FACULTAD DE INFORMATICA
UNLP

TESINA DE LICENCIATURA

TITULO: Análisis y uso de los frameworks de Eclipse para la definición de DSLs

AUTORES: Luciano Omar Andino - Germán Esteban Ruiz

DIRECTOR: Lic. Luis Mariano Bibbó

CODIRECTOR: Dra. Claudia Pons

CARRERA: Licenciatura en Informática

Resumen

El propósito del Modelado Específico de Dominio (DSM) es crear modelos para un dominio utilizando un lenguaje enfocado y especializado conocido como Lenguaje Específico de Dominio (DSL). Incluye la idea de generación automática de código ejecutable directamente desde los modelos, para que las aplicaciones finales puedan ser generadas a partir de estas especificaciones en alto nivel. Así, se libera al desarrollador de las tareas de codificación y mantenimiento del código fuente y se incrementa significativamente su productividad. En este trabajo, se hizo un estudio detallado de la metodología DSM y su comparación con otras técnicas existentes. Además se incluyó un análisis de las herramientas para la construcción de DSLs, que existen dentro de la plataforma Eclipse. Para finalizar el trabajo se presentó un caso de estudio donde se definió e implementó un DLS para ambientes colaborativos que permitió integrar los conceptos tratados.

Líneas de Investigación

Las siguientes son las líneas de investigación en las cuales se encuadra el presente trabajo:

-Ingeniería de Software
-Modelado Específico de Dominio
-Lenguajes Específicos de Dominio
-Entorno de desarrollo Eclipse
-Sistemas colaborativos

Conclusiones

Como conclusión podemos resaltar los beneficios existentes al contar con DLS para la resolución de problemas en dominios acotados. Generalmente se la considera una tarea ardua, sobre todo al considerar el costo de su implementación sobre el beneficio que se puede obtener a partir de su uso.
Gracias las herramientas que existen en la actualidad y en particular dentro del ambiente Eclipse, la creación de este tipo de lenguajes se convirtió en una tarea más sencilla.
Aunque la mayor complejidad sigue focalizada en el proceso de identificar los objetos del dominio, por sobre la implementación concreta del lenguaje.

Trabajos Realizados

Como aportes de este trabajo se puede mencionar:

-Se presentó el proceso de creación de un DSL, detallando los pasos necesarios para su definición.
-Se enunciaron los beneficios de definir y utilizar un DLS en lugar de utilizar otras técnicas de desarrollo.
-Se expusieron diferentes plugins de Eclipse para implementar DSLs. En cada caso se indicó su objetivo, su forma de uso y las ventajas y desventajas que poseen.
-Por último, se implementó un DSL para modelar sistemas colaborativos donde se integraron todos los conceptos vistos en este trabajo.

Trabajos Futuros

Como trabajos futuros se plantean las siguientes líneas:

-Implementar una herramienta integradora que simplifique la creación de un DSL.
-Extender la implementación del DSL para ambientes colaborativos agregándole una fase final en la que se genere el esqueleto de una posible implementación.
-Implementar un plugin que facilite la construcción de figuras para editores generados con GMF.
-Investigar la forma de solucionar el problema que se presenta debido a la evolución de los metamodelos y su consecuente desincronización respecto de los modelos ya generados.

Fecha de la presentación: Agosto de 2009

Agradecimientos

Agradezco a mi madre, quien me dio la posibilidad de estudiar, y me acompañó con su apoyo durante toda la carrera. A mis compañeros de la facultad, Alejandra, Arturo, Marcos, y especialmente Germán y David; con quienes no solo compartí muchas horas de estudio, sino que hoy son dos de mis más grandes amigos. A Gabriela por su apoyo y porque fue clave para que pudiéramos perseverar hasta concluir este trabajo. A mis compañeros de trabajo por escucharme diariamente hablando sobre esta tesina. A todos los amigos con quienes la vida me ha premiado, por su apoyo, palabras de aliento y buenos deseos. A nuestro director Luis Mariano Bibbó y a nuestra Codirectora Claudia Pons, por todo su aporte y orientación.

Luciano

A Gaby, mi mayor agradecimiento. Con este trabajo concluyo un largo período de incertidumbre en cuanto a la terminación de mi carrera. Y sin su gran apoyo y constantes empujoncitos, nunca hubiera podido hacerlo.
A mis papás Alicia y Arnaldo, por darme las posibilidad de estudiar en una ciudad muy lejana.
A mis hermanos, Cecilia, Mariano y Federico, que me aguantaron durante la época de estudios.
A mis compañeros de la facu, Luciano, David y Arturo, por la amistad brindada.
A nuestros directores, Claudia y Luisma, por darnos las guías que nos permitió hacer este trabajo.
A Lautaro, por ser lo más preciado de mi vida.

Germán

Índice

Índice de Figuras

1 Introducción

El modelado específico de dominio DSM (Domain-Specific Modeling en inglés) [16],[17],[18] es una metodología de la ingeniería de software cuyo propósito es crear modelos para un dominio, utilizando un lenguaje enfocado y especializado para el mismo. El modelado específico también incluye la idea de generación automática de código y la creación de código ejecutable directamente desde los modelos. De esta manera, las aplicaciones finales serán luego generadas a partir de estas especificaciones en alto nivel. Esta automatización es posible si ambos, el lenguaje y los generadores, se ajustan a los requisitos de un único dominio. Al liberar al desarrollador de las tareas de codificación y mantenimiento del código fuente se incrementa significativamente su productividad. Además, como el código no se escribe manualmente, se reducen los defectos en las aplicaciones resultantes y se mejora la calidad de estos productos. Estos lenguajes se denominan DSLs (por su nombre en inglés: Domain-Specific Language) [3],[4],[9] y permiten especificar la solución usando directamente conceptos del dominio del problema. Estos DSLs proporcionan soluciones expresadas en el mismo lenguaje y con el mismo nivel de abstracción del problema. De esta manera, los expertos del dominio pueden entender, validar, modificar y eventualmente desarrollar programas en este lenguaje.

DSM persigue un incremento en la abstracción, más allá de lo provisto por los lenguajes de programación y técnicas de modelado genéricas actuales. Utiliza conceptos del dominio del problema para describir la solución. Además, surge como un medio para la desconexión entre las etapas de modelado e implementación de las metodologías existentes.

En las metodologías convencionales, durante la etapa de modelado, se genera un conjunto de modelos para abstraer aspectos de implementación que son complejos, o muchas veces desconocidos, en los momentos tempranos del desarrollo. Luego, en la etapa de implementación, los modelos son interpretados por los programadores, quienes manualmente convierten las descripciones abstractas en implementaciones concretas. Esta traducción manual e informal, es propensa a errores frecuentes en el desarrollo, los cuales resultan en una disminución de la calidad del software y en la productividad alcanzada durante su construcción.

Dado que el costo de mantener los modelos actualizados es mayor que el beneficio inicial obtenido por los mismos, usualmente no son actualizados al cambiar aspectos en la implementación, lo cual los torna inservibles como documentación. Rara vez es posible mantener actualizados los modelos y la implementación automáticamente en su totalidad. El grado en que esto puede llevarse a cabo depende de cuán parecidos sean entre sí los modelos y sus implementaciones.

En la Figura 1-1 se describen esquemáticamente diferentes maneras de relacionar el modelo con el código fuente y la aplicación final, incluyendo la metodología DSM. Las alternativas más usuales en la actualidad son:

(a) No incluir ningún modelo, lo cual es razonable para software de tamaño reducido.

(b) Se incluyen modelos que luego son descartados, debido a que el costo de mantenerlos actualizados es mayor a los beneficios que aportan.

(c) Se utilizan modelos para visualizar el código mediante alguna técnica de ingeniería inversa, lo cual puede facilitar su comprensión, pero no agrega semántica alguna al mismo.

(d) Ida y vuelta entre modelos y código, intentando actualizar los cambios producidos en cualquiera de los dos en su contraparte. Esto es posible de manera automática sólo cuando los formatos son estructuralmente parecidos o cuando se actualizarán los aspectos comunes entre ambas partes.

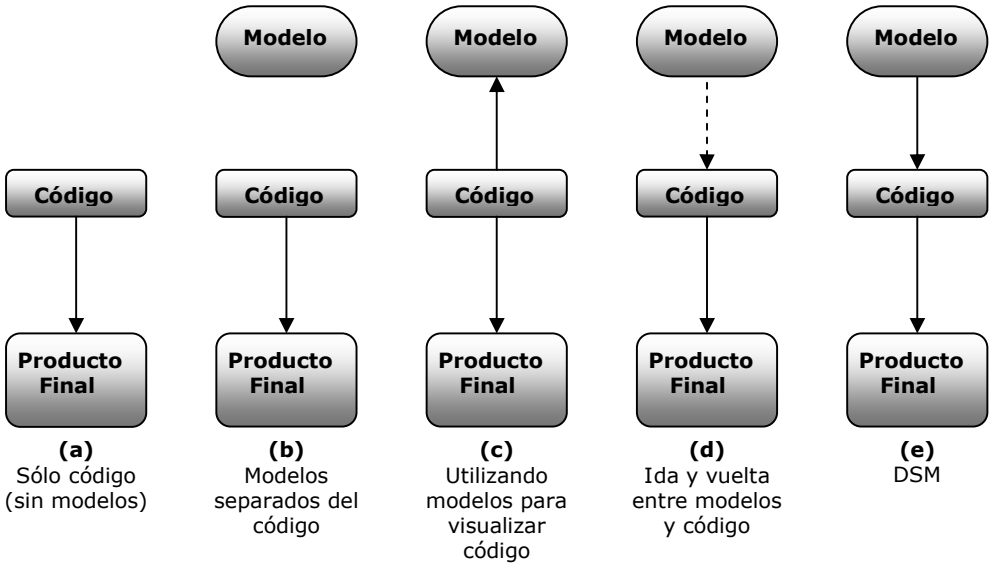


Figura 1-1 Distintas formas de relacionar el código con los modelos y su comparación con la metodología DSM.

La metodología DSM (e), a diferencia de los casos anteriores, tiene a los modelos como la fuente de información fundamental para derivar el código, obteniéndose luego la implementación final a partir de este último. Al generarse el código fuente desde los modelos, no se presentan los problemas anteriormente comentados a causa de la separación entre las etapas de modelado e implementación.

En la actualidad, el hecho de que los lenguajes de Tercera Generación permiten, a través de compiladores, generar código de bajo nivel confiable el cual no tiene que ser modificado por los desarrolladores, muestra que esta

derivación es factible y además conlleva a un incremento considerable en la productividad.

En la práctica, cada solución DSM diseña un lenguaje y un generador de código dentro de una organización en particular. Se enfocan en dominios pequeños, porque son más fáciles de definir y permiten mejores posibilidades para su automatización. Como los conceptos del lenguaje son utilizados dentro de la organización, se reduce considerablemente el tiempo de aprendizaje del mismo. Usualmente, las soluciones en DSM son utilizadas en relación a un producto particular, una línea de producción, un ambiente específico o una plataforma.

El desafío de los desarrolladores y empresas se centra en la definición de los lenguajes de modelado, la creación de los generadores de código y la implementación de framework específicos del dominio; elementos claves de una solución DSM. Estos no se encuentran demasiado distantes de los elementos de modelado de la filosofía MDD. MDD, sigla que corresponde a Model Driven Development o en castellano Desarrollo Dirigido por Modelos, promete mejorar el proceso de construcción de software basándose en un proceso guiado por modelos y soportado por potentes herramientas. El adjetivo "dirigido" (driven) en MDD, a diferencia de "basado" (based), enfatiza que este paradigma asigna a los modelos un rol central y activo: son al menos tan importantes como el código fuente. Los modelos se van generando desde los más abstractos a los más concretos aplicando transformaciones y/o refinamientos, hasta finalmente llegar al código. Las transformaciones entre modelos constituyen el motor principal de MDD. Actualmente puede observarse que las discrepancias entre DSM y MDD han comenzado a disminuir. Se puede comparar el uso de modelos así como la construcción de la infraestructura respectiva en DSM y en MDD. En general, DSM usa los conceptos dominio, modelo, metamodelo y meta-metamodelo como MDD sin mayores cambios y propone la automatización en el ciclo de vida del software.

Como contra, el desarrollo de un lenguaje es considerado una tarea difícil, ya que requiere un profundo conocimiento del dominio y mucha experiencia en el tema. Pocas personas tienen ambas habilidades, y debido a eso, la decisión de desarrollar un DSL es pospuesta muchas veces indefinidamente.

Finalmente, existe una familia importante de herramientas para crear soluciones en DSM que permiten diseñar un lenguaje específico de dominio, para luego construir herramientas de modelado y generadores de código para ese lenguaje.

- Microsoft con las "Software Factories" [24]. Una *Software Factory* es una línea de producción de software que combina herramientas de desarrollo extensible, tales como Visual Studio Team System, con contenido empaquetado como DSLs, patrones y frameworks, basados en recetas para construir tipos específicos de aplicaciones. Visual Studio provee

herramientas para definir los metamodelos así como su sintaxis concreta y editores.

- Metacase con su producto MetaEdit+ [21]. MetaEdit+ es una herramienta comercial, basada en repositorios, que usa una arquitectura cliente/servidor. El ambiente está implementado en VisualWorks.
- Eclipse [6] con su proyecto llamado Eclipse Modeling Project. Este proyecto está compuesto por otros subproyectos *open source* para el desarrollo de lenguajes de modelado y generación de código.

En esta tesina, haremos enfoque en la propuesta de Eclipse, ya que es la única de código abierto entre las mencionadas. Los proyectos en los que trabaja Eclipse se enfocan principalmente en construir una plataforma de desarrollo abierta, la cual está compuesta por frameworks extensibles y otras herramientas que permiten construir y administrar software. También permite a los usuarios mejorar el ambiente y extender su funcionalidad a través de la creación de plugins.

En particular, el proyecto de modelado en Eclipse se llama Eclipse Modeling Project (EMP en adelante) [14], el cual define un conjunto de plugins relacionados al modelado y a las tecnologías MDSD. Este proyecto está organizado lógicamente en subproyectos

Los proyectos que se presentarán entonces son los siguientes:

- EMF para la especificación de la sintaxis abstracta
- GMF, Eugenia, EMFText entre otros, para el desarrollo de la sintaxis concreta.
- MOFScript y JET para definir el procesamiento semántico.

En capítulo siguiente se presentarán los Lenguajes Específicos de Dominio, se mencionarán sus características y se los comparará con los lenguajes de propósito general. En el capítulo 3 se verá el proceso para definir la sintaxis abstracta y en el siguiente los plugins que brinda EMP para especificarla. El capítulo 5 abarcará la sintaxis concreta del lenguaje y el capítulo siguiente las herramientas de Eclipse que dan soporte para ello. En el capítulo 7 se tratará la semántica del lenguaje, para luego indicar los plugins que la implementan. El capítulo 8 contendrá la presentación de un caso de estudio y su implementación con las herramientas ya mencionadas. Por último, en el capítulo 9 se tratarán las conclusiones del trabajo.

Lenguajes Específicos de Domino

2

Desafortunadamente, no se encuentra entre la bibliografía una definición satisfactoria de DSL. Algunas de ellas son:

Un DSL es un lenguaje que permite, a través de notaciones apropiadas y abstracciones, expresar un dominio de problemas específico.

Un DSL es un lenguaje de programación o especificación de lenguaje dedicado a problemas de un dominio particular.

Un DSL es un lenguaje de programación o un lenguaje de especificación ejecutable que ofrece, a través de notaciones apropiadas y abstracciones, poder de expresividad focalizado en un dominio de problemas particular.

Debido a que la estas definiciones no llegan a cubrir el real significado de DSL, la mayoría de los autores se valen de ejemplos para dejar claro el concepto. Es más, muchas de las notaciones y lenguajes ampliamente utilizados hoy día pueden considerarse lenguajes específicos del dominio. Algunos de ellos bastante conocidos son: la notación BNF, SQL, LATEX, entre otros. La figura 2-1 muestra algunos de los lenguajes con su correspondiente dominio.

DSL	**Dominio**
Excel macro language	Planilla de cálculo
HTML	Páginas web con hipertexto
LATEX	Generación de documentos
Make	Construcción de software
SQL	Consultas en base de datos
VHDL	Diseño de hardware

Figura 2-1 Algunos de los DSLs más utilizados.

A lo largo de esta década se ha revitalizado el interés por los lenguajes específicos del dominio (DSL) con el surgimiento del paradigma del desarrollo dirigido por modelos en especial con MDA [19], Desarrollo Específico del Dominio y las factorías de software. Debemos tener en cuenta que este concepto no es algo nuevo, ya en los años 50 se idearon DSL para programar aplicaciones en máquinas controladas numéricamente. A modo de ejemplo se puede mencionar BNF para especificar gramáticas, que fue desarrollado

durante 1959. Los llamados lenguajes de cuarta generación (4GLs) son usualmente DSLs para aplicaciones con fuerte uso de bases de datos.

Se puede encontrar también a los DSL bajo otros nombres. Por ejemplo, pequeños lenguajes, macros, lenguajes de aplicación y lenguajes orientados a problemas. En todos estos casos, se sigue haciendo referencia a la idea que se presenta como DSL.

2.1. Lenguajes específicos de dominio vs Lenguajes de propósito general

Tradicionalmente se ha programado utilizando lenguajes que, en mayor o menor grado, nacieron con el propósito de poder abordar cualquier tipo de problemas, dejando al margen su propia naturaleza. Aunque esta afirmación resulta algo relativa, pues casi siempre el diseño de cualquier lenguaje de propósito general ha estado orientado al desarrollo en ciertos ámbitos, como puede ser Fortran para cálculo numérico, Cobol para programación de gestión, Pascal para la enseñanza de la programación o PHP y ASP para la programación de aplicaciones Web. Este tipo de lenguajes denominados de propósito general, aunque en muchos casos son considerados lenguajes de alto nivel, no aportan la suficiente abstracción como para atacar directamente el problema sin tratar con aspectos técnicos internos y alejados del dominio del problema.

Esta es la razón de que aunque proporcionan una gran potencia en relación a los lenguajes ensambladores todavía es posible pensar en un paso más que los acerque a los conceptos de los problemas como medio para facilitar la programación y así, acortar el tiempo de desarrollo y reducir los costes de la producción de software.

No existe una definición formal para saber si un lenguaje es restringido a un dominio en particular y cual no. En la literatura existen varias posiciones acerca de COBOL, si es un DSL para aplicaciones de negocios, o no. Si definiéramos una escala, por un lado al lenguaje BNF como representante del grupo de los DSL y por el otro a C++ como representante del grupo de los GPLs, Cobol estaría más cerca de este último.

Un DSL se diseña para suplir necesidades muy específicas dentro de un dominio de aplicación concreto, contrastando así con los lenguajes de propósito general. El objetivo es crear un lenguaje especializado para modelar artefactos de software de manera más sencilla, en el ámbito para el que fue determinado. Este ámbito establecerá los conceptos que deberán aparecer en el DSL. Además, el lenguaje debe ser capaz de generar el código final de estos modelos.

Los DSL por el simple hecho de estar diseñados específicamente para un dominio de aplicación son, por naturaleza, mucho más expresivos si los comparamos con los lenguajes de propósito general. También resultan más sencillos para el programador, que conoce perfectamente el dominio para el que fue diseñado el DSL. Podemos decir que los DSL acercan la programación

a personal cualificado en el dominio pero con escasos conocimientos de programación. Pero el desarrollo de un DSL es muy difícil, requiere conocimiento amplio del dominio y de desarrollo de lenguajes. Los expertos no suelen tener ambos conocimientos. Además las técnicas utilizadas son muy variadas y precisan de una mayor atención en todos los factores.

Otro problema fundamental es el aumento en el costo de las tareas de documentación, soporte técnico, mantenimiento y estandarización, en las que se produce un sustancial aumento en esfuerzo y tiempo. Es un factor a tener en cuenta especialmente en proyectos de gran magnitud.

2.2. Escalas de un DSL de acuerdo a su grado de ejecución.

Un DSL puede ser ejecutable de varias maneras y en varios grados, aún si es no ejecutable. De acuerdo a eso, los programas escritos con un DSLs son frecuentemente llamados especificaciones, descripciones o definiciones. A continuación se identifican algunos puntos en la escala de ejecutabilidad de un DSL.

Hay DSLs con una semántica de ejecución bien definida (por ejemplo, HTML, o el lenguaje de macros de Excel)

Hay DSLs que sirven como lenguaje de entrada a un generador de aplicaciones. Estos lenguajes son también ejecutables, pero frecuentemente tienen un carácter más declarativo y una semántica de ejecución pobremente definida en comparación a los detalles de la aplicación generada. El generador de la aplicación es un compilador para el DSL en cuestión.

Hay DSLs que no pretenden ser ejecutables, pero sin embargo, son útiles para la generación de una aplicación. El lenguaje de especificación de sintaxis BNF es un ejemplo de un DSL puramente declarativo que puede también ser usado como lenguaje de entrada para un generador de parsers.

