.2004. http://www.arcstyler.com/

Ingeniería de Modelos con MDA. Estudio comparativo de Optimal J y ArcStyler. Jesús Rodríguez Vicente. Tutor: Jesús Joaquín García Molina. Proyecto Informático Junio 2004.

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