2.3. Lenguajes de modelado vs lenguajes de programación

Tradicionalmente se tiene una distinción importante entre los lenguajes de modelado y los lenguajes de programación (tal vez hecha debido a que las diferencias entre las comunidades de modelado y programación). Una razón para esto es que los lenguajes de modelado solían tener una semántica abstracta e informal, mientras que los lenguajes de programación son significativamente más concretos debido a que necesitan ser ejecutables. Pero los lenguajes de modelado y programación son muy similares: ambos tienen sintaxis concreta, sintaxis abstracta y semántica. Si existe una diferencia, es el nivel de abstracción al que apunta el lenguaje. Por ejemplo, UML tiende a hacer foco es la especificación mientras que Java en la implementación. Sin embargo, aun esta distinción es borrosa: Java ha sido ampliamente extendido con características declarativas, como aserciones, y por otro lado, UML tiene sus versiones ejecutables.

Otra distinción común entre los lenguajes de modelado y los de implementación radica en forma de definir su sintaxis concreta. Los lenguajes de modelado suelen definir la sintaxis en forma de diagrama mientras que los lenguajes de programación de manera textual. Sin embargo, esta manera de representar su sintaxis no fuerza esta distinción: no tiene nada de importancia que un lenguaje de modelado no tenga sintaxis textual o que un lenguaje de programación no tenga sintaxis visual. Es solo una elección al representarlas. Actualmente existen definiciones textuales para UML y algunas herramientas que proveen unas vistas de programas escritos en Java, por ejemplo.

2.4. Lenguajes Gráficos vs Lenguajes Textuales

Existe una diferencia importante entre los lenguajes textuales y los lenguajes gráficos que radica básicamente en los conceptos de linealidad y paralelismo. Una expresión en un lenguaje textual tiene una estructura lineal, mientras que en un lenguaje gráfico tendrá una estructura más compleja, paralela, no lineal. Cada sentencia en un lenguaje textual es una serie de símbolos (o tokens como se conoce en el campo de diseño de parsers) ubicados en un cierto orden. En los lenguajes gráficos, los símbolos, tales como rectángulos o flechas, son los elementos básicos de una expresión gráfica. La diferencia esencial entre estos dos tipos de lenguajes es que en los lenguajes gráficos los símbolos pueden ser conectados de más de una manera, cosa que no ocurre con los textuales. Para ejemplificar esto, en la figura 2-2 se muestra una expresión típicamente no lineal graficada en forma lineal. La figura 2-2a muestra como se vería un diagrama de clases UML expresado en forma lineal. La expresión del mismo diagrama en forma no lineal se ve en la figura 2-2b.

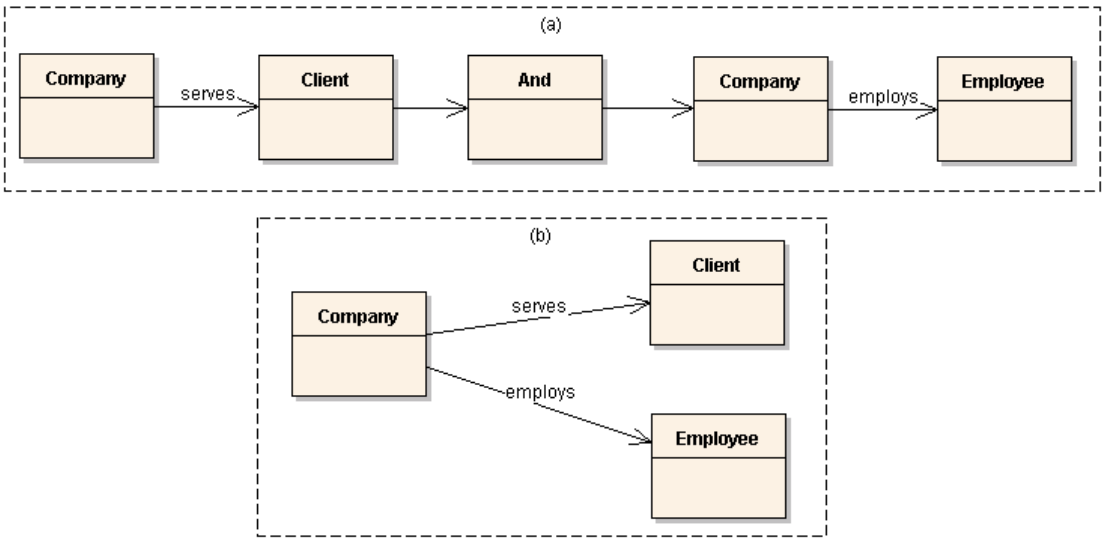


Figura 2-2 Una expresión lineal y una no lineal

El problema con las expresiones lineales es que un mismo objeto (Company en el ejemplo) debe aparecer dos veces porque no puede ser conectado a más de un elemento al mismo tiempo. Esto implica que se deba entender que dos ocurrencias de un mismo objeto representan la misma cosa.

Muchos lenguajes tienen un híbrido de sintaxis textual y gráfica. Por ejemplo, la notación de atributos y operaciones en un diagrama de clases en un UML es una sintaxis textual embebida en una gráfica.

2.5. DSLs vs Frameworks

Los frameworks se basan en la reutilización de diseños e implementaciones orientados a reducir los costos e incrementar la calidad del software. Un framework es una aplicación incompleta y reusable, que puede ser especializada para construir una aplicación a medida. Los frameworks no solo describen los componentes que lo forman sino también la forma en que interactúan entre ellos.

Frecuentemente se dice que los frameworks y APIs son lo mismo que DSLs, permitiendo al desarrollador trabajar con un conjunto diferente de conceptos. Esto es particularmente cierto, ya que es difícil separar la herramienta del lenguaje. El desarrollador, utilizando frameworks o APIs, parece estar trabajando con diferentes conceptos, frecuentemente con cierto nivel de abstracción. Pero al mismo tiempo, la salida –el programa que el desarrollador está creando- es escrito en un lenguaje de programación convencional.

Un DSL ofrecería a sus desarrolladores no solo un conjunto de conceptos más abstractos, sino también una nueva sintaxis. En cambio, tanto los framewoks como las APIs sólo agregan estructuras más complejas, y generalmente más abstractas, a un lenguaje existente. Es decir que no crean un nuevo lenguaje.

2.6. Expertos del dominio vs expertos en computación

Algunas personas creen que el uso de los DSL convierte a las personas que son expertas en el dominio en programadores de ese lenguaje, sin necesitar ayuda de especialistas en informática. Esta forma de pensar ha sido errónea en varios casos. A modo de ejemplo, Fortran se creó pensando que los programadores iban a ser reemplazados en su totalidad por matemáticos, ya que eran los expertos para el dominio que cubría ese lenguaje. Pero la historia demostró lo contrario.

Otros van un paso más adelante y piensan que el lenguaje específico de dominio puede ser desarrollado exclusivamente por el experto del dominio. Actualmente algo de esto está pasando, ya que se presentan casos en que compañías del mismo rubro colaboran para generar formatos comunes de

intercambio de datos. Esto permite definir una jerga común, la que aunque no sea un DSL en sí (correspondería mejor a crear una librería o un framework), da las bases para que se pueda definir en algún momento.

También hay que considerar el esfuerzo para desarrollar el nuevo lenguaje esté justificado con el soporte que va a brindar, así como la cantidad de usuarios que va a tener. Las herramientas para desarrollar DLS son tan complejas como lo son los lenguajes de propósito general, y si el esfuerzo de su desarrollo no es menor a las tareas que puede reemplazar, no tiene sentido su implementación.

2.7. Formalismos usados para especificar lenguajes

Muchos de los formalismos usados para definir lenguajes solamente permiten indicar una parte de la especificación del mismo. A continuación se detallan algunos formalismos usados para ese propósito.

2.7.1. Gramáticas libres de contexto

Las gramáticas libres de contexto permiten especificar la mayoría de los lenguajes de programación, de hecho, la sintaxis de la mayoría de ellos está definida mediante este tipo de gramáticas. Cuando se crea una gramática libre de contexto, se construye una especificación de una sintaxis concreta. La gramática entonces debe establecer todas las palabras clave y otros símbolos que luego están presentes en un programa del lenguaje nuevo.

Como cualquier gramática formal, una libre de contexto se puede definir mediante una 4-tupla:

$G = <V,\Sigma,S,P>$

V es el conjunto finito de caracteres no terminales o variables.

Σ es un conjunto finito de símbolos terminales, disjunto a V.

S es la variable inicial, la cual es usada para representar la sentencia completa. Debe ser elemento de V

P es un conjunto finito de producciones. Se define como una relación desde V a $(V \cup \Sigma)^+$.

A modo de ejemplo se puede mostrar la gramática libre de contexto de un lenguaje que consiste en todas las cadenas que se pueden formar con las letras a y b, de manera tal que la cantidad de a-es sea distinta al total de b-es.

$G = < \{S, U, V, T\}, \{a,b\}, S, P >$

donde las relaciones de P serian las siguientes:

$S \rightarrow U \mid V$

$U \rightarrow TaU \mid TaT$

$V \rightarrow TbV \mid TbT$

$T \rightarrow aTbT \mid bTaT \mid \varepsilon$

Siendo S el símbolo inicial, T genera todas las cadenas con la misma cantidad de letras a que b, U genera todas las cadenas con más letras a, y V todas las cadenas con más letras b.

La estructura subyacente de una gramática libre de contexto es un árbol de derivación, donde cada nodo representa:

- la aplicación de una regla de producción durante el reconocimiento de un elemento.
- un símbolo básico, es decir, un símbolo terminal.

Las distintas alternativas al obtener la expresión a partir del símbolo inicial pueden generar en algunos casos árboles de derivación distintos. En estos casos, decimos que la gramática es ambigua. Este tipo de gramática es más difícil de procesar, ya que el parser no puede conocer siempre que regla aplicar.

Las gramáticas libre de contexto solamente permiten especificar lenguajes textuales. No se puede crear un lenguaje gráfico usando únicamente una gramática libre de contexto. Si quisiéramos hacerlo necesitaríamos una extensión de estas gramáticas llamada gramática de atributos

2.7.2. Gramática de atributos

Las gramáticas de atributos son una extensión de las gramáticas libre de contexto a las cuales se les asocian atributos a sus símbolos no terminales. Estos atributos pueden ser de cualquier tipo, por lo que en muchos casos poseen una estructura compleja. A su vez, se utilizan para esconder información del elemento que representan. La forma abstracta completa de un elemento se puede crear como un atributo de un elemento concreto, como es una palabra clave.

Estos atributos tienen que ser consistentes, tanto en lo que significan como en el número de los mismos asociado a cada terminal. Sus valores se calculan mediante acciones, reglas o funciones semánticas asociadas a cada producción. Por lo que en general, las funciones semánticas toman como argumentos cero o más atributos de los símbolos que forman parte de la producción, y sólo pueden tomar como argumento los atributos de los símbolos a los que están asociados en la producción.

Los atributos se dividen en dos grupos: sintetizados y heredados. Los atributos sintetizados salen de la evaluación de las reglas y pueden usar valores de atributos heredados. Los atributos heredados son pasados desde los nodos ancestros del árbol de evaluación.

En algunos casos, los atributos sintetizados se utilizan para pasar información semántica hacia arriba. Esto permite a los compiladores asignar valores semánticos a las construcciones sintácticas y así, poder validar chequeos semánticos a la gramática asociada.

Un ejemplo de uso de gramática de atributos puede estar en una gramática que represente la construcción de programas Java. Así, al reconocer el código que implementa una clase, se puede construir el árbol de sintaxis abstracta que la representa y asignarlo al atributo del símbolo no terminal relacionado con la regla que representa la palabra clave "class".

2.7.3. Gramáticas de grafos

Se comenzaron a utilizar desde los setentas para representar lenguajes gráficos. Son más complejas que las anteriores, ya que utiliza dos mecanismos estructurales juntos. Por un lado, los diagramas son representados como grafos dirigidos donde los vértices indican las relaciones entre las partes del diagrama y los atributos o etiquetas que contienen los nodos y vértices representan elementos concretos del modelo. Y la aplicación de las reglas gramaticales de los diagramas forma una estructura que también se representa con un grafo. Comparado con una gramática libre de contexto, que tiene un árbol como estructura subyacente, esta representación es mucho más compleja.

El grafo que representa la aplicación de las reglas gramaticales se llama grafo de derivación. En este tipo de grafo, los nodos son asimismo grafos que representan los pasos de reconocimiento del diagrama ingresado. Todo esto logra una estructura demasiado compleja para tener un uso práctico en la definición de lenguajes y es por eso que se opta por mecanismos más simples.

2.7.4. Perfil UML

La creación de un perfil de UML [25] es una forma muy común de crear un lenguaje nuevo, aunque haya gente que no sepa que lo está haciendo cuando lo usa. Para crear un perfil, se toma como base la especificación de UML existente y se la modifica agregando estereotipos, valores con etiquetas y restricciones. Estos no son más que adornos que se agregan a la sintaxis concreta de UML para obtener la sintaxis buscada.

También hay libertad en indicar la semántica del nuevo lenguaje. UML define solamente las guías generales del significado de los conceptos del nuevo lenguaje. Por ejemplo, cuando se define uno nuevo basado en la construcción Class de UML, ya se sabe que los sistemas que representarán

los programas contendrán algún tipo de entidad instancia. Y cuando se definen nuevas asociaciones, ya se sabe que estos sistemas tendrán algún tipo de relaciones entre estas entidades.

Cabe destacar que el perfil en si mismo solo nos brinda el modelo de sintaxis abstracta del lenguaje. El modelo de sintaxis concreta, así como el mapeo a la sintaxis abstracta está dado por los estereotipos y la especificación de UML. Pero aún así la semántica sigue indefinida y debe ser especificada de alguna otra manera.

2.7.5. Metamodelado

Hace algunos años, los lenguajes se definían usando la gramática Backus Naur Form (BNF), en la cual se describe que secuencia de tokens forman una expresión correcta dentro del lenguaje. Este método es útil para lenguajes textuales, como lo son los lenguajes de programación. Ya que los lenguajes de modelado no tienen que estar basados en texto, generalmente no lo están (ya que tienen una sintaxis gráfica, como UML) y se necesita un mecanismo diferente para definirlos. Este mecanismo de definición se llama metamodelado.

La definición de UML en los noventa no solo proveyó un lenguaje de modelado sino que también popularizó el metamodelado como un medio para especificar lenguajes. Un metamodelo es un modelo, generalmente de la forma de un diagrama de clases, que describe los modelos que son parte del lenguaje. Por su facilidad de uso, fue la opción elegida en este trabajo. Sus características propias se verán en los siguientes capítulos con mayor detalle.

2.8. Definición de un lenguaje específico de dominio

Para definir un nuevo lenguaje es necesario definir su sintaxis abstracta, su sintaxis concreta y su semántica. Las siguientes secciones detallan estas tareas.

2.8.1. Sintaxis abstracta

La sintaxis abstracta de un lenguaje describe su vocabulario, es decir, los conceptos provistos por el lenguaje y como estos conceptos se pueden combinar para crear modelos. Una sintaxis abstracta consiste entonces de una definición de los conceptos, de las relaciones que existen entre estos conceptos y las reglas de buena formación que determinan como dichos conceptos pueden ser combinados de una manera correcta.

Consideremos un lenguaje de máquinas de estados simple, como la que se ve en la figura 2-3. En este caso, la sintaxis abstracta de este lenguaje puede incluir conceptos como estado, transición y evento. Además, habrá relaciones entre estos conceptos tales como que una transición está

relacionada con dos estados, uno fuente y otro destino. Finalmente, las reglas de buena formación se definen para asegurar, por ejemplo, que dos transiciones no se desencadenen por el mismo evento.

Es importante enfatizar en este punto que la sintaxis abstracta del lenguaje es independiente de su sintaxis concreta y de la semántica asociada. La sintaxis abstracta define la forma y la estructura de los conceptos del lenguaje, sin considerar su presentación o su significado.

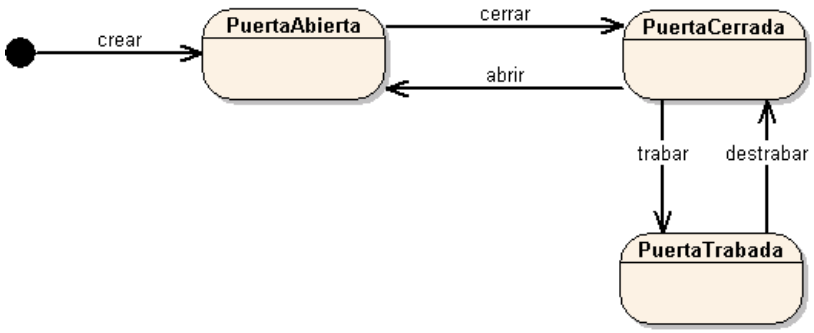


Figura 2-3 Estados de una puerta.

2.8.2. Sintaxis concreta

Todos los lenguajes proveen una notación que facilita la representación y la construcción de modelos o programas escritos en dicho lenguaje. Esta notación también se la conoce como sintaxis concreta. Las clases de sintaxis concretas se pueden dividir en dos tipos principales: sintaxis textual y sintaxis visual.

Una sintaxis textual permite escribir modelos o programas de manera textual [15]. Puede tener varias formas, pero típicamente consiste de una sección de declaraciones, donde se declaran los objetos y variables que van a estar disponibles dentro del programa y de un conjunto de sentencias asociadas.

El código Java siguiente muestra una sintaxis textual donde se declara una clase con una variable local y los correspondientes métodos para accederla.

```
public abstract class Person {
   private String name;

   public String getName(){
        return name;
   }
   public String setName(String newName){
```

```
            name:= newName;
        }
    }
```

La ventaja de las sintaxis textuales es la posibilidad de modelar o escribir expresiones complejas. Sin embargo, más allá de cierta cantidad de líneas se vuelve muy difícil de comprender y manejar.

Una sintaxis visual presenta un modelo o programa en forma de diagramas. Una sintaxis visual consiste de un número de iconos gráficos que representan vistas de un modelo subyacente. Un ejemplo muy conocido de sintaxis visual es un diagrama de clases, el cual provee una buena forma representar una vista de las relaciones y conceptos de un modelo.

El principal beneficio de una sintaxis visual es la habilidad de expresar mucha información de una manera intuitiva y entendible. La desventaja es obvia: solamente ciertos detalles pueden ser expresados en un gráfico ya que si se quisiera expresar todo el mismo se volvería demasiado complejo e incomprensible.

En la práctica se utiliza una mezcla de ambas, ya que se gana el beneficio de las dos. Así, los lenguajes frecuentemente usan las notaciones visuales para representar las vistas de alto nivel de un modelo, mientras que las sintaxis textuales se usan para capturar los detalles.

2.8.3. Semántica

La sintaxis abstracta no define que significan los conceptos dentro de un lenguaje. Por lo tanto se precisa información adicional para capturar la semántica de dicho lenguaje. Es importante definir la semántica de un lenguaje ya que se establece que se representa y que significan las construcciones en dicho lenguaje. De otra forma se podría malinterpretar.

Por ejemplo, aunque se tiene una idea intuitiva de que se quiere representar con una máquina de estados, es probable que la semántica detallada de un lenguaje esté abierta a la mal interpretación si no es definida de manera precisa. ¿Qué es exactamente un estado? ¿Qué significa que ocurra una transición? ¿Qué pasa si dos transiciones salen del mismo estado? ¿Cuál será elegida? Todas las respuestas a estas preguntas deben ser capturadas por la semántica de este lenguaje.

2.8.4. Mapeos

En el mundo real, los lenguajes no están aislados, sino que están relacionados con otros lenguajes. Esto se puede dar de varias maneras:

- Vía traducción donde los conceptos de un lenguaje son traducidos a conceptos de otro lenguaje equivalencia semántica. Un lenguaje puede tener conceptos cuyo significado esté relacionado con los conceptos de otro lenguaje

- Abstracción donde el lenguaje puede estar relacionado con un segundo lenguaje el cual tiene un diferente nivel de abstracción.

La captura de estas relaciones es una parte importante de la definición del lenguaje y sirve para posicionar el nuevo en el contexto del mundo. Además, los mapeos existentes entre los componentes internos del lenguaje, como entre la sintaxis concreta y la abstracta son una parte importante de su arquitectura.

2.8.5. Extensibilidad

Los lenguajes no son entidades estáticas: cambian y evolucionan a lo largo del tiempo. Por ejemplo, se pueden agregar nuevos conceptos o se pueden quitar algunos. La habilidad de extender un lenguaje de una manera precisa y organizada es vital para dar soporte a la adaptabilidad del lenguaje. Esto permite que el lenguaje se adapte a un dominio de aplicación nuevo y que evolucione para responder a nuevos requisitos.

2.9. Beneficios al definir un DSL.

¿Por qué es necesario definir un DSL? Como respuesta a esta pregunta podríamos decir que simplemente, porque permiten especificar el dominio de mejor manera:

Existen notaciones apropiadas, especificas dentro de un dominio, que ya se encuentran establecidas y están fuera del alcance de los lenguajes de propósito general. Un DSL permite que la notación específica de ese dominio sea más apropiada desde el comienzo. Esto es importante ya que está directamente relacionado con las mejoras en productividad asociadas al uso del DSL.

Los constructores y abstracciones apropiados para un dominio específico no siempre pueden ser mapeados en una manera sencilla en funciones u objetos de una librería. Es decir, un lenguaje de propósito general junto con las librerías podrían expresar estos constructores, pero indirectamente, de una manera más engorrosa. De nuevo, en este punto, un DSL podría incorporar constructores específicos dentro de un dominio desde el comienzo.

Al contrario de los GPLs, un DSL no necesita ser ejecutable. El modelo que se construye se puede utilizar para representar el programa a implementar en otro lenguaje.

2.10. Resumen

En este capítulo se vio que los DSLs se diseñan para suplir necesidades específicas dentro de un dominio de aplicación concreto, contrastando así con los lenguajes de propósito general. Su objetivo es crear un lenguaje

especializado para poder, de forma más sencilla, modelar artefactos de software en el ámbito para el que fue determinado. Además se mencionó la importancia del trabajo en conjunto de los expertos del dominio con el grupo de desarrollo.

Se vio que hay distintos formalismos para especificar los DSLs, algunos con más sencillez que otros. Entre los existentes se presentaron las gramáticas libres de contexto, de atributos, de grafos, los perfiles de UML y el metamodelado. Por la facilidad de uso del último, es la opción elegida para este trabajo.

Por último se vio que para especificar un DSL es preciso definir su sintaxis abstracta, concreta y la semántica del mismo.

En el capítulo siguiente se profundiza la definición de la sintaxis abstracta y se presenta la forma de especificarla con la propuesta de la OMG.

3 Sintaxis abstracta

El primer paso en el diseño de un lenguaje de modelado es construir su sintaxis abstracta. Este paso es tan importante como construir un modelo del sistema en un diseño orientado a objetos. La sintaxis abstracta describe los conceptos y las relaciones existentes entre ellos. Además, define reglas que determinan si un modelo escrito en ese lenguaje es un modelo válido. Estos conceptos, relaciones y reglas identificados en este paso proveen el vocabulario y la gramática para construir los modelos usando este lenguaje. Así, esta sintaxis constituye la base sobre la cual los otros artefactos del lenguaje se van a definir.

3.1. Modelado de la sintaxis abstracta

Como se mencionó, el propósito del modelo de la sintaxis abstracta es describir los conceptos del lenguaje y las relaciones existentes entre ellos. En el contexto de la definición del lenguaje, un concepto puede ser cualquier cosa que represente parte del vocabulario del lenguaje. El término de sintaxis abstracta enfatiza el hecho de que el foco se encuentra en la representación abstracta de estos conceptos y no en su representación concreta. Como consecuencia de esto, la sintaxis abstracta se enfoca en las relaciones estructurales existentes entre los conceptos del lenguaje y no en describir su semántica.

El modelo de una sintaxis abstracta también debe describir las reglas con las cuales se determina si un modelo escrito en ese lenguaje esta bien formado, es decir, es sintácticamente válido. Esto proporciona una descripción más detallada de las reglas sintácticas del lenguaje que si solo describiéramos los conceptos y sus relaciones. Las reglas de buena formación son particularmente útiles cuando se quiere implementar una herramienta que soporte del lenguaje, ya que se pueden usar para validar la correctitud de los modelos creados.

3.2. El proceso para modelar la sintaxis abstracta

Se pueden enumerar algunos pasos importantes en el desarrollo del modelo de una sintaxis abstracta, como son la identificación de conceptos, el modelado de los mismos, el modelado de la arquitectura, la validación y el testing de los modelos. Estos pasos se describen con más detalle en las secciones siguientes.

3.2.1. La identificación de los conceptos

El primer paso en el modelado de la sintaxis abstracta de un lenguaje es utilizar cualquier información disponible que ayude a identificar los conceptos del lenguaje, y cualquier regla obvia que posibilite validar o invalidar esos modelos.

Los conceptos del dominio del problema tienen algunas características que los hacen buenos candidatos a la hora de definir el lenguaje:

- Son conocidos y usados: se usan frecuentemente mientras se habla con los clientes, y también entre los desarrolladores. Esto hace que no sea necesario introducir nuevos conceptos en el lenguaje, ya que se construye con esos existentes, usados y aceptados. Incluso la gente se siente más cómoda con esos conceptos ya que no tienen que invertir tiempo en aprenderlos.
- Frecuentemente ya tienen establecida una semántica: el lenguaje se puede definir entonces en base a estas definiciones existentes. Por ejemplo, en el dominio de las aplicaciones para dispositivos móviles, se conoce el concepto de soft key: son dos o más teclas cuya función cambia basado en el contexto de la aplicación. De la misma manera, en todos los dominios existen conceptos conocidos inherentes a ellos. Si la semántica de un concepto en particular no puede ser definida, lo más probable es que no sea un concepto relevante para el lenguaje.
- Establecen un mapeo natural con el problema que se quiere resolver con el lenguaje. Hace que la creación del modelo sea más fácil y hace posible operaciones como el reuso de elementos del modelo en el dominio. Una estrecha relación con el dominio hace que los modelos sean fáciles de leer, entender, recordar, chequear y comunicar a los demás integrantes del equipo de desarrollo.

  Aunque no todos los conceptos del dominio del problema son candidatos a ser conceptos del lenguaje.
- Los conceptos comunes que tienen todas las aplicaciones o productos no son normalmente parte del lenguaje. ¿Por qué querríamos modelar algo que permanece siempre igual? En vez de eso debería proveerse mediante librerías o componentes o ser producido directamente por el generador. Al modelar un lenguaje deberían ser modelados solamente los conceptos que permiten describir alguna variación.
- Los conceptos que pueden ser identificados a partir de una combinación de otros existentes también deben ser excluidos. Por ejemplo, en el caso de telefonía móvil, las acciones de clickear, cancelar y seleccionar pueden ser representadas por diferentes relaciones de navegación: una para la selección, otra para cancelar y otra para acceder a los menús.

Es muy importante mantener el lenguaje lo más acotado posible, ya que se vuelve más fácil de aprender y usar.

3.2.2. Diferentes fuentes de conceptos

Identificar los conceptos relevantes para el lenguaje depende en gran medida de las ideas creativas y de la experiencia en el dominio. Se tiene una ventaja si se ha visto involucrado en tareas similares en el pasado. Siempre que sea posible se debe consultar a personas con más experiencia en el tema, como lo son los expertos en el dominio, arquitectos, y líderes de los equipos de desarrollo. Sus opiniones y puntos de vista son la principal fuente para la creación de la solución. Se los puede entrevistar, observarlos en su trabajo diario o usar otros mecanismos que revelen características del problema. Y hay que tener en cuenta que la persona que va a especificar el lenguaje e implementarlo como una herramienta no tiene que tener necesariamente experiencia en el dominio.

Los conceptos candidatos para el lenguaje de modelado se pueden encontrar en diferentes contextos. Se los puede hallar en la jerga utilizada, así como en el vocabulario de uso. Frecuentemente los conceptos usados existen por una buena razón y es que la gente los encuentra relevantes y concisos cuando discuten por un producto y sus características. El vocabulario provee entonces un buen punto de partida, ya que la gente no piensa en las soluciones en términos de código. Esto significa también que no hay necesidad de introducir términos nuevos, que no sean familiares o llamar los conceptos existentes con otros nombres.

Otras fuentes típicas para encontrar conceptos candidatos incluyen:

- La arquitectura: una descripción de la arquitectura es una buena fuente de información ya que aquí se opera con conceptos del dominio. Usualmente es la parte más estable y revela abstracciones importantes acerca de los elementos de la aplicación y su comportamiento
- Productos existentes, aplicaciones y correspondientes manuales: estos capturan la estructura, el comportamiento y la semántica, pero desafortunadamente hacer un análisis completo no siempre es práctico, ya que podría consumir demasiado tiempo examinar exhaustivamente todos esos productos existentes. Sería más conveniente entonces seleccionar un conjunto de aplicaciones representativo para inspeccionarlas con mayor detalle. Naturalmente en ese conjunto seleccionado deberían encontrarse las que tengan una relación más estrecha con las que quieren ser desarrolladas en el futuro. Luego también se podría inspeccionar otras aplicaciones con el mismo dominio u otras versiones de las mismas aplicaciones.
- Especificaciones disponibles: Analizando descripciones existentes de las características, aplicaciones o productos se puede comprender la estructura del dominio y traducirla a un esquema conceptual. Documentos de requisitos son especialmente buenos para alcanzar el nivel de abstracción ya que usualmente se enfocan en el dominio del problema más que en su implementación. Los requisitos también involucran la

terminología del cliente, ampliando así el uso del DSL: Los clientes podrán entonces leer y chequear sus modelos. Las especificaciones no necesitan ser formales o estar basadas en algún lenguaje de especificación. De hecho, los modelos hechos sin ninguna restricción permite a los modeladores capturar el dominio de una manera más efectiva. Inspeccionar como se quiere encarar el problema y usar vistas alternativas revelan inmediatamente la propuesta que se piensa que es más natural.

- Patrones: Si la compañía tiene definidos un grupo de patrones, pueden ser una fuente de conceptos del dominio y revelar estructuras comunes dentro del mismo.
- Código: Se puede identificar abstracciones de una manera *botton-up*. Esto implica examinar las aplicaciones existentes y generalizar las características encontradas en ellas. Esto no está limitado solamente a encontrar abstracciones para el lenguaje. Los conceptos provenientes del *look and feel* del sistema y los del experto en el dominio tienen un nivel más alto de abstracción que aquellos originados a partir del código, por lo que prevalecen sobre estos.

A veces, estas fuentes de conceptos candidatos para el lenguaje no están disponibles, ya sea porque el dominio es nuevo y no hay experiencias pasadas disponibles o porque no existen especificaciones. En este caso, el conocimiento del dominio está disperso en la organización y se encuentra solo en la mente de los individuos. Lo ideal entonces es crear ejemplos concretos de formas alternativas para especificar el problema. Ejemplos múltiples con sus respectivos prototipos ayudarán a identificar abstracciones y pueden ser usados luego para probar la solución.

Es muy importante que durante este proceso se distinga entre la sintaxis concreta del lenguaje y la sintaxis abstracta. Es muy común que la estructura de la sintaxis abstracta refleje su sintaxis concreta. Por ejemplo, consideremos un lenguaje que provee múltiples vistas superpuestas de un pequeño número de conceptos del modelo. En este caso, la sintaxis abstracta debería modelar solo los conceptos del modelo y no su sintaxis concreta.

Modelar los diagramas es un error muy común. Si se está en duda, se pueden responder las siguientes preguntas para ayudar a identificar los conceptos necesarios:

- ¿El concepto tiene un significado o es puramente para presentación? Si es solo para la presentación, entonces pertenece a la sintaxis concreta.
- ¿El concepto se deriva de otro o es una vista de una colección de otros objetos más primitivos? En el último caso, se debe definir una relación entre el concepto más rico y los más primitivos.

En general, los mejores modelos de sintaxis abstracta son los más simples. Cualquier complejidad debida a una representación en diagramas debe ser deferida a los modelos de sintaxis concreta.

3.2.3. Casos de uso

Una técnica útil para identificar los conceptos del lenguaje de modelado es considerar diferentes casos de uso asociados al uso del lenguaje. Esto es muy parecido a escribir una interfase para un metamodelo, es decir, una colección de operaciones que deberían ser usadas cuando se interactúa con el metamodelo. Algunos casos de uso típicos para esto incluyen crear y eliminar los elementos del modelo, pero también deberían incluir operaciones ricas semánticamente como transformar o ejecutar el modelo. Las descripciones detalladas que resultan sobre el análisis de los casos de uso son una gran fuente de conceptos para el lenguaje.

3.3. Modelado de los conceptos

Tan pronto como se hayan identificado las partes relevantes del lenguaje de modelado, se deben formalizar. La mejor forma de hacerlo es definiendo un metamodelo.

Metamodelar simplemente significa modelar el lenguaje: mapear los conceptos del dominio a los elementos del lenguaje, como los objetos, las propiedades de esos objetos, y las relaciones que existan entre ellos. El proceso de metamodelar debe ser soportado por un lenguaje de metamodelado que es a su vez, un lenguaje específico de dominio para especificar lenguajes de modelado. El lenguaje estándar para metamodelar es MOF (MetaObject Facility) [20] definido por la OMG [23]. Las siguientes secciones muestran las 4 capas de modelado definidas por la OMG en mayor detalle.

3.3.1. La arquitectura 4 capas de modelado definida por OMG

Usando un lenguaje de modelado, podemos crear modelos; un modelo especifica que elementos pueden existir en un sistema. Si se define la clase Persona en un modelo, se pueden tener instancias de Persona como Juan, Pedro, etc. Por otro lado, la definición de un lenguaje de modelado especifica que elementos pueden existir en un modelo. Por ejemplo, el lenguaje UML especifica que dentro de un modelo se pueden usar los conceptos Clase, Estado, Paquete, etc. Debido a esta similitud, se puede describir un lenguaje por medio de un modelo, usualmente llamado metamodelo. El metamodelo de un lenguaje describe que elementos pueden ser usados en el lenguaje.

En UML se pueden usar, entre otros elementos, clases, atributos y asociaciones, ya que el metamodelo de UML define que es una clase, que es un atributo y que es una asociación.

Como un metamodelo es también un modelo, el metamodelo en sí mismo debe estar escrito en un lenguaje bien definido. Este lenguaje se llama metalenguaje. Desde este punto de vista, BNF es un metalenguaje. En la Figura 3-1 se muestra gráficamente esta relación.

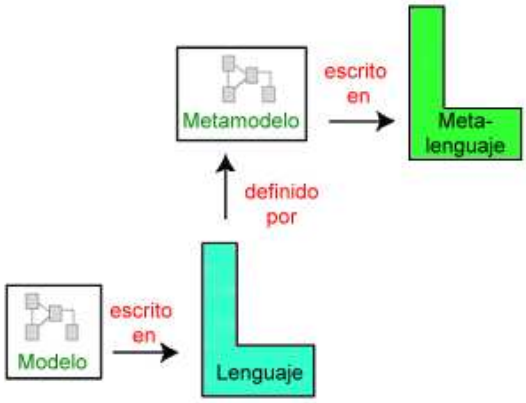


Figura 3-1 Modelos, Lenguajes, Metamodelos y Metalenguajes

El metamodelado entonces es un mecanismo que permite definir formalmente lenguajes de modelado, como por ejemplo, UML y OCL. La arquitectura de cuatro capas de modelado es la propuesta de la OMG orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos. Los cuatro niveles definidos en esta arquitectura se denominan comúnmente: M3, M2, M1 y M0.

Nivel M0: Instancias

En el nivel M0 se encuentran todas las instancias "reales" del sistema, es decir, los objetos de la aplicación. Aquí no se habla de clases ni atributos, sino de entidades físicas que existen en el sistema.

Para entender mejor la relación entre los elementos en las distintas capas, se presenta un ejemplo de cómo se modela un sistema con UML.

Por ejemplo, supongamos que se tiene un sistema de un videoclub, donde se maneja información acerca de los clientes y las películas de las cuales se dispone.

En la Figura 3-2 se muestra un diagrama de objetos UML donde puede verse las distintas entidades que almacenan los datos necesarios para este sistema. Se tiene un videoclub llamado "Videomanía", del cual se conoce el nombre y la dirección. Además, este videoclub tiene un cliente, Juan García, del cual queremos guardar su dirección y su dirección de email. Por último, el videoclub tiene una película con título "The Ring" de la cual se conoce la duración.

Todas estas entidades son instancias pertenecientes a la capa M0.

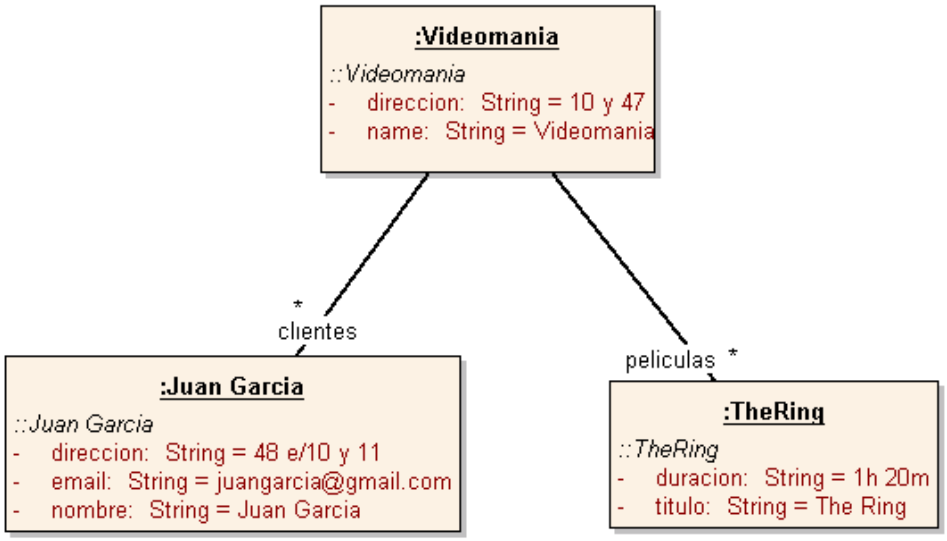


Figura 3-2 Entidades de la capa M0 del modelo de cuatro capas

Nivel M1: Modelo del sistema

Por encima de la capa M0 se sitúa la capa M1, que representa el modelo de un sistema de software. Los conceptos del nivel M1 representan categorías de las instancias de M0. Es decir, cada elemento de M0 es una instancia de un elemento de M1. Sus elementos son modelos de datos, por ejemplo entidades como "Cliente", "Videoclub", atributos como "nombre" y relaciones entre estas entidades.

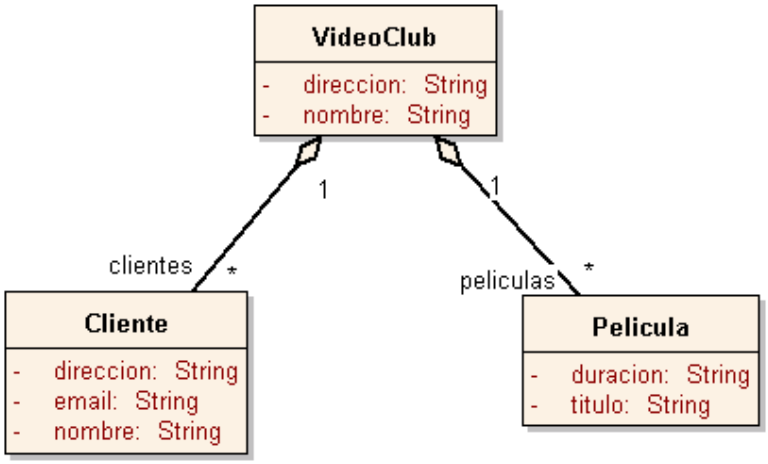


Figura 3-3 Modelo del sistema

En el nivel M1 aparece entonces la entidad Videoclub la cual representa los videoclubs del sistema, tales como "Videomanía", con los atributos

nombre y dirección. Lo mismo ocurre con la entidad Cliente y Película. En la figura 3-3 se muestra un modelo de clases UML para este ejemplo.

Videomanía puede verse ahora como una instancia de Videoclub. De la misma manera, el cliente de nombre Juan García puede verse como una instancia de la entidad Cliente y "The Ring" como una instancia de Película. En la figura 3-4 se muestra la relación entre el nivel M0 y el nivel M1.

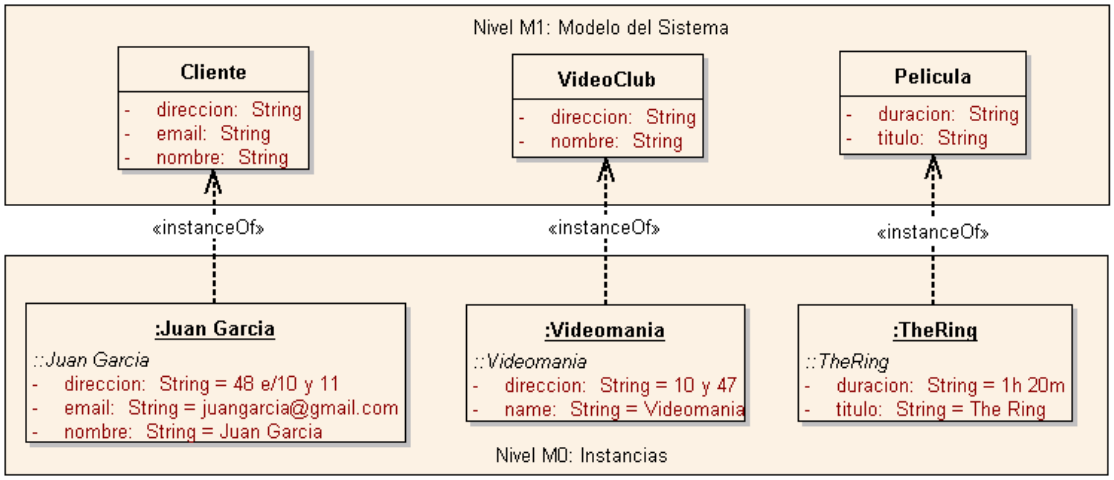


Figura 3-4 Relación entre el nivel M0 y el nivel M1

Nivel M2: Metamodelo

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

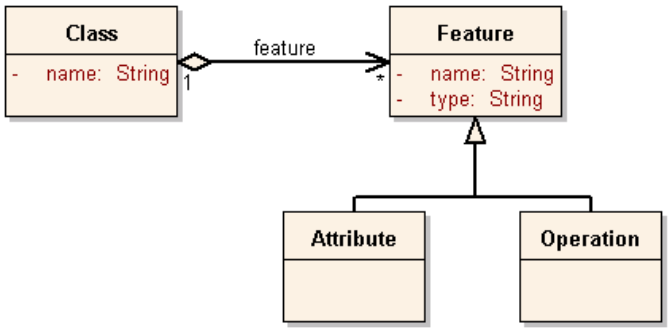


Figura 3-5 Parte del metamodelo UML

Siguiendo el ejemplo, la entidad Cliente será una instancia de la metaclase Class del metamodelo UML. Sus atributos, nombre y dirección serán instancias de la metaclase Attribute. En la figura 3-6 se muestra los

elementos del modelo Videoclub instanciados a partir de las metaclases del metamodelo UML.

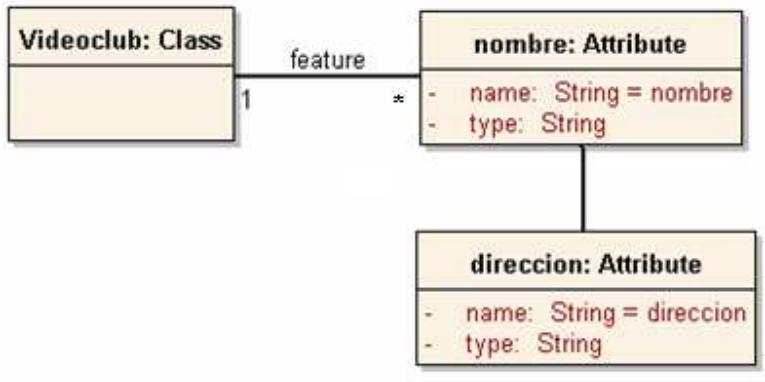


Figura 3-6 Modelo instanciado a partir del metamodelo UML

La figura 3-7 muestra la relación entre los elementos del nivel M1 con los elementos del nivel M2.

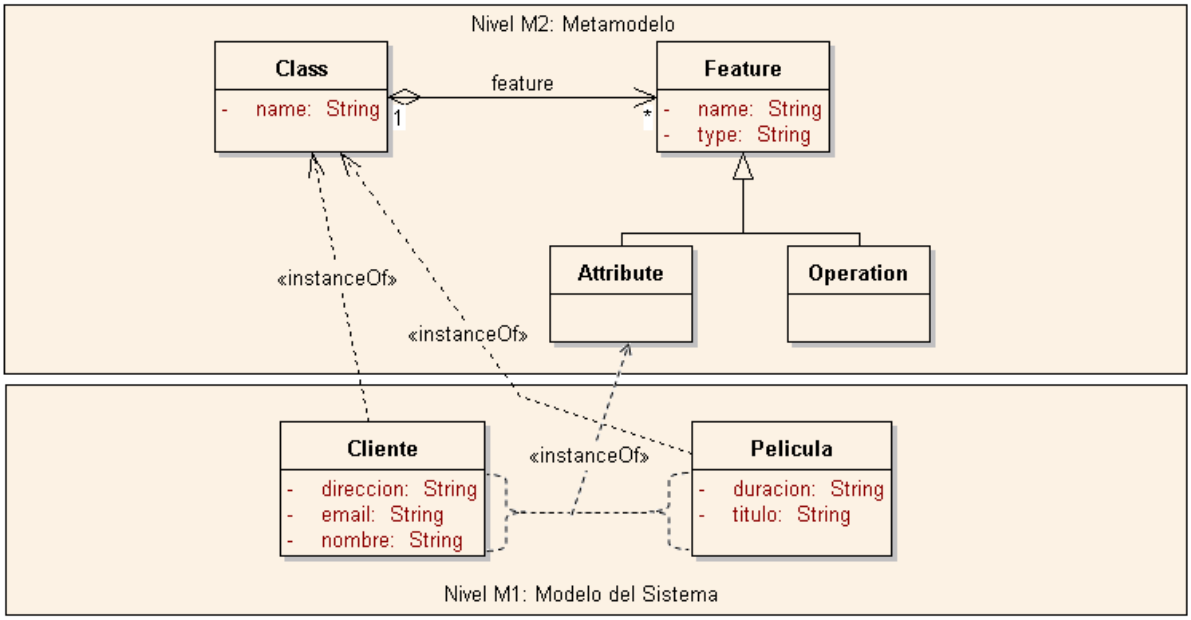


Figura 3-7 Relación entre el nivel M1 y el nivel M2

Nivel M3: Meta-metamodelo

De la misma manera podemos ver los elementos de M2 como instancias de otra capa, la capa M3 o capa de meta-metamodelo. Un meta-metamodelo (OMG, 2003) es un modelo que define el lenguaje para representar un metamodelo. La relación entre un meta-metamodelo y un metamodelo es análoga a la relación entre un metamodelo y modelo.

Es el nivel más abstracto, que permite definir metamodelos concretos. Dentro de la OMG, MOF es el lenguaje estándar de la capa M3. Esto supone que todos los metamodelos de la capa M2 son instancias de MOF. MOF puede ser usado para definir metamodelos orientados a objetos, como es el caso de UML y también para otros metamodelos, como es el caso de las redes de Petri o metamodelos para servicios web.

La figura 3-8 muestra la relación entre los elementos del metamodelo UML (Nivel M2) con los elementos del metamodelo de MOF (Nivel M3). Puede verse que las entidades de la capa M2 son instancias de las metaclases MOF de la capa M3

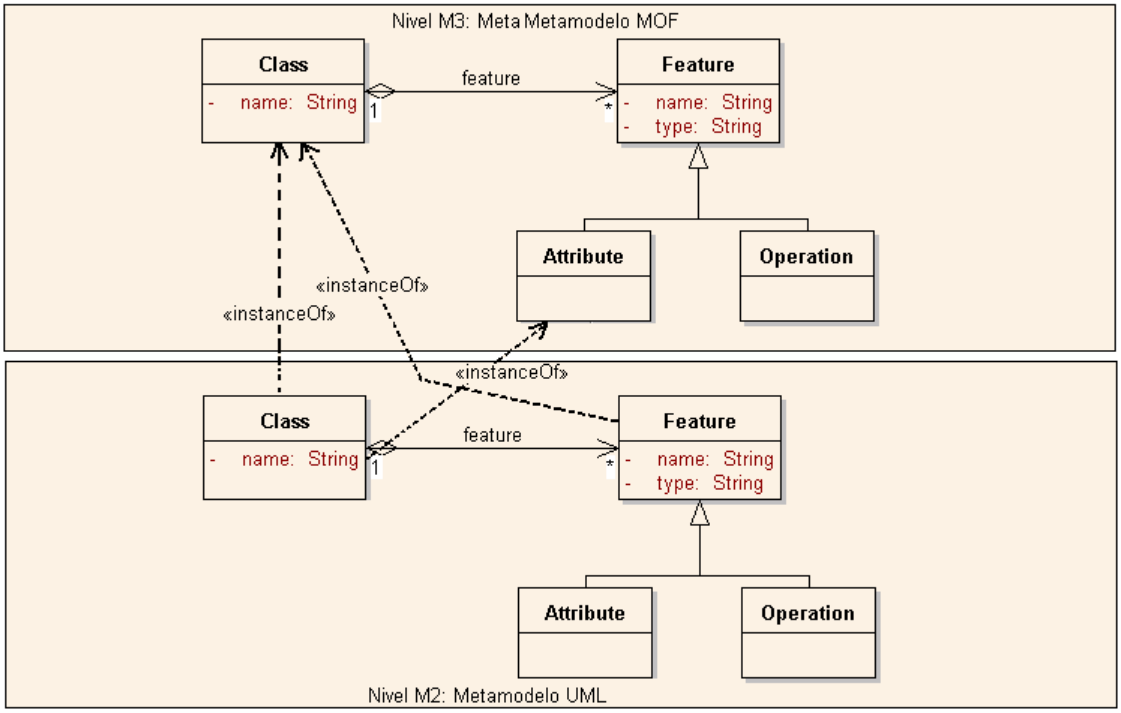


Figura 3-8 Relación entre el nivel M2 y el nivel M3

Por último, como vista general, la figura 3-9 muestra las cuatro capas de la arquitectura de modelado, indicando las relaciones entre los elementos en las diferentes capas. Puede verse que en la capa M3 se encuentra el meta-metamodelo MOF, a partir del cual se pueden definir distintos metamodelos en el nivel M2, como UML, JAVA u OCL. Instancias de estos metamodelos serán los elementos del nivel M1, como modelos UML o modelos Java. A su vez, instancias de los elementos M1 serán los objetos, como los objetos UML.

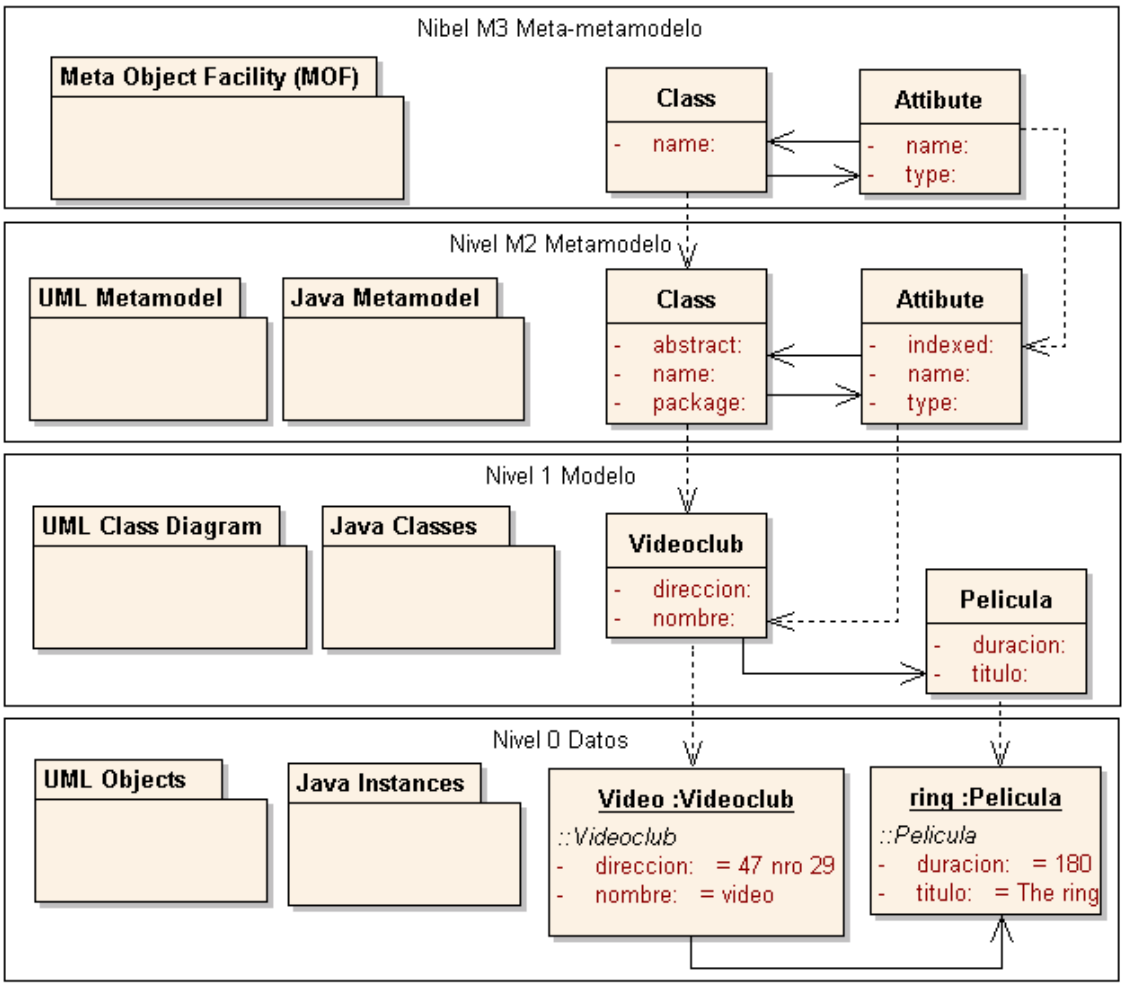


Figura 3-9 Vista general de las relaciones entre los 4 niveles

3.4. Reglas de buena formación

Una vez que se construye un modelo básico se comienzan a identificar ejemplos de modelos validos e inválidos que se pueden escribir con el lenguaje creado. Esta información se usa para definir un conjunto de reglas de buena formación que regulan los modelos ilegales. Cuando se definen estas reglas se busca identificarlas de la forma más general posible y se investiga que pasa cuando se las combinan. En algunos casos, las reglas pueden generar conflictos inesperados. Reusar los componentes de un lenguaje existente puede minimizar el esfuerzo de escribir reglas de buena formación, ya que se puede reutilizar las restricciones disponibles mediante la herencia de las mismas.

3.4.1. El lenguaje OCL

Los lenguajes gráficos de modelado son ampliamente aceptados en la industria. Sin embargo su falta de precisión ha originado la necesidad de utilizar otros lenguajes de especificación, como por ejemplo OCL (Object Constraint Language) [26], para definir restricciones adicionales. Con el uso de OCL se ofrece al diseñador la posibilidad de crear modelos precisos y completos del sistema en etapas tempranas del desarrollo. Sin embargo para estimular su uso en la industria es necesario contar con herramientas que permitan la edición y evaluación de las especificaciones expresadas en OCL.

Cualquier modelo puede ser enriquecido mediante información adicional o restricciones sobre los elementos del modelo. Estas restricciones pueden ser escritas en lenguaje natural, pero en ese caso no es posible chequear su validez y pueden ser mal interpretadas.

Un diagrama UML, como por ejemplo un diagrama de clases, en general no está lo suficientemente refinado para proveer todos los aspectos relevantes de una especificación. Existe, entre otras cosas, una necesidad de describir restricciones adicionales sobre objetos del modelo. Esas restricciones son a veces descriptas en lenguaje natural, pero en la práctica esto resulta siempre en ambigüedades. Como consecuencia, para escribir restricciones que no resulten ambiguas se han desarrollado algunos lenguajes formales, los cuales tienen la desventaja de tener que ser utilizados por personas con un gran conocimiento matemático. Esto los hace difíciles de usar por los modeladores de sistemas.

OCL fue desarrollado para solucionar este problema. Es un lenguaje formal y al mismo tiempo fácil de leer y escribir. Es un lenguaje puro de especificación, el cual garantiza estar libre de efectos laterales. Cuando se evalúa una expresión OCL simplemente retorna un valor, sin hacer modificaciones en el modelo. OCL no es un lenguaje de programación, es decir, no es posible escribir la lógica de un programa o flujos de control en OCL. No se pueden invocar procesos o activar operaciones que no sean de consulta con OCL.

Como OCL es un lenguaje con tipos, cada expresión del mismo tiene un tipo definido. Para que una expresión esté bien formada, debe ajustarse a las reglas de operaciones entre tipos del lenguaje; por ejemplo, no puede compararse un Integer con un String.

Especificación de una restricción en OCL

OCL permite especificar diferentes tipos de restricciones:

- Invariantes

- Pre y Postcondiciones

Con estas restricciones se pueden definir reglas de buena formación para el modelo.

Self

Cada expresión OCL está escrita en el contexto de una instancia de un tipo específico. En una expresión OCL la palabra reservada self es usada para referirse a esa instancia, a la cual se le especifica la restricción.

Invariantes

Una expresión OCL es una invariante si se estereotipa la expresión como <<invariant>>. Si una expresión OCL es una invariante para un elemento, debe evaluar verdadera para todas las instancias del elemento en todo momento. Por lo tanto, todas las expresiones OCL que denotan invariantes son del tipo Boolean.

El elemento del cual predica la expresión OCL, forma parte de la invariante y se escribe bajo la palabra clave *context* seguido del nombre del elemento. La etiqueta inv: declara que la restricción es una invariante.

Por ejemplo, para la definición de un metamodelo se podría pedir que no pueda especificarse una jerarquía múltiple. En este caso la invariante debe definirse sobre la meta-metaclase EClass de la siguiente manera:

```
context EClass inv:
   self.eSuperTypes <= 1
```

En general, la sintaxis de una invariante es:

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

Pre y Postcondiciones

Una expresión OCL es una precondición o postcondición si se estereotipa la expresión como << precondition >> o << prostcondition >> respectivamente. Una pre o postcondición debe estar ligada a una operación o método. Se muestra agregando "pre" o "post" en la declaración según corresponda.

En este caso, la instancia contextual self se refiere al elemento al que pertenece la operación.

En el caso de ser una precondición, la restricción establece una condición que debe cumplirse antes de ejecutar la operación. En el caso de ser una postcondición, la restricción establece una condición que debe cumplirse después de ejecutar la operación.

En general, la sintaxis para definir pre y postcondiciones es la siguiente:

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
pre : param1 > ...
post: result = ...
```

La variable param1 se refiere a los parámetros de la operación para la cual se define la restricción. La variable result se refiere al valor de retorno de la operación.

Si una expresión OCL es definida como invariante, la restricción debe cumplirse en todo momento, en cambio en las precondiciones y postcondiciones debe cumplirse antes y después de ejecutar la operación, según sea el caso. En una expresión OCL utilizada como postcondición los elementos se pueden decorar con el postfijo "@pre" para hacer referencia al valor del elemento al comienzo de la operación.

Opcionalmente, se puede escribir el nombre de la pre o postcondición después de las palabras claves "pre" o "post", lo que permite que la restricción sea referenciada por nombre. En el siguiente ejemplo, el nombre de la precondición es parameterOK, mientras que el nombre de la postcondición es resultOK.

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
pre parameterOk: param1 > ...
post resultOk : result = ...
```

Package Context

Para especificar explícitamente en que paquete de invariantes pertenece una invariante, una pre o postcondición, estas restricciones deben ser encerradas entre las palabras claves 'package' y 'endpackage', con la siguiente sintaxis:

```
package Package::SubPackage
```

```
        context X inv:
        ... some invariant ...

        context X::operationName(..)
        pre: ... some precondition ...

endpackage
```

Un archivo OCL puede contener cualquier número de invariantes, pre y postcondiciones.

Reglas de buena formación a nivel modelo

Una invariante es una expresión OCL que debe ser verdadera para todas las instancias del elemento del cual predica, en cualquier momento.

Es posible especificar invariantes a nivel modelo. Estas restricciones aseguran que los objetos a ser instanciados cumplen determinadas propiedades. Estas restricciones predican entonces sobre los elementos del modelo. Por ejemplo, en el contexto del videoclub, una regla de buena formación seria la siguiente.

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

Self en este caso se refiere a una instancia de Cliente. Esta invariante debe cumplirse para cada uno de los clientes concretos, es decir, para cada una de las instancias de la clase Cliente. Esta invariante chequea que los emails de los clientes contengan el carácter '@'.

Puede observarse que para definir una propiedad sobre los elementos del modelo es necesario predicar sobre la clase de la cual esos elementos son instancias.

Reglas de buena formación a nivel metamodelo

De manera análoga, si quisiera definir restricciones para los modelos hay que predicar sobre las clases de las cuales los modelos son instancias. Las reglas de buena formación a nivel metamodelo predican sobre los elementos del metamodelo. Estas restricciones predican sobre los meta-elementos definidos en el metamodelo, por ejemplo, dado el metamodelo de UML, se predica sobre Class, Association, etc. Un ejemplo para esta clase de reglas es que en una Clase no existan atributos con el mismo nombre o que una operación no tenga parámetros repetidos.

```
context Association
  inv WFR_1_Association:
  --[1] Los roles de una asociacioón deben tener distinto nombre
```

```
self.memberEnd ->forAll(p,q| p.name = q.name implies p = q)
```

Self en este caso se refiere a una instancia de Association. Esta invariante debe cumplirse para cada una de las instancias de Association, es decir, para cada una de las asociaciones definidas en el modelo.

La regla anterior valida que las asociaciones tengan nombres de rol distinto en cada uno de sus extremos.

Reglas de buena formación a nivel meta-metamodelo

Si ahora quisiéramos definir un nuevo metamodelo, por ejemplo, el metamodelo relacional, y quisiéramos asegurarnos que este metamodelo este bien formado, es necesario definir reglas de buena formación sobre los meta-meta-elementos o elementos de MOF. Las reglas de buena formación a nivel meta-metamodelo son restricciones que aseguran que un metamodelo está bien formado. Un ejemplo para esta clase de reglas es que un paquete no contenga metaclases con el mismo nombre o que una metaclase no tenga meta-atributos repetidos.

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

La regla anterior valida que todas las metaclases pertenecientes a un paquete tengan distinto nombre.

3.5. Operaciones y queries

Las operaciones y los queries deben ser definidos apropiadamente. Un ejemplo de operación es la creación de nuevos elementos del modelo, el seteo de los valores de los atributos, etc. Un ejemplo de queries es verificar el estado de alguna propiedad, para usar ese valor como entrada a una restricción u operación, o con propósitos de validación.

3.6. Validación y testing

Es de vital importancia validar la correctitud del modelo de sintaxis abstracta. Una técnica útil es construir instancias de la sintaxis abstracta que soporten los modelos dados por los ejemplos. Un diagrama de objetos es una manera útil de capturar esta información ya que muestra objetos (instancias de clases) y links (instancias de asociaciones). Existen herramientas disponibles que ayudan a hacer diagramas de objetos y que también

chequean si son validos con respecto al modelo y a cualquier restricción OCL que haya sido definida.

Pero la mejor manera de testear la correctitud de la definición de un lenguaje es construir una herramienta que lo implemente. Solamente entonces se puede testear el lenguaje por los usuarios finales con total libertad.

3.7. Resumen

El primer paso en el diseño de un lenguaje de modelado es construir su sintaxis abstracta. Durante este capítulo se listaron diferentes estrategias para identificar conceptos del dominio necesarios para su especificación. También se presentó la arquitectura de cuatro capas de modelado definida por la OMG y la importancia de los metamodelos en la definición de la sintaxis abstracta. Por último se presentó el lenguaje OCL que hace posible definir restricciones sobre el metamodelo creado.

Para implementar la sintaxis abstracta, es necesario disponer de herramientas que faciliten esta tarea. En el siguiente capítulo se presentan dos plugins para Eclipse que ayudan a su desarrollo.

4 Plugins para definir sintaxis abstracta

En este capítulo se presentan dos plugins para Eclipse que ayudan a definir la sintaxis abstracta de un lenguaje: EMF para la definición del metamodelo y OCL para la especificación de las reglas de buena formación y validación del lenguaje.

4.1. Implementación de MOF – Ecore

El metamodelo MOF está implementado mediante un plugin para Eclipse llamado Ecore. Este plugin respeta las metaclases definidas por MOF. Las mismas mantienen el nombre del elemento que implementan y agregan como prefijo la letra "E", indicando que pertenecen al metamodelo Ecore. Por ejemplo, la metaclase EClass implementa la metaclase Class de MOF.

El proyecto EMF [2] es un framework para modelado, que permite la generación automática de código para construir herramientas y otras aplicaciones a partir de modelos de datos estructurados. La información referida a este proyecto, así como también la descarga del plugin pueden encontrarse en [7].

EMF comenzó como una implementación del metalenguaje MOF. En los últimos años se extendió su uso, ya que se utilizó en la implementación de una gran cantidad de herramientas. Todo esto impactó en mejoras en la eficiencia del código generado. Se usó, por ejemplo, para implementar XML Schema Infoset Modelo (XSD), Servicio de Data Objects (SDO), UML2, y Web Tools Platform (WTP) para los proyectos Eclipse. Además EMF se utiliza en productos comerciales, como Omondo, EclipseUML, IBM Rational y productos WebSphere.

EMF permite usar un modelo como el punto de partida para la generación de código, e iterativamente aplicar varios ciclos de refinamiento y generación, hasta obtener el código requerido. Aunque también prevé la posibilidad de que el programador necesite modificar ese código, es decir, se contempla la posibilidad de que el usuario edite las clases generadas, para agregar o modificar métodos y variables de instancia. Siempre se puede regenerar desde el modelo cuando se necesite, y las partes agregadas serán preservadas en el resultado final.

4.1.1. Introducción

Como se dijo anteriormente, la generación de código es posible a partir de la especificación de un modelo. De esta manera, permite que el desarrollador

se concentre en el modelo y delegue en el framework los detalles de la implementación.

El código generado incluye clases Java para manipular instancias de ese modelo como así también clases adaptadoras para visualizar y editar las propiedades de las instancias desde la vista "propiedades" de Eclipse. Además se provee un editor básico en forma de árbol para crear instancias del modelo. Y por último, incluye un conjunto de casos de prueba para permitir verificar propiedades. En la figura 4-1 pueden verse dos pantallas. La de la izquierda muestra los cuatro plugins generados con EMF. Dentro de los plugins pueden verse los paquetes y las clases correspondientes. La pantalla de la derecha, muestra el editor en forma de árbol.

El código generado por EMF es eficiente, correcto, y fácilmente modificable. El mismo provee un mecanismo de notificación de cambios de los elementos, una implementación propia de operaciones reflexivas y persistencia de instancias del modelo. Además provee un soporte básico para rehacer y deshacer las acciones realizadas. Por último, establece un soporte para interoperabilidad con otras herramientas en el framework de Eclipse, incluyendo facilidades para generar editores basados en Eclipse y RCP.

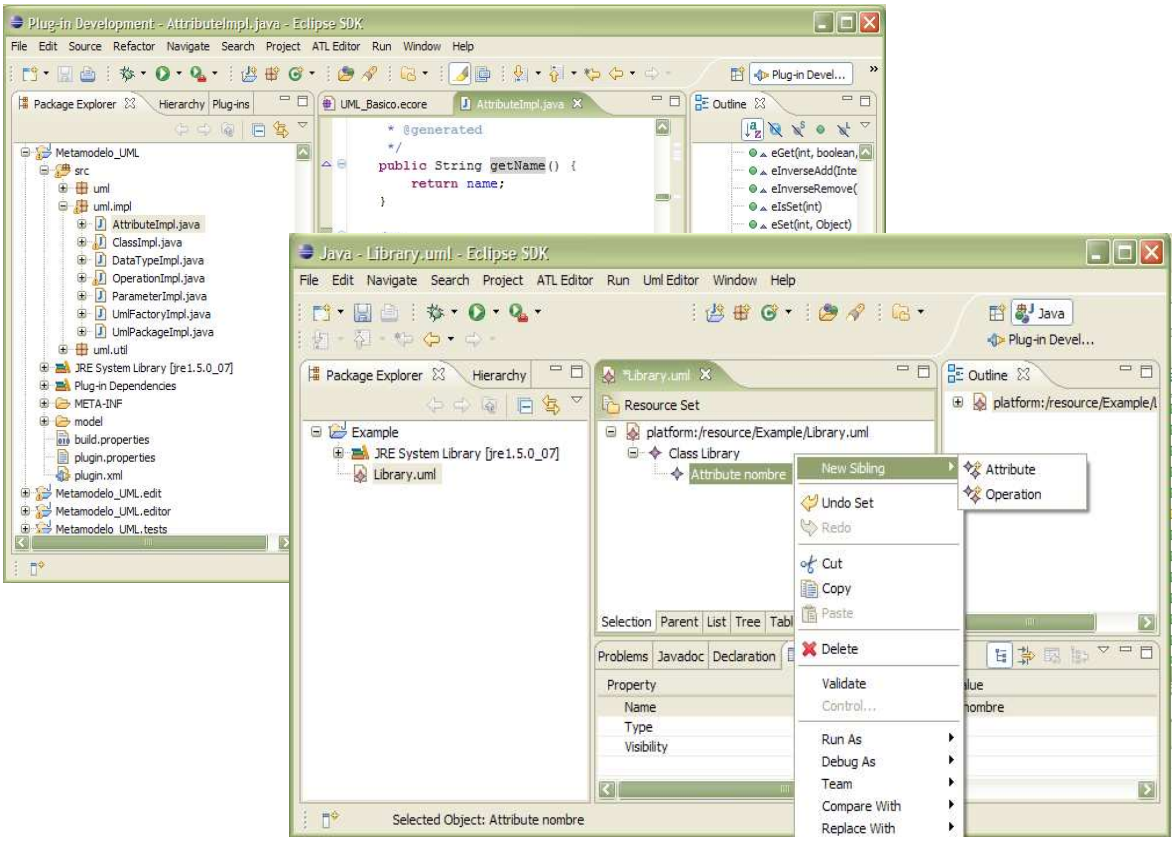


Figura 4-1 Plugins generados con EMF

4.1.2. Meta-metamodelo

En EMF los modelos se especifican usando un meta metamodelo llamado Ecore. Ecore es una implementación de MOF. Ecore en sí, es un modelo EMF y su propio metamodelo.

Existen algunas diferencias entre Ecore y MOF, pero aun así, EMF puede leer y escribir serializaciones de MOF haciendo posible un intercambio de datos entre herramientas que lo utilicen. La figura 4-2 muestra las clases mas importantes del meta metamodelo Ecore.

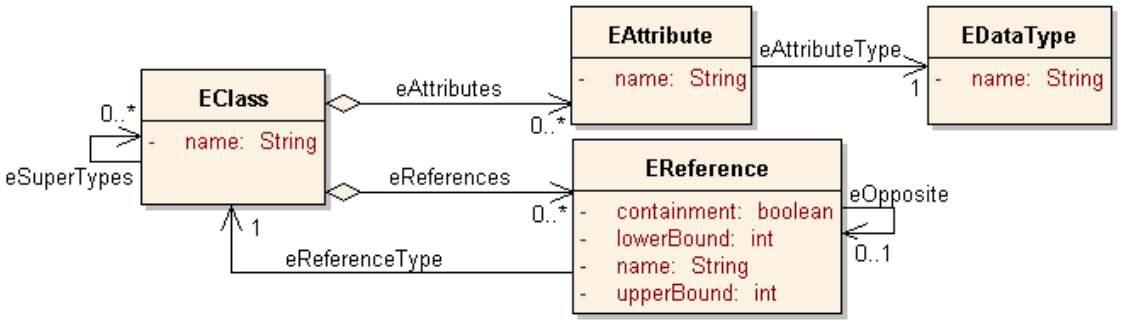


Figura 4-2 Parte del meta metamodelo Ecore

Las clases que se muestran en la figura son:

1- EClass: se usa para representar una clase modelada. Posee un nombre, atributos y referencias.
2- EAttribute: se usa para representar un atributo. Tiene un nombre y un tipo.
3- EReference: se usa para representar un final de asociación entre clases. Tiene un nombre, un boleano indicando si es agregación y una referencia al destino.
4- EDataType: se usa para representar el tipo de un atributo. Puede ser primitivo o un tipo de Java.

Entre las diferencias que existen entre Ecore y MOF, la principal está en el tratamiento de las relaciones entre las clases. MOF tiene a la asociación como concepto primario, definiéndola como una relación binaria entre clases. Además tiene finales de asociaciones con la propiedad de navegabilidad. En cambio Ecore, define solamente EReferences como un rol de una asociación, sin finales de asociación ni Association como metaclase.

Los modelos son instancias del metamodelo Ecore y son guardados en formato XMI (XML Metadata Interchange), que es la forma canónica para especificar un modelo.

4.1.3. Pasos para generar código a partir de un modelo

Un metamodelo puede especificarse con un editor gráfico de Ecore, o de cualquiera de las siguientes formas: como un documento XML, como un diagrama de clases UML o como interfaces de Java con Anotaciones. En este último caso, EMF las inspecciona y a partir de ellas deduce las propiedades del modelo. Para hacer esto, examina los métodos "getters" buscando los que tienen anotaciones de modelado. Por ejemplo, los que de desean incluir se señalan con @model.

EMF provee asistentes para interpretar ese metamodelo y convertirlo en un modelo EMF. Es decir, en una instancia del meta-metamodelo Ecore, que es el metamodelo usado por EMF (ver Figura 4-3).

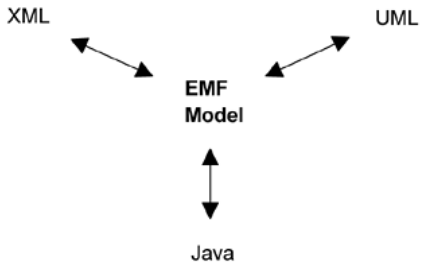


Figura 4-3 Puntos de partida para obtener un modelo EMF

La ventaja de que se permita partir de un modelo UML, es que este es un lenguaje conocido y por lo tanto no se requiere entrenamiento para usarlo. Además las herramientas de UML suelen ser amigables y proveen gran funcionalidad. Solamente se debe tener en cuenta que la herramienta permita exportar un modelo core serializado, de tal manera que pueda ser usado como entrada para el framework EMF.

A partir de un modelo representado como instancias de Ecore, EMF puede generar el código para ese modelo.

EMF provee un menú contextual con cinco opciones:

1- Generate Model Code
2- Generate Edit Code
3- Generate Editor Code
4- Generate Test Code
5- Generate All

A través de estas opciones de menú permite generar cuatro plugins. Con la primera opción se genera un plugin que contiene el código java de la implementación del modelo. Es decir, un conjunto de clases Java que

permiten crear instancias de ese modelo, hacer consultas, actualizar, persistir, validar y controlar los cambios producidos en esas instancias. Con la segunda opción se genera un plugin con las clases necesarias por el editor. Contiene un conjunto de clases adaptadoras que permitirán visualizar y editar propiedades de las instancias en la vista "propiedades" de Eclipse. Estas clases proveen una vista estructurada y permiten la edición de los objetos de modelo a través de comandos. Con la tercera opción se genera un editor para el modelo. Este plugin define además la implementación de un asistente para la creación de instancias del modelo. Con la cuarta opción se generan casos de prueba, que son esqueletos para verificar propiedades de los elementos. Finalmente, con la última opción se pueden disparar la generación de todos los casos mencionados previamente.

El código generado para el modelo tiene las siguientes características:

- Por cada clase del modelo Ecore se crean dos elementos en Java: una interface y la clase que la implementa. Esta separación es una decisión de diseño impuesta por EMF. Con esto se intentó seguir la filosofía de buena práctica de programación que implica separar la interfaz de la implementación. Además es un patrón necesario para soportar herencia múltiple.
- Las interfaces generadas extienden directa o indirectamente a la interface EObject. La interface EObject es el equivalente de EMF a java.lang.Object, es decir, la base de todos los objetos en EMF. EObject y su correspondiente implementación EObjectImpl proveen el comportamiento general para implementar operaciones de reflexibilidad y persistencia sobre las instancias de los elementos modelo.
- Se generan mecanismos de notificaciones para todos los objetos modelados. Las notificaciones de cambios siguen el patrón de diseño Observer [10]. Su especificación se encuentra en la interface Notifier, la cual es extendida por EObject. Este mecanismo se pone en marcha cada vez que se modifica un atributo, ya que en el código generado incluye la invocación de la notificación en su "setter".
- Se crea una clase Factory para instanciar los objetos del modelo y una clase Package con los accesos a todos los metadatos del modelo.
- Facilita la persistencia de objetos ya que el framework provee un mecanismo simple y configurable para manejarla. Por defecto incluye la serialización a XMI. El framework EMF se combina con el código generado para permitir grabar los objetos en cualquier forma persistente que uno quiera. Si no se utiliza la brindada por defecto, será necesario agregar la funcionalidad que lo permita.

En resumen, el código generado es limpio, simple, y eficiente. La idea es que el código que se genera sea lo mas parecido posible al que el usuario hubiera escrito, si lo hubiera hecho a mano. Debido a que es generado, se puede afirmar que es correcto. Como se mencionó, el generador de código de

EMF produce archivos que pretenden que sean una combinación entre las partes generadas y las partes modificadas por el programador. Se espera que el usuario edite las clases generadas, para agregar o editar métodos y variables de instancia. Hay que destacar que siempre se puede regenerar desde el modelo cuando se necesite, y las partes agregadas serán preservadas durante la regeneración. EMF usa los marcadores *@generated* en los comentarios Javadoc de las interfaces, clases y métodos generados para identificar las partes generadas. Cualquier método que no tenga ese marcador se mantendrá sin cambios luego de una regeneración de código. Si hay un método en una clase que está en conflicto con un método generado, la versión existente tendrá prioridad.

En la figura 4-4 puede verse el editor generado por EMF. A la derecha se encuentra la vista outline, que muestra el contenido del modelo que se está editando con una vista de árbol. Cuando se selecciona un elemento en el outline se muestra también seleccionado en la primera página del editor, que tiene una forma de árbol similar.

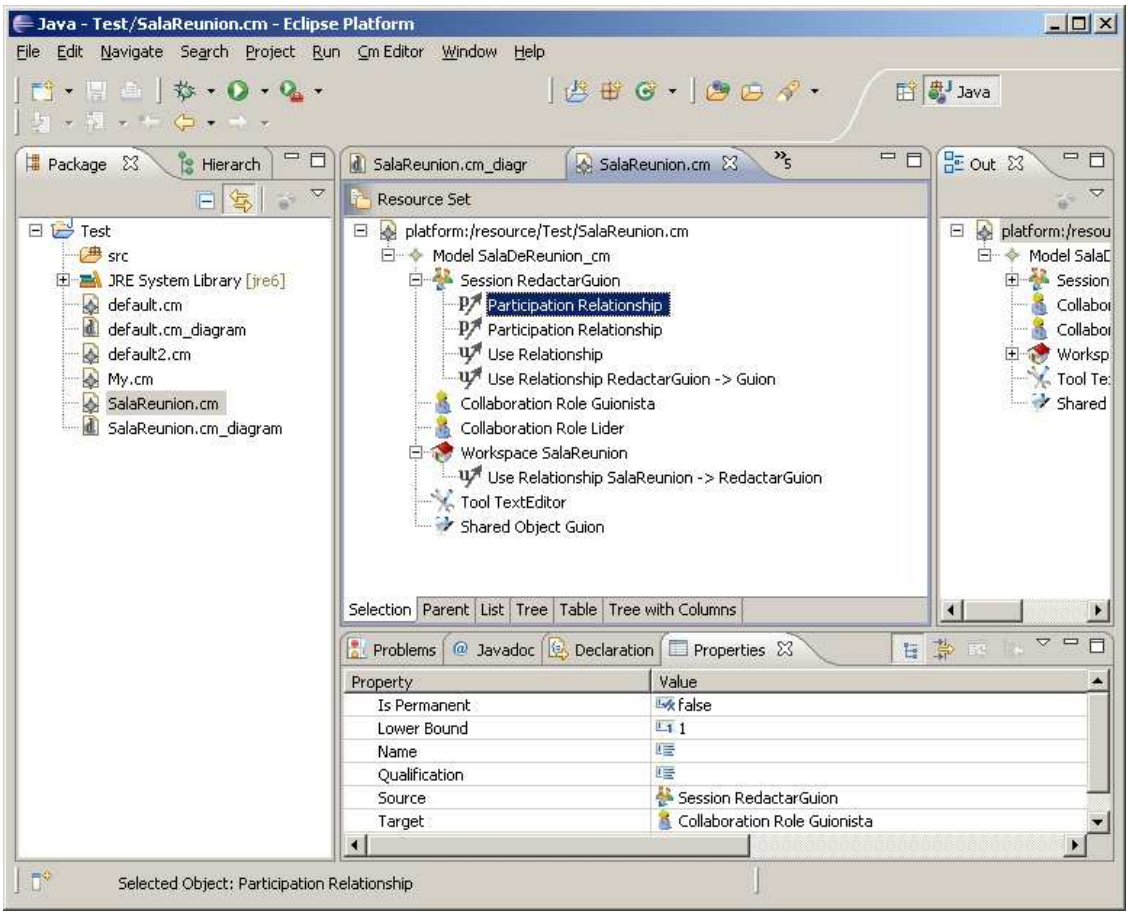


Figura 4-4 Editor generado con EMF

Independientemente de donde se seleccione el elemento, se muestran sus propiedades en la vista de propiedades. Esta vista permite editar los atributos del elemento y las referencias a otros del modelo (seleccionando de una lista de posibles candidatos).

Permite arrastrar con el mouse los elementos para moverlos, cortarlos, copiarlos, pegarlos y borrarlos. Además estas acciones soportan deshacer y rehacer.

Las otras páginas del editor, Parent, List, Tree, Table, TableTree permiten otras visualizaciones de los elementos, como en forma de tabla y en forma de lista.

4.2. OCL

El plugin OCL [11] permite asegurar que el modelo esté correctamente formado ya que facilita la definición de restricciones sobre el modelo Ecore para luego evaluarlas. Además provee facilidades para definir los cuerpos de métodos agregados al modelo Ecore y reglas para calcular valores derivados.

OCL es un lenguaje libre de efectos laterales. Esto significa que ninguna expresión OCL puede modificar elementos del modelo. Ni siquiera los objetos temporarios que se crean al resolver la expresión (como strings, colleciones, tuplas, etc).

4.2.1. EMF y OCL

El uso de OCL dentro de EMF se realiza mediante la inclusión de anotaciones en el metamodelo. En las mismas se indica, además de la implementación, el tipo de regla OCL a definir. Estas anotaciones complementan el sistema de anotaciones de EMF. Este último sólo provee el esqueleto de los métodos en que se validan las restricciones.

Para poder utilizar el mecanismo de validaciones que provee el plugin de OCL es necesario incluir además un conjunto de plantillas JET para guiar el proceso de generación de código. En general, el código generado no es una implementación en Java de las reglas definidas, sino que contiene una invocación que luego será interpretada por el plugin OCL.

Dentro de un modelo Ecore es posible definir tres tipos de expresiones OCL: invariantes, propiedades derivadas y cuerpos para las operaciones.

Invariantes:

Este tipo de expresiones OCL definen restricciones sobre el modelo. Las mismas son almacenadas en los metadatos de EMF. Es decir, no se traducen a código JAVA. Esto tiene como ventaja que se permite cambiar la definición

de la restricción y volver a evaluar el modelo sin tener que regenerar el código.

Al definir una restricción en el modelo Ecore y generar el código, se crea para ella un método de validación. Dentro de este método se parsea la restricción y se construye un query ejecutable para chequear si el modelo es válido o no.

Propiedades derivadas:

EMF implementa las propiedades derivadas como *features* estructurales los cuales son marcados como *transient*, es decir, no persistentes y *volatile*, es decir, que no es necesario alocar espacio para almacenarlos.

Al definir una propiedad derivada en el modelo Ecore y generar el código, se crea para dicha restricción un método parecido al explicado para la validación de restricciones salvo por los siguientes detalles:

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

Operaciones:

OCL se usa frecuentemente para especificar pre y poscondiciones para las operaciones. Una tercera clase de expresiones definidas sobre las operaciones es la expresión *body*, la cual define el valor de la operación en términos de sus parámetros y las propiedades disponibles dentro del contexto.

Una vez más, al generar el código, se crea el método definido en la clase el cual tiene definida una implementación. El contexto de la expresión OCL, es una operación lo cual asegura que los nombres y tipos de los parámetros son visibles.

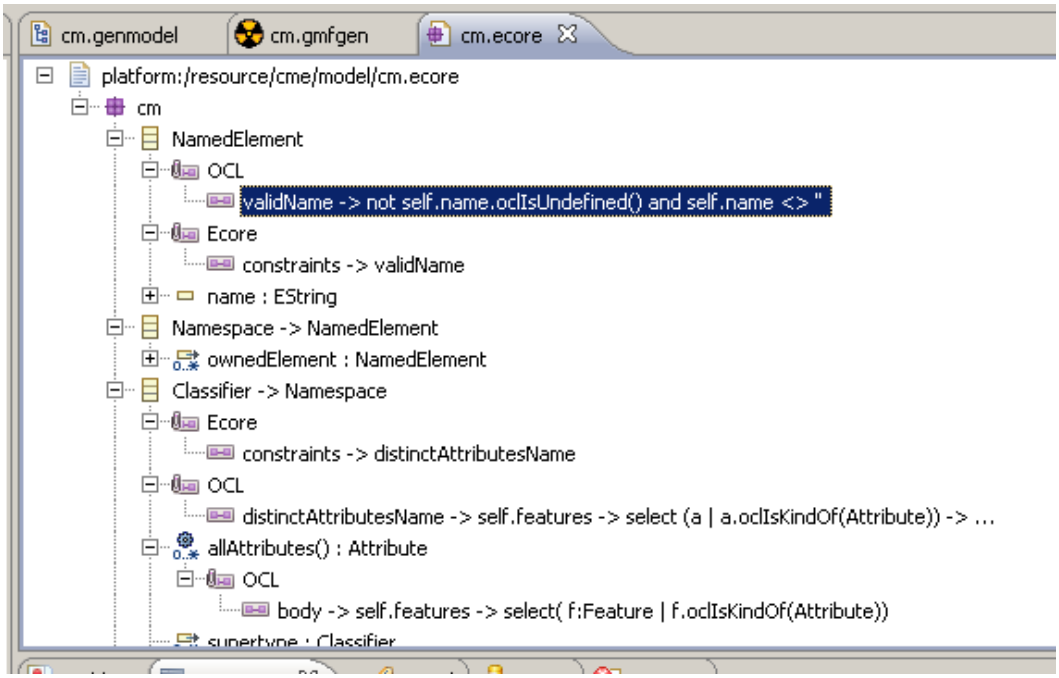


Figura 4-5 Expresiones OCL

En la figura 4-5 se puede ver la especificación del invariante validName que fuerza a que todos los elementos del modelo tengan nombre asignado. Además puede verse la definición de la operación allAttributes, la cual devuelve todos los atributos de la entidad Classifiers.

4.3. Resumen

En este capítulo se presentó el plugin EMF para definir la sintaxis abstracta. Además, se introdujo el plugin OCL que permite la definición y posterior evaluación de restricciones definidas en el lenguaje y como éste puede ser usado conjuntamente con EMF.

De la misma manera que la sintaxis abstracta juega un rol central en la descripción del lenguaje, la sintaxis concreta define como se mostrará ese lenguaje al usuario. El siguiente capítulo se centra en este tema.

5 Sintaxis concreta

La sintaxis abstracta describe el vocabulario y la gramática del lenguaje, pero no define como se mostrará ese lenguaje al usuario. Estas vistas son descriptas por la sintaxis concreta, y pueden ser en forma de diagrama o en forma de texto.

Muchos de los lenguajes de modelado usan sintaxis concreta en forma de diagramas, como máquinas de estado o los diagramas de clases; aunque frecuentemente se ven limitados por el tamaño del diagrama. Por otro lado, existen algunos lenguajes que tienen sólo una sintaxis textual. Ejemplo de esto último es el lenguaje OCL.

El proceso de definir la sintaxis concreta tiene dos pasos: el primero es interpretar la sintaxis y asegurar que sea válida. El segundo es usar la sintaxis concreta para instanciar la sintaxis abstracta. Estos pasos son igualmente aplicables si la sintaxis es en forma de texto o de diagrama, aunque existe una diferencia importante en la manera en que las sintaxis concretas se construyen por el usuario final. Los diagramas se crean comúnmente en forma interactiva e incremental y, como consecuencia, la sintaxis debe ser interpretada en paralelo con la interacción del usuario. Por otro lado, la sintaxis textual es frecuentemente interpretada en *batch*. El usuario construye un modelo completo usando la sintaxis, y luego éste es pasado al intérprete. Este estilo es comparable a los compiladores e intérpretes de los lenguajes de programación.

De la misma manera que la sintaxis abstracta juega un rol central en la descripción del lenguaje, la sintaxis concreta es crucial al diseñarlo. Además se puede definir como un elemento separado en la descripción del lenguaje.

El rol de la sintaxis concreta es representar un programa a los sentidos del usuario. Usualmente se hace por medio de una herramienta, siendo la más importante de ellas el editor, el cual se utiliza para crear o cambiar el programa. El tipo de editor usado influye en el modo en que se piensa en la sintaxis concreta y su mapeo a la sintaxis abstracta. También hay que considerar si el lenguaje definido es textual o gráfico.

5.1. Fases en el proceso de reconocimiento

La manera en que se deriva la forma abstracta desde su representación concreta, es un proceso de reconocimiento. Se debe identificar que partes de la forma concreta representan cuales elementos de la forma abstracta. Este proceso se define en forma gradual. No hay un punto preciso donde la representación concreta se convierte en la forma abstracta. De todas maneras, el resultado final de este proceso será lo que se conoce como la forma abstracta del programa.

Desde teoría de compiladores se conoce mucho sobre este proceso de reconocimiento. Sin embargo la tecnología ya no pone foco en lenguajes textuales, por lo que se debe agrandar el espectro para incluir lenguajes gráficos.

Tradicionalmente, el compilador transforma una cadena de caracteres en un árbol sintáctico. Para incluir lenguajes gráficos se necesitan los siguientes ajustes:

- La salida se debe instanciar a un grafo, en lugar de un árbol.
- La entrada no siempre será una cadena de caracteres, por lo que otro formato debe ser elegido.

Sin embargo cuando se miran de cerca las distintas fases de este proceso, los ajustes que se deben realizar son mínimos. La figura 5-1 muestra las similitudes entre un lenguaje gráfico y uno textual.

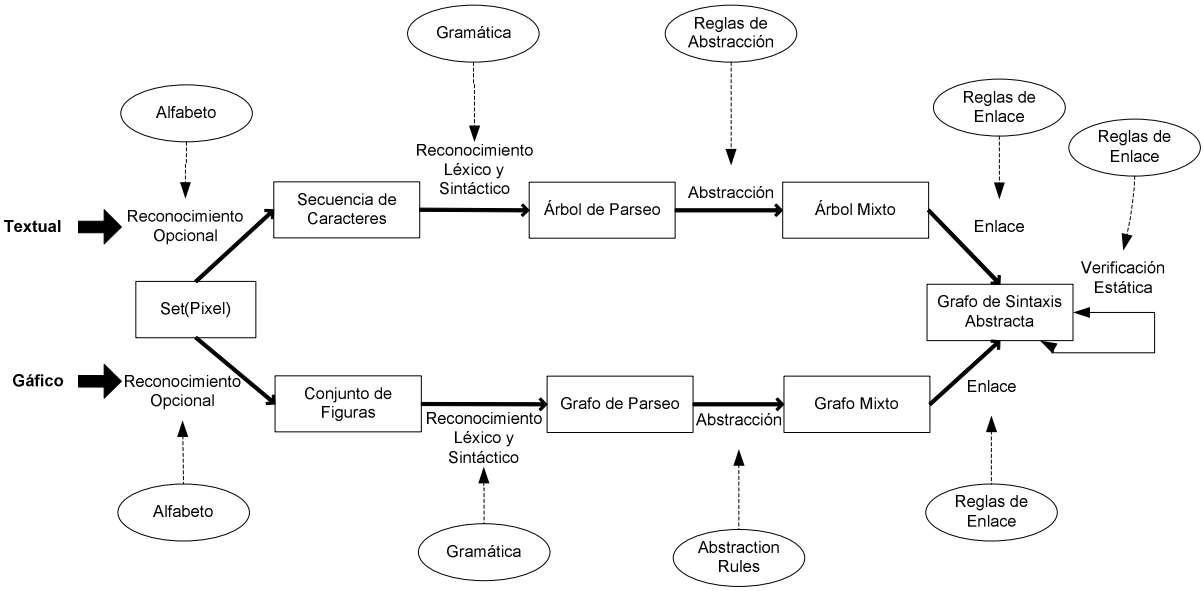


Figura 5-1 Proceso de reconocimiento para lenguajes textuales y gráficos

5.1.1. Reconocimiento óptico

El proceso de reconocimiento de lenguajes textuales y gráficos comienza con datos puros, los cuales consisten de un conjunto de pixeles, es decir un bitmap. En el caso textual OCR (optical character recognition) es utilizado para derivar a una secuencia de caracteres. Se debe tener en cuenta que OCR necesita conocimiento sobre los tipos de caracteres a ser reconocidos, así como la fuente y tipo de letra. Aunque es normal que se utilicen entradas

vía teclado, en algunos casos se deben considerar programas escritos en manuscrita.

En el caso gráfico, las herramientas de reconocimiento óptico no son muy utilizadas. Existen contados casos de uso para reconocer algunos tipos de lenguajes gráficos.

El resultado de esta fase es un conjunto de símbolos que incluye alguna forma de conocimiento acerca de su posición dentro del mismo. Esta información puede ser relativa a un punto fijo o depender de los otros. Por ejemplo, en el primer caso se podría decir que la ubicación del símbolo es x=30 e Y=28, mientras que para el segundo caso sería: la ubicación del símbolo es debajo del elemento X y a la izquierda del elemento Y.

Es común que los elementos de una entrada de caracteres se consideren como elementos con posición relativa, ya que cada uno está antes y después de otro carácter.

5.1.2. Análisis léxico y sintáctico

Las siguientes fases en un compilador son las de análisis léxico o scanning y análisis sintáctico o parsing. En la primera un grupo de caracteres son combinados para formar palabras. En la segunda, estas palabras se combinan para obtener un árbol de parseo.

Estas fases son muy similares y, aunque haya técnicas que las implementen de formas diferentes, el hecho de que el parsing se haga luego de hacer el scannig lo simplifica. Ejemplos de generadores de scanners y parsers para lenguajes textuales son Yacc y Lex, Antlr y javaCC.

Mientras que en el caso textual el resultado es un árbol de parseo, en el cado de los lenguajes gráficos el resultado debería ser un grafo de parseo. La mayoría de los nodos en este grafo son símbolos con posición, mientras otros representan conceptos más complejos. Todos los nodos se encuentran conectados según su posición o su grupo sintáctico.

5.2. Análisis semántico - Abstracción

Los grafos de parseo incluyen nodos que representan elementos puramente sintácticos. En lenguajes textuales lo son palabras claves y caracteres especiales, como por ejemplo los corchetes. En los lenguajes gráficos, lo son las flechas y rectángulos, que están presentes en el grafo de parseo. La primera acción de la fase de análisis es abstraerse de estos elementos puramente sintácticos para formar el grafo de sintaxis abstracta, o árbol de sintaxis abstracta para el caso de lenguajes textuales. Esta fase junto con el binding and checking se conoce como análisis de semántica estática.

5.3. Análisis semántico - Binding

El grafo de sintaxis abstracto no es aún el resultado requerido por procesamientos futuros tales como generación de código o transformación de modelos. Este grafo puede contener aún elementos sin ligar los cuales representen una referencia a otro elemento en el grafo que no está conectado. Estas referencias podrían apuntar a partes que podrían no existir por un error del usuario o a una especificación incompleta.

5.4. Análisis semántico - Chequeo estático

Finalmente algunos chequeos son realizados en el grafo de sintaxis abstracta. El mas conocido es el chequeo de tipos, el cual se puede realizar cuando todas las variables están ligadas. Otro chequeo que se debe realizar en esta fase es el análisis de control de flujo para reconocer código inalcanzable. Aunque ambos ejemplos se pueden encontrar en lenguajes textuales, no significa que en los lenguajes gráficos no se realicen verificaciones similares. Por ejemplo, de acuerdo a la especificación de UML la flecha implements debe ir solamente desde una clase a una interface. La mayoría de las herramientas UML no permiten a los usuarios dibujar esa flecha en otro caso. Pero, cuando el diagrama es importado desde otra fuente, es necesario realizar el chequeo.

5.5. Dos tipos de editores

En general existen dos tipos de editores los cuales pueden ser utilizados para lenguajes textuales y gráficos.

1 – Editor de formato libre, o editor orientado a símbolo, el cual permite al usuario ingresar símbolos de un alfabeto predefinido en cualquier posición.

2 – Editor estructurado, u orientado a sintaxis, el cual permite que el usuario ingrese solamente estructuras gramáticamente válidas.

Los editores de texto pertenecen a la primera categoría. Para los lenguajes gráficos, los editores los editores de formato libre se conocen como editores de gráficos vectoriales. Estos facilitan ubicar y manipular figuras simples (líneas, rectángulos y elipses) pero no soportan relaciones entre ellas. Esto dificulta la tarea de edición ya que al mover alguna figura se pierden los vínculos existentes con las otras.

Una de las formas más populares de edición orientada a sintaxis para lenguajes textuales es la que sugiere opciones para continuar el código. Estos editores utilizan conocimiento de la gramática y otra información para

ofrecer al usuario posibles formas para continuar en lo que está escribiendo. De esta manera el editor ayuda al usuario a escribir expresiones válidas, aunque sin restricciones ya que hay casos donde el usuario quiere escribir expresiones inválidas.

Es más común que para los lenguajes gráficos se utilicen editores estructurados, los cuales restringen al usuario en cuanto a lo que se puede programar. La mayoría de las herramientas CASE y otros ambientes de desarrollo gráfico implementan este tipo de editor, aunque algunos con más restricciones que otros. Por ejemplo, en algunas herramientas UML no se permite dibujar una relación entre clases pertenecientes a diferentes paquetes si no existe una relación de inclusión entre los mismos. En cambio otras lo permiten y presentan una advertencia. Por otro lado, existen algunas que ni si quiera lo advierten.

El uso de editores estructurados facilita el proceso de reconocimiento, ya que no es necesario parsear y abstraer el código ingresado. A veces tampoco es necesaria la fase de binding porque ya está incorporada en el editor cuando se le presentan opciones al usuario para crear elementos sin que haya otra alternativa.

5.6. Modelo de sintaxis concreta

La descripción de la sintaxis concreta debe incluir un alfabeto de símbolos junto con las reglas que indican como transformar esos elementos en un grafo de sintaxis abstracta. La descripción completa de esta sintaxis debe definir:

- Un alfabeto que provea los símbolos básicos
- Reglas de escaneo y parseo, las cuales establecen la forma de combinar esos símbolos para construir estructuras sintácticas.
- Reglas de abstracción que indican que elementos de la forma concreta no tienen su contraparte en la forma abstracta
- Reglas de binding que establecen como juntar o separar elementos de la sintaxis concreta, que representen los mismos o distintos elementos de la sintaxis abstracta.

Por último se podrían incluir reglas de chequeo, pero es mejor hacerlo en la descripción de la sintaxis abstracta. Esto permite que el editor que se implemente sea menos restrictivo.

Por ejemplo, la sintaxis concreta de una asociación de un diagrama UML está compuesta por una línea sólida, los nombres de los roles y sus cardinalidades, la navegabilidad, etc. Esto se corresponde con lo dicho en el primer ítem de la lista previa. Dentro del conjunto de reglas de escaneo y parseo, se podría especificar en que casos el fin de línea y sus adornos se

consideran juntos. Esto se podría hacer definiendo una distancia máxima entre ellos. Para el caso de reglas de abstracción, se podría especificar el color que se utilizará para dibujar las figuras, aunque esto es irrelevante. Por último, una regla de binding podría decir que, si dos elementos pertenecen al mismo paquete y tienen el mismo nombre, hacen referencia al mismo objeto en la sintaxis abstracta. Otra regla de este tipo podría decir que, si el tipo de un atributo tiene el mismo nombre que una clase existente, se hace referencia a esta.

5.7. Resumen

La sintaxis abstracta describe el vocabulario y la gramática del lenguaje, pero no define como se mostrará al usuario. Estas vistas son descriptas por la sintaxis concreta, y pueden ser en forma de diagrama o en forma de texto. En este capítulo se presentaron distintas maneras en las que se deriva la forma abstracta desde su representación concreta: por reconocimiento óptico, scanning y parsing, etc. Se mostraron los distintos tipos de editores, para lenguajes textuales y gráficos. Por último se explicó que elementos son necesarios para la definición de la sintaxis concreta.

En el siguiente capítulo se presentan plugins que permiten implementar una sintaxis concreta para un DSL.

6 Plugins para sintaxis concreta

En este capítulo se abordará un análisis sobre algunos de los plugins de Eclipse más conocidos para la generación de la sintaxis concreta, tanto en forma gráfica como textual.

6.1. Plugin para sintaxis gráfica

Dentro de lo que se refiere a la sintaxis concreta en forma gráfica, se trabajó con el framework GMF (Grafical Modeling Framework), y se hizo un análisis del plugin EuGENia.

6.1.1. Graphical Modeling Framework (GMF)

GMF [12],[13] es un framework de código abierto, desarrollado para el entorno Eclipse, que permite construir editores gráficos. Algunos ejemplos de estos editores son los editores de UML, de Ecore, de procesos de negocio y de flujo.

Los editores gráficos generados con GMF están completamente integrados a Eclipse y comparten las mismas características con otros editores, como vista overview, la posibilidad de exportar el diagrama como imagen, cambiar el color y la fuente de los elementos del diagrama, hacer zoom animado del diagrama e imprimirlo.

En la figura 6-1 pueden verse los principales componentes y modelos utilizados durante el desarrollo basado en GMF. El primer paso es la definición del metamodelo del dominio (archivo con extensión .ecore), el cual se define en términos del meta-metamodelo Ecore. A partir de allí, se derivan otros modelos necesarios para la generación del editor, como el modelo de definición gráfica y el modelo de definición de tooling. En el primero se especifican las figuras que pueden ser dibujadas en el editor. En el segundo, se define la información de los elementos en la paleta del editor, los menús, etc. Una vez definidos estos modelos, se combinan sus elementos a través de otro modelo que los relaciona. Es decir que, este último, relaciona los elementos del modelo de dominio con representaciones del modelo gráfico y elementos del modelo de tooling. Estas relaciones definen el modelo de mapping (archivo con extensión gmfmap).

Por último, una vez definidos todos los modelos y relacionados a través del archivo mapping, GMF provee un generador de modelo que permite definir todos los detalles de implementación anteriores a la fase de generación de código (archivo gmfgen). Esta generación de código produce

un plugin editor que interrelaciona la notación con el modelo de dominio. Además provee persistencia y sincronización de ambos.

6.1.2. Pasos para definir un editor gráfico

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

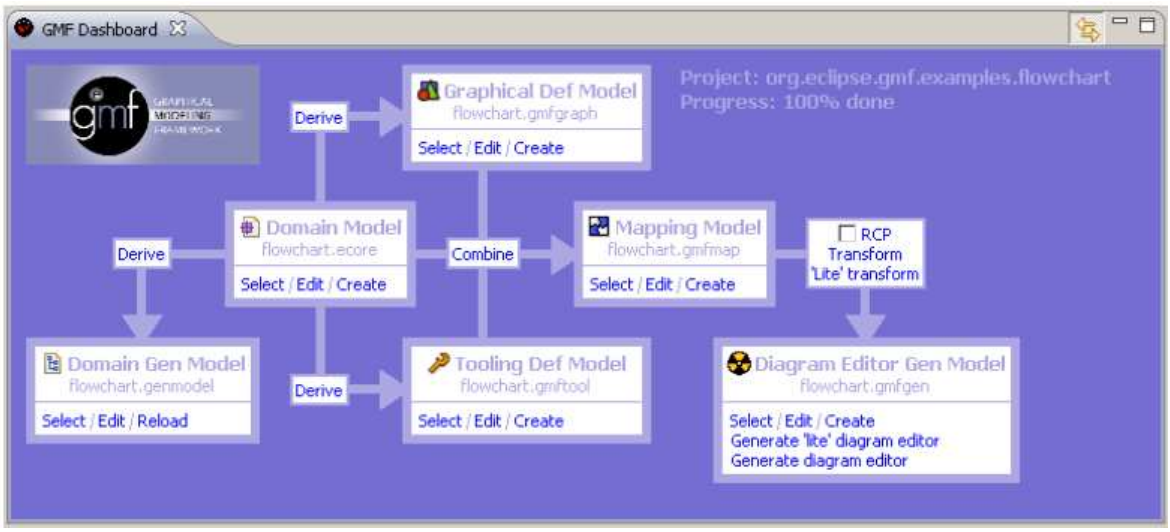


Figura 6-1 Componentes y modelos en GMF

Modelo de dominio

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

Modelo de definición gráfica

El modelo de definición gráfica se usa para definir figuras, nodos, links, compartimientos y demás elementos que se mostrarán en el diagrama. Este modelo está definido en un archivo con extensión ".gmfgraph". En la figura 6-2 puede verse el editor.

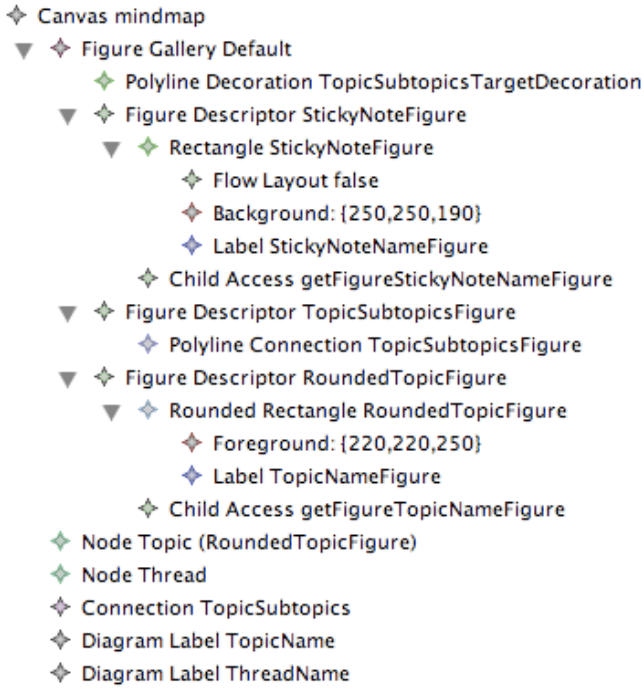


Figura 6-2 Editor del modelo de definición gráfica

Definición de herramientas

El modelo de definición de herramientas se usa para especificar la paleta, las herramientas de creación, las acciones que se desencadenan detrás de un elemento de la paleta, las imágenes de los botones, etc. En la figura 6-3 puede verse el editor para la creación de este modelo.

Figura 6-3 Editor del modelo de definición de herramientas

Definición de las relaciones entre los elementos

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

En la figura 6-4 puede verse el editor para este modelo. Es en este lugar donde se especifica: que elemento del metamodelo va a ser instanciado, con que herramienta o con que entrada de la paleta, y que vista le corresponde en el editor.

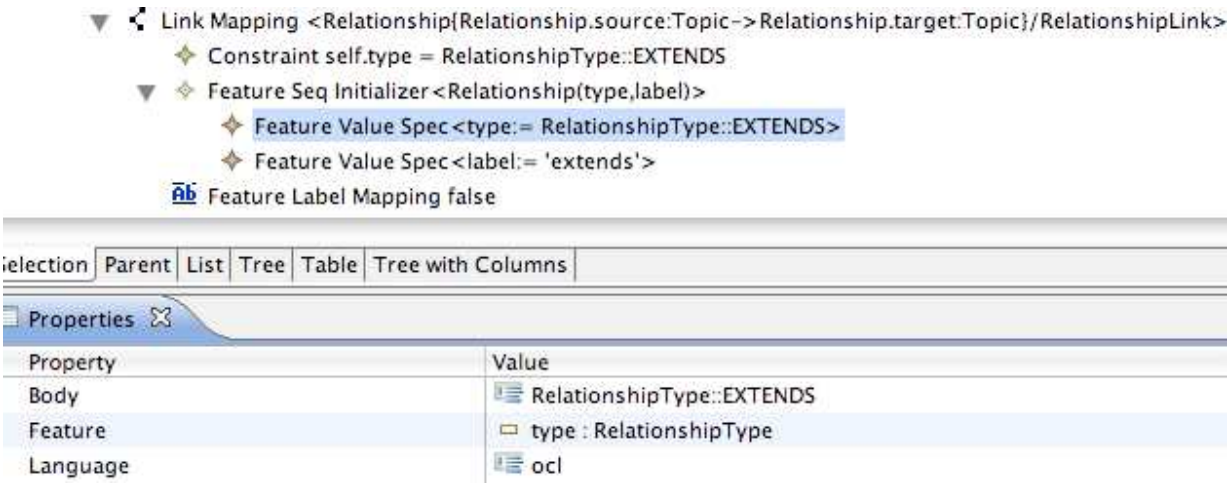


Figura 6-4 Editor del modelo de definición de relaciones

Generación de código para el editor gráfico

Una vez que se definieron los elementos gráficos y de mapeo, resta la generación del código. Para llevar a cabo este paso, primero se debe crear el modelo generador para especificar las propiedades inherentes a la generación de código. Este modelo se define en un archivo muy similar al del genmodel definido en EMF. Este paso se realiza en forma automática por medio del menú contextual, sobre el archivo de mapeo, con la opción "Create generator model...".

El modelo generador permite configurar los últimos detalles antes de la generación de código. Por ejemplo, permite hacer los siguientes ajustes:

- Cambiar las extensiones del diagrama y del dominio, que por defecto es el nombre del metamodelo del lenguaje.
- Cambiar el nombre del paquete para el plugin que se quiere generar.
- Configurar si el diagrama se guardará junto con la información del dominio en un único archivo.

Vista del editor gráfico

En la figura 6-5 puede verse el editor gráfico cuyo código es completamente generado por GMF. A la izquierda se encuentra el explorador de paquetes, que muestra los recursos que se encuentran en cada proyecto. Por defecto, cada modelo creado por el editor está formado por dos archivos: uno conteniendo las instancias del metamodelo, y otro conteniendo la información gráfica. Ambos deben mantenerse sincronizados para tener un modelo consistente. A la derecha se encuentra el outline, que es una vista con escala reducida que permite ver la totalidad del diagrama. En la parte inferior de la pantalla se ve la vista de propiedades, desde la cual se pueden modificar las propiedades de los elementos. La parte central de la pantalla muestra el editor, con su paleta de elementos. Como se dijo anteriormente, este editor está integrado al entorno Eclipse, ya que muchas de las

funcionalidades propias están disponibles por medio de menús contextuales, y también desde la barra de herramientas de Eclipse. Por ejemplo el zoom, tipos de fuente, etc.

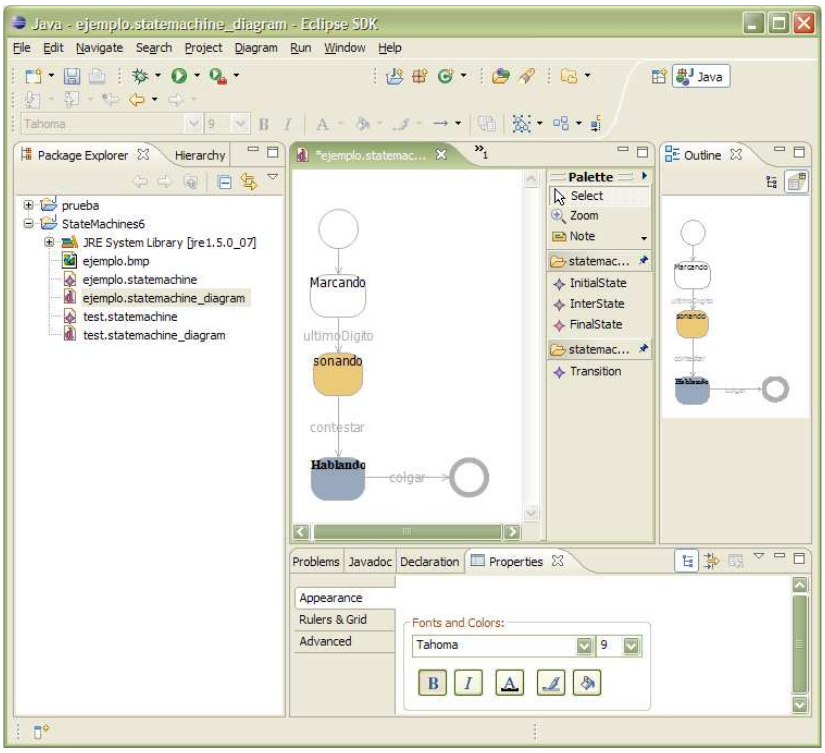


Figura 6-5 Vista del editor gráfico

6.1.3. EuGENia

Como se vio en la sección anterior, GMF es un plugin para generar editores visuales. Estos editores están basados en tres modelos, el gfmgraph, gmftool y gmfmap, cada uno con su propia sintaxis, lo que requiere un manejo adecuado de los mismos.

EuGENia es un plugin que se construye sobre GMF. A partir de un modelo Ecore se generan automáticamente los modelos de los archivos gmfgraph, gmftool y gmfmap. Este modelo Ecore debe estar marcado con anotaciones especiales definidas por EuGENia, lo que hace la construcción de esos tres modelos mucho mas fácil.

Ejemplos de esas anotaciones son:

- @gmf.diagram, el cual debe marcar el elemento raíz del modelo. A esta anotación se pueden agregar atributos como *onefile=true*, que especifica que el diagrama estará guardado junto con el modelo en un solo archivo, o *model.extension*, que indica la extensión del archive del modelo.
- @gmf.node, que indica que el elemento aparecerá en los diagramas como un nodo. Esta anotación acepta los siguientes detalles:
    - label: debe indicar el nombre del atributo del elemento que aparecerá como label del nodo.
    - Figure: indica la figura que representa al elemento. Puede ser un rectángulo, una elipse, un rectángulo con las puntas redondeadas, o la clase Java que implementa la figura.
- @gmf.link, indica que el elemento aparecerá como un link en los diagramas. A esta anotación se le pueden agregar detalles como source, target, sourceDecoration y targetDecoration. Los valores que pueden tomar los dos últimos son none, flecha, rombo.

6.2. Plugin para sintaxis textual

Además de las facilidades para crear editores visuales, en Eclipse existen plugins que permiten definir lenguajes textuales. A continuación se listan algunos de ellos.

6.2.1. EMFText

Desarrollado por un equipo de la Universidad Técnica de Dresden, EMFText [8] es una herramienta que permite a los usuarios definir una sintaxis textual para un modelo Ecore. Este plugin genera los componentes para cargar, editar y guardar instancias del modelo. La fuente para la generación es un archivo con extensión .cs, el cual contiene la especificación

de la sintaxis concreta del lenguaje. Combinando esta información con el metamodelo, EMFText deriva una gramática libre de contexto y la exporta como una especificación de un parser, la cual contiene las acciones semánticas que cubren la instanciación del metamodelo. En la figura 6-6 se ve gráficamente esta funcionalidad.

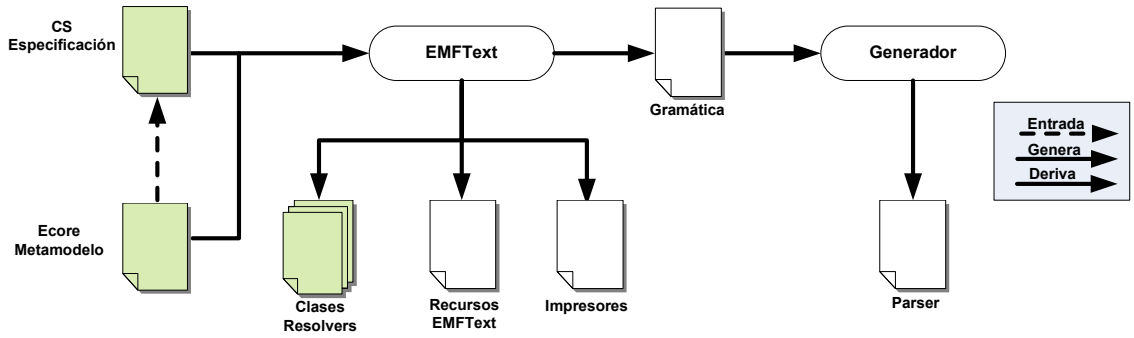


Figura 6-6 Componentes generados por EMFText

Dentro de las facilidades que tiene el editor mencionado, se pueden indicar colores para los diferentes tokens, y soporta sugerencias para completar el código, con lo cual el editor propone palabras para utilizar como siguiente token.

Mientras que el parser es utilizado para cargar las instancias del modelo desde representaciones textuales, un componente de impresión es necesario para hacer el camino inverso. Esto es, ir de una instancia del metamodelo en memoria a su representación textual. El resultado impreso debería también ser reconocido por el parser para obtener una instancia igual a la original. A su vez, la impresión debería ser generada ya formateada, de manera que facilite su lectura. Para lograr esto, dentro del archivo .cs se pueden agregar caracteres especiales para indicar en que posición hay que espacios en blanco o retornos de carro para lograr una buena impresión.

EMFText también genera un conjunto de resolvers, los cuales convierten los tokens de la gramática a su contraparte en el metamodelo y viceversa. Aunque la implementación generada generalmente cubre las conversiones necesarias, permite que el usuario la modifique para adecuarla a su metamodelo. Y estos cambios permanecerán en las siguientes generaciones, ya que estas clases modificadas no son sobrescritas.

6.2.2.Xtext

Xtext es el componente principal del proyecto TMF de Eclipse el cual permite desarrollar una gramática de un DSL usando un lenguaje similar a BNF. Permite generar un metamodelo basado en Ecore, un editor de texto para Eclipse y su correspondiente parser. Desafortunadamente, la intención de Xtext es comenzar con la gramática y producir el modelo Ecore, en lugar

de, a partir de un modelo existe derivar una gramática que lo soporte. Sin embargo, permite realizar transformaciones desde y hacia modelos Ecore, con lo cual se maneja interoperabilidad con otras herramientas basadas en EMF, como pueden ser QVT y MOFScript. Y si es necesario, permite en la especificación de la gramática usar un metamodelo existente por medio de un mecanismo de importación.

Provee facilidades para trabajar con el lenguaje generado. Crea un editor de texto completamente funcional, con resaltador de sintaxis, asistentes de código, vista de outline, etc. También registra un ResourceFactory de EMF para la extensión asociada a los archivos del DLS generado, lo que permite leer (pero no escribir) instancias del modelo.

Actualmente el proyecto se encuentra en etapa de incubación y la única versión disponible funciona solamente con la versión reciente de Eclipse llamada Galileo.

6.2.3.TCS

TCS es otro plugin que también permite definir un DSL con sintaxis textual. Su nombre corresponde a la sigla Textual Concrete Syntax y su primera versión fue definida durante el 2003 dentro del Inria para la especificación de la segunda versión de ATL. Desde mediados del 2007 es parte del proyecto Generative Modeling Technologies (GMT) de Eclipse. Al momento de escribir este capítulo, GMT es un proyecto de investigación que aún se encuentra en estado de incubación y no cuenta aún con versiones terminadas para instalar.

Las metas que intenta cumplir son similares a las que brinda EMFText. Provee:

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

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

6.3. Resumen

En este capítulo se presentaron algunos plugins para la definición de sintaxis concreta de un lenguaje. Se mencionó que existen dos formas de sintaxis concreta: gráfica y textual.

Para la definición en forma gráfica se presentaron GMF y Eugenia. Para la definición en forma textual se presentaron EMFText, TCS y XText.

El último paso en la definición de un DSL implica definir su semántica. Esta describe que sucede en una computadora cuando una instancia del mismo es ejecutada. En el capítulo siguiente profundiza sobre este tema.

7 Semántica

El propósito de este capítulo es describir como se puede usar el metamodelado para describir la semántica de los lenguajes de modelado o los de programación. La semántica de un lenguaje de modelado describe el significado del mismo en términos de su comportamiento, sus propiedades estáticas o su traducción a otro lenguaje.

El capítulo comienza con la motivación de por que es necesaria la semántica de un lenguaje, y luego introduce algunas propuestas de las descripciones de diferentes semánticas usando metamodelado. Estas propuestas intentan proveer una forma de construir modelos con semánticas independientes de plataforma, haciendo que la semántica de un lenguaje sea intercambiable entre herramientas de metamodelado.

¿Qué es la semántica de un lenguaje?

En términos generales, la semántica de un lenguaje describe el significado de los conceptos del mismo. La semántica del lenguaje describe que sucede en una computadora cuando una instancia del mismo es ejecutada. Es por eso que cuando se utiliza un lenguaje, es necesario asignar un significado a sus conceptos para lograr entender como usarlo. Por ejemplo, en el contexto de un lenguaje de modelado, la definición de lo que es una máquina de estados o una clase, es una parte clave al modelar un aspecto específico en cierto dominio.

Existen varias maneras de describir el significado de un concepto de lenguaje. Por ejemplo, se pueden definir en términos de otros conceptos a los cuales se les dio un significado previamente, describiendo sus propiedades y comportamiento o especializando otro concepto existente. De la misma manera se podría hacer describiendo las propiedades que comparten todas las posibles instancias del concepto, o indicando una relación o mapeo entre los conceptos de un lenguaje con pensamientos y experiencias de conceptos del mundo real.

7.1. Objetivo de la semántica

Una semántica es esencial para comunicar el significado de los modelos o programas, entre los miembros del equipo de desarrollo de un sistema. La semántica tiene un rol central ya que define las capacidades del lenguaje como su ejecución, análisis y transformación. Por ejemplo, un lenguaje que soporta comportamiento como las maquinas de estado, requiere una semántica para describir como se van a ejecutar los programas o modelos escritos en el mismo.

Tradicionalmente la semántica de muchos lenguajes de modelado era escrita de manera informal, algunas veces con lenguaje natural, y otras con ejemplos. Por ejemplo, gran parte de la especificación de UML, en las versiones 1.x, utilizan el lenguaje natural para las descripciones semánticas.

Una semántica informal trae consigo varios de los siguientes problemas:

- Debido a que los usuarios tienen que asignar un significado informal o intuitivo a los modelos, hay un riesgo importante de mala interpretación y, por consiguiente, puede llevar a que los usuarios hagan un mal uso del lenguaje.
- Una semántica informal no puede ser interpretada o entendida por las herramientas. Los desarrolladores de herramientas deben implementar entonces su propia interpretación de la semántica. Desafortunadamente, esto significa que el mismo lenguaje puede ser implementado de diferentes maneras. De esta forma, dos herramientas distintas pueden ofrecer implementaciones contradictorias de la misma semántica. Por ejemplo, la misma máquina de estados puede ejecutar de manera diferente según la herramienta que se este utilizando.
- Una semántica informal hace mas complicada la tarea de definir lenguajes nuevos. Dificulta identificar áreas en las cuales los conceptos son semánticamente equivalentes, o donde hay contradicciones. Además hace más complicado extender un lenguaje existente, sobre todo si la semántica del mismo no está bien definida.
- Los estándares requieren una semántica definida de forma precisa, ya que sin ella puede ser mal interpretado.

7.2. Semántica y metamodelos

Mientras está claro que la semántica es una parte crucial en la definición de un lenguaje, no queda muy en claro como se la debe describir. Una propuesta es expresar la semántica en términos de un lenguaje matemático formal. En este sentido, existen muchas publicaciones describiendo la semántica de lenguajes de modelado. Pero la complejidad de estas descripciones matemáticas resulta difícil de entender y tienen un uso práctico limitado. Una segunda propuesta es expresarla en términos de un lenguaje de programación externo. Esta es una propuesta mucho más práctica, pero aun así el resultado es específico a un lenguaje de programación, lo que compromete la característica de independencia de plataforma. Y mas aún, el hecho que se describa en un lenguaje de programación, hace que el proceso de definición no sea intuitivo.

Una estrategia alternativa es describir la semántica usando metamodelos, lo cual brinda algunos beneficios importantes. Primero, la semántica del lenguaje está completamente integrada a la definición del mismo. Esto significa que está incluida dentro de los otros elementos del lenguaje, como la sintaxis concreta, mappings, sintaxis abstracta, etc. Segundo, como se usa el mismo lenguaje de metamodelado para las definiciones de todos los

lenguajes, las definiciones semánticas se convierten en aserciones reusables que pueden ser integradas y extendidas con relativa facilidad. Finalmente y lo más importante es que las definiciones semánticas son independientes de plataforma, se pueden intercambiar y, si son entendidas por la herramienta que las importa, las puede usar para conducir la forma en que esta interactúa con el lenguaje. Por ejemplo, una semántica que describe como se ejecuta una máquina de estados puede ser usada para guiar simuladores a través de un conjunto de herramientas.

Es importante notar que un metamodelo de la semántica es bastante diferente del modelo de sintaxis abstracta de un lenguaje, el cual define la estructura del mismo. Sin embargo, un modelo de sintaxis abstracta es pre requisito para la definición de la semántica, ya que la semántica agrega una capa al significado de los conceptos definidos por la sintaxis abstracta. La semántica en este sentido debería ser distinguida de la semántica abstracta, la cual expresa las reglas que indican si una expresión del lenguaje esta bien formada. Las reglas de la semántica estática son usadas por las herramientas como verificadores de tipos.

7.3. Propuestas

Existen varias propuestas diferentes para describir la semántica de un lenguaje en un metamodelo. En esta sección se examinan las propuestas más importantes y se enumeran ejemplos de su aplicación. Todas estas propuestas fueron motivadas por otras para definir la semántica que han sido ampliamente aplicadas en los dominios de los lenguajes de programación. La diferencia principal es que se usan metamodelos para expresar las definiciones semánticas.

Estas propuestas incluyen:

- Por traducción: traducir los conceptos de un lenguaje en conceptos de otro lenguaje que tiene semántica precisa.
- Operacional: modelar el comportamiento operacional de los conceptos del lenguaje. Es decir, describiendo como se interpreta un concepto como secuencias de pasos computacionales. Esto frecuentemente se representa como un sistema de transición de estados que muestra como progresa el sistema a través de los estados.
- Extensional: extender la semántica de conceptos de lenguajes existentes.
- Denotacional: construyendo objetos matemáticos, llamados denotaciones, que representan el significado del programa.

Cada propuesta tiene sus ventajas y desventajas. En la práctica, se usa una combinación de estas basadas en la naturaleza de los conceptos del lenguaje. En cada caso, es importante notar que la semántica es descripta en el lenguaje de metamodelado, es decir que no se usa ninguna representación externa formal. Las siguientes secciones describen cada una de estas propuestas con un pequeño ejemplo de su aplicación.

7.3.1. Semántica definida por traducción

La semántica definida por traducción se basa en dos nociones:

- La semántica de un lenguaje se define cuando se traduce el lenguaje en otra forma, llamado lenguaje destino.
- El lenguaje destino puede ser definido por un pequeño número de construcciones primitivas que tienen una semántica bien definida.

La intención de esta propuesta por traducción es definir el significado de un lenguaje en términos de conceptos primitivos que sí tienen su propia semántica bien definida. Comúnmente esos conceptos primitivos tienen una semántica operacional.

La ventaja de esta propuesta es que si existe una maquinaria que ejecute el lenguaje destino, es posible obtener una semántica ejecutable para el lenguaje vía la traducción. Esta propuesta esta muy relacionada al rol jugado por un compilador cuando implementa un lenguaje de programación más rico en términos de primitivas ejecutables.

La principal desventaja de esta propuesta es que la información se pierde durante el proceso de transformación. Cuando el resultado es una colección de primitivas, no resulta obvio como estaban relacionadas con los conceptos originales. Hay maneras para evitar esto, por ejemplo marcar los conceptos originales. O mantener información acerca del mapeo entre los dos modelos.

Esta propuesta por traducción puede ser incorporada a la definición del lenguaje de diferentes maneras:

- Dentro de un metamodelo del lenguaje, traduciendo un concepto en otro que tiene semántica, por ejemplo, en el caso de una sentencia case en una secuencia de sentencias if, donde las sentencias if tienen una semántica operacional.
- Entre metamodelos de lenguajes, traduciendo un metamodelo en otro. Por ejemplo, un metamodelo de sintaxis abstracta de UML puede ser mapeado en un metamodelo de un lenguaje pequeño, bien definido como XCore, o a un lenguaje de programación como Java.

7.3.2. Semántica operacional

Una semántica operacional describe como se pueden ejecutar los modelos o los programas escritos en un lenguaje. Esto involucra la construcción de un intérprete. Por ejemplo, una sentencia de asignación V:=E se puede describir por un intérprete que ejecuta los pasos que implica evaluar la expresión E y luego cargar el valor en la variable V.

La ventaja de una semántica operacional es que puede ser expresada en términos de operaciones sobre el lenguaje mismo. En contraste con la semántica definida por traducción que se define en términos de otro lenguaje, posiblemente distinto. Como resultado, una semántica operacional puede ser fácil de entender y escribir.

Escribir un intérprete como parte de un metamodelo depende de si lenguaje de metamodelado es ejecutable.

Comúnmente la definición de un intérprete para un lenguaje sigue un patrón en el cual los conceptos se asocian a una descripción operacional de la siguiente forma:

- Se definen operaciones sobre conceptos que implementan su semántica operacional, por ejemplo, una acción tiene una operación *run()* que causa un cambio de estado, mientras que una expresión puede tener una operación *eval()* que evalúa la expresión
- Las operaciones tienen un ambiente como parámetro el cual es una colección de ligaduras a variables que serán usadas para la evaluación del comportamiento de los conceptos y el objeto resultado, que es el objeto que cambia como resultado de la acción o que representa el contexto de evaluación.
- Las operaciones retornan un resultado de la evaluación: un booleano en el caso de una expresión estática, o cambian el valor del objeto destino, en el caso de una acción.

7.3.3. Semántica extensional

En la propuesta extensional, la semántica de un lenguaje se define como una extensión de otro lenguaje. Los conceptos de modelado del nuevo lenguaje heredan la semántica de los conceptos en el otro lenguaje. Además, pueden extender la semántica, por ejemplo agregando nuevas capacidades.

Las ventajas de esta propuesta es que los conceptos semánticos complejos pueden ser reusados con un mínimo esfuerzo. Por ejemplo, una entidad de negocios no necesita definir que significa crear nuevas instancias, ya que puede heredarlo de Class.

La propuesta extensional tiene algunos puntos en común con la noción de perfil de UML. Un perfil provee una colección de estereotipos que pueden ser vistos como subclases de los elementos de UML o de MOF. Sin embargo, usando la extensión en el nivel de metamodelado, se provee un mayor poder expresivo, ya que puede arbitrariamente agregar extensiones semánticas a los nuevos conceptos. Una herramienta podría hacer uso de esta información para permitir una implementación rápida de un nuevo lenguaje de modelado, ya que podría reconocer que ha ocurrido una extensión y usar los estereotipos para adaptar los símbolos del lenguaje de modelado original para soportar el nuevo lenguaje.

7.3.4. Semántica denotacional

El propósito de la semántica denotacional es asociar objetos matemáticos, como números, tuplas, o funciones con cada concepto del lenguaje. El concepto se dice que denota el objeto matemático, y el objeto se llama

denotación del concepto. Los objetos asociados con el concepto se llaman el dominio semántico del concepto.

Una semántica denotacional puede ser pensada como una semántica por ejemplos. Proveyendo todos los posibles ejemplos de los significados de los conceptos es posible definir de manera precisa que significan. Para describir la semántica de un Integer, se necesita el conjunto de todos los números positivos, es decir, la denotación de Integer es 0..infinito. Las descripciones denotacionales de la semántica tienden a ser estáticas, es decir, enumeran las instancias válidas de los conceptos de una manera no ejecutable.

A continuación se listan unos ejemplos comunes de las relaciones denotacionales encontradas en los metamodelos:

- La denotación de una Clase es la colección de todos los objetos que pueden ser instancia de la Clase.
- La denotación de una Action es una colección de todos los posibles cambios de estado que pueden resultar de su invocación.
- La denotación de una Expresión es la colección de todos los posibles resultados que se pueden obtener evaluando la expresión.

Una semántica denotacional se puede definir en un metamodelo construyendo un modelo de la sintaxis abstracta del lenguaje, un dominio semántico y el mapeo semántico que las relaciona. Las restricciones se escriben para describir cuando las instancias del dominio semántico son válidas con respecto a su sintaxis abstracta. Por ejemplo, las restricciones se pueden escribir para decir cuando un objeto es una instancia válida de una clase.

7.4. Plugins para definir semántica

Como se mencionó, existen varias formas de definir la semántica del lenguaje. En las siguientes secciones se tratarán dos implementaciones que permiten definir semántica por traducciones.

7.4.1. Java Emitter Template (JET)

Java Emitter Template (JET) es un subproyecto de Eclipse Modeling Framework (EMF) enfocado en la traducción de modelos a su representación textual.

JET es la herramienta que provee por defecto EMF para la generación código en forma automática. Requiere que se especifiquen plantillas, como entrada para el proceso de traducción, indicando el formato que tendrá el código generado. Estas plantillas se escriben con una sintaxis muy parecida a la de JSP, lo que hace que sean fáciles de comprender para el común de los desarrolladores Java. De esta manera, JET puede utilizarse para generar cualquier tipo de archivo de texto, como SQL, XML, código Java y cualquier otro tipo de representaciones textuales.

Un template de JET es un archivo cuya extensión termina en "jet". Por convención de EMF, la primer parte de la extensión del archivo corresponde al tipo del resultado de la traducción. Por ejemplo, el template correspondiente a la traducción al lenguaje Java tendrá extensión "javajet", mientras que el de la traducción a XML será "xmljet".

El código siguiente corresponde a un fragmento del template utilizado para la generación del código Java necesario para la ejecución de las validaciones de OCL ("ValidatorClass.javajet"):

```
public class <%=genPackage.getValidatorClassName()%> extends
<%=genModel.getImportedName("org.eclipse.emf.ecore.util.EObjectValidator
")%>
{
<%if (genModel.getCopyrightText() != null) {%>
      public static final
<%=genModel.getImportedName("java.lang.String")%> copyright =
"<%=genModel.getCopyrightText()%>";<%=genModel.getNonNLS()%>

<%}%>
      public static final <%=genPackage.getValidatorClassName()%>
INSTANCE = new <%=genPackage.getValidatorClassName()%>();

      public static final String DIAGNOSTIC_SOURCE =
"<%=genPackage.getInterfacePackageName()%>";<%=genModel.getNonNLS()%>

<%int count = 0; for (GenClass genClass : genPackage.getGenClasses())
{%>
<%for (GenOperation genOperation : genClass.getInvariantOperations())
{%>
      public static final int <%=genClass.getOperationID(genOperation)%>
= <%=++count%>;

<%}}%>
      private static final int GENERATED_DIAGNOSTIC_CODE_COUNT =
<%=count%>;

      protected static final int DIAGNOSTIC_CODE_COUNT =
GENERATED_DIAGNOSTIC_CODE_COUNT;

<%for (GenPackage baseGenPackage :
genPackage.getAllValidatorBaseGenPackages()) {%>
      protected <%=baseGenPackage.getImportedValidatorClassName()%>
<%=genPackage.getValidatorPackageUniqueSafeName(baseGenPackage)%>Validat
or;

<%}%>
```

A continuación se muestra parte del código generado por este template:

```
public class CmValidator extends EObjectValidator {
      public static final CmValidator INSTANCE = new CmValidator();

      public static final String DIAGNOSTIC_SOURCE = "cm";
```

```
	private static final int GENERATED_DIAGNOSTIC_CODE_COUNT = 0;

	protected static final int DIAGNOSTIC_CODE_COUNT =
GENERATED_DIAGNOSTIC_CODE_COUNT;

	private static final String OCL_ANNOTATION_SOURCE =
"http://www.eclipse.org/ocl/examples/OCL";

	private static final OCL OCL_ENV = OCL.newInstance();

public CmValidator() {
		super();
	}
```

En realidad, la generación de código con JET consta de dos pasos: traducción y generación. En el primero se traduce el template a una representación en código Java; y en el segundo se realiza la generación del archivo de texto utilizando la clase creada en el primer paso. Esto lo asemeja más a JSP, ya que no solo comparten la sintaxis, sino que también la forma en que trabajan.

7.4.2. MOFScript

La herramienta MOFScript [22] permite la transformación de cualquier modelo MOF a texto. Por ejemplo, permite la generación de código Java, EJB, JSP, C#, SQL Scripts, HTML o documentación a partir de los modelos. La herramienta está desarrollada como un plugin de Eclipse, el cual soporta el parseo, chequeo y ejecución de scripts escritos en MOFScript.

El lenguaje de transformación MOFScript es un lenguaje que fue enviado al pedido de propuestas de lenguajes de transformación de modelo a texto lanzado por el OMG, pero que no resultó seleccionado. MOFScript está basado en QVT, es un refinamiento del lenguaje operacional de QVT. Es un lenguaje textual, basado en objetos y usa OCL para la navegación de los elementos del metamodelo de entrada. Además, presenta algunas características avanzadas, como la jerarquía de transformaciones y mecanismos de rastreo.

Algunas características de MOFScript son:

- El lenguaje permite especificar mecanismos de control básicos como iteraciones y sentencias condicionales.

- El lenguaje permite manipulación de strings con operaciones básicas.

- La herramienta tienen la habilidad de generar texto a partir de cualquier modelo basado en un metamodelo definido en MOF, como por ejemplo, modelos UML o cualquier otro metamodelo.

- La herramienta permite especificar el archivo salida para la generación del texto.

- La herramienta mantiene la trazabilidad entre los modelos y el texto generado.

- El editor de scripts posee ayuda para completar el código.

- La ingeniería inversa todavía no es parte de la herramienta.

La herramienta asume que la extensión ".m2t" refiere a archivos de MOFScript. Estos archivos pueden ser ubicados dentro del Eclipse para que luego puedan ser compilados y ejecutados.

7.5. Resumen

Durante este capítulo se explicó la importancia de la semántica de un lenguaje, y como se puede utilizar el metamodelo para describirla.

En el pasado, la semántica de los lenguajes de modelado era escrita de manera informal, mediante el uso de lenguaje natural o con ejemplos. Esto generaba ambigüedades y originaba problemas en su interpretación. Para evitarlos surgió la necesidad de definir la semántica de una manera precisa.

Una propuesta para esto es escribir la semántica utilizando metamodelos. De esta manera estará integrada a la definición del lenguaje. Como las definiciones semánticas son independientes de plataforma, se pueden intercambiar y, si son entendidas por la herramienta que las importa, se pueden usar para conducir la forma en que esta interactúa con el lenguaje.

Para describir la semántica de un lenguaje en un metamodelo se presentaron las siguientes alternativas: definida por traducción, en forma operacional, extensional y denotacional.

Adicionalmente se presentaron dos plugins para definir la semántica por traducción. Se hizo un breve análisis sobre JET, uno de los plugins que provee EMF para la traducción de modelos a una representación textual. Esta traducción se realiza especificando una plantilla escrita en un lenguaje muy parecido a JSP.

Otro plugin que se vio para realizar este tipo de traducciones fue MOFScript. En este caso, la forma en que se debe realizar la traducción se especifica en un archivo con un lenguaje textual basado en objetos, que usa OCL para la navegación de los elementos del metamodelo de entrada.

Como una forma de mostrar en forma integral los conceptos definidos durante todo este trabajo, en el capítulo siguiente se introduce el caso de estudio.

8 Caso de Estudio

En este capítulo se aplicarán los conceptos visto en forma integral para construir un DLS en particular. Para esto se eligió un lenguaje específico para modelar sistemas colaborativos. La definición completa de este lenguaje se encuentra en el trabajo [1].

Siguiendo los pasos descriptos en el capítulo 3, se listan los conceptos candidatos dentro del dominio, para luego definir la sintaxis abstracta. A continuación, se la implementa utilizando los plugins vistos en el capítulo 4. A partir de allí se definen las sintaxis concretas, una en forma gráfica y otra textual. Luego se las implementa usando los plugins explicados en el capítulo 6. Por último, se muestra un plugin con el cual se podría implementar la semántica del lenguaje.

8.1. Introducción al lenguaje CM (*Collaborative Modeling*)

Los sistemas colaborativos [5] son aquellos tipos de aplicaciones, para grupos u organizaciones, que surgen de la unión de computadoras, de grandes bases de información y de tecnologías de comunicación. Es una tecnología diseñada para facilitar el trabajo en grupo (groupware en inglés) y se puede usar para comunicar, cooperar, coordinar, resolver problemas, competir o negociar. La definición más común de groupware los describe como sistemas basados en computadoras que soportan grupos de personas involucradas en una tarea común (u objetivo) y que proveen de una interfaz a un ambiente compartido.

Las nociones de una "tarea común" y "ambiente compartido" son cruciales en esta definición. A partir de ellas, se excluyen, por ejemplo, a los sistemas multiusuarios, donde los usuarios comparten algunos recursos (como espacio en disco, tiempo de procesamiento, memoria, etc) pero no comparten en realidad una tarea común.

En un primer acercamiento a los sistemas groupware se pueden distinguir, por un lado, las aplicaciones donde los usuarios realizan actividades en forma simultánea (sincrónica) llamadas "real time groupware". Y por otro lado están aquellas en las que los usuarios realizan sus actividades en forma asincrónica y se denominan "non real time groupware".

A continuación se enumeran los diferentes conceptos que se pueden encontrar dentro de las aplicaciones colaborativas.

Usuario: Representa a un usuario dentro del sistema el cual cuenta con un conjunto de propiedades que lo identifican. Por ejemplo un nombre, un apodo, su imagen u otros datos que lo distingan.

Rol: Agrupa un conjunto de propiedades, conocimientos y responsabilidades que tendrá un usuario en un determinado momento. Los

usuarios podrán cambiar el rol dinámicamente y este cambio será manejado por los protocolos de colaboración. Por ejemplo, en algunas sesiones un usuario podrá actuar como "*mediador"*, en algunas situaciones o como "*participante"* en otras.

Objetos colaborativos: Son los elementos creados por los propios usuarios que se pueden manipular en forma colaborativa. Por ejemplo, en un ambiente de educación a distancia, un alumno puede editar elementos colocados por un compañero o el tutor. La particularidad de estos objetos es que son editados por distintos usuarios y dependiendo de la herramienta y del protocolo la edición puede ser en forma sincrónica o asincrónica. Los objetos colaborativos más comunes en los ambientes groupware son documentos de texto, dibujos, etc.

Espacio de trabajo: Es el lugar en que se lleva a cabo la colaboración. El espacio de trabajo define las actividades que se realizarán, los roles involucrados y las herramientas a utilizar. Un ejemplo de esto es un aula virtual donde se realizan actividades de enseñanza. Estas pueden ser clases al estilo tradicional o de consulta. En las clases tradicionales se usarán herramientas de video conferencia y foros, mientras que en las de consultas se utilizarán herramientas para comunicación vía mail o chat. Los roles que intervendrán serán: docente, tutor y alumno. Finalmente podemos describir protocolos diferentes para la clase y para la consulta. En la clase la maestra podría coordinar la participación de los alumnos, dándoles la palabra en el caso que ellos la soliciten y en las consultas, los tutores podrían tener las mismas atribuciones que los alumnos al momento de participar en la colaboración.

Sesión: Una sesión es un período de interacción soportada por un sistema. En general un usuario ingresa a la misma identificándose a través de un nombre de usuario y contraseña, e indica su liberación en forma explícita, o simplemente cierra la aplicación que maneja la sesión. La administración de la sesión, entre otras cosas, sirve para llevar un registro de las actividades de los usuarios a lo largo de la misma.

Herramientas colaborativas: Son programas que, a diferencia de las aplicaciones monousuarios, contemplan ser utilizados por un grupo de personas. Naturalmente, el grupo manipula elementos que llamaremos objetos colaborativos que son los productos que se obtienen como resultado de la colaboración. Dentro del conjunto de herramientas colaborativas se pueden mencionar: chat, pizarrón compartido y editores colaborativos. Estas herramientas pueden ser utilizadas sin importar el protocolo elegido. Por ejemplo, el Chat podría ser usado de forma que los usuarios puedan enviar mensajes en forma sincrónica o asincrónica.

Escenario Colaborativo: Es la integración de un conjunto de espacios de trabajo. Frecuentemente las aplicaciones colaborativas complejas necesitan más de un espacio de trabajo. Por ejemplo, una universidad virtual estará compuesta por distintos espacios de trabajo, como el aula, la biblioteca, etc. Los escenarios contienen los protocolos que estructuran el acceso y el uso, por parte de los roles, de los diferentes espacios de trabajo. Esto es porque

no todos los roles tendrán acceso a todos los espacios de trabajo. De la misma manera algunos roles se convertirán a otro al cambiar el espacio de trabajo. Los escenarios colaborativos son un caso particular de Workspace ya que puede contener otros de manera tal que combinados generen uno más complejo.

Asociación colaborativa: Permite relacionar elementos del diseño. Con esta asociación se vinculan las herramientas que existen en un espacio de trabajo, los roles que pueden participar de una sesión, etc. Se pueden distinguir distintos tipos de asociaciones para representar las diferentes maneras en que se pueden relacionar los elementos. Por ejemplo para indicar la ubicación de objetos y roles dentro de los espacios, para indicar el uso por parte de los usuarios de los objetos colaborativos, para modelar la participación de los usuarios en las sesiones y para modelar la relación de las sesiones con los espacios.

8.2. Definición de la sintaxis abstracta

A continuación se mostrarán los diagramas de clases correspondientes para la implementación del lenguaje que permite modelar ambientes colaborativos. Así mismo se incluirá una descripción asociada a cada entidad definida.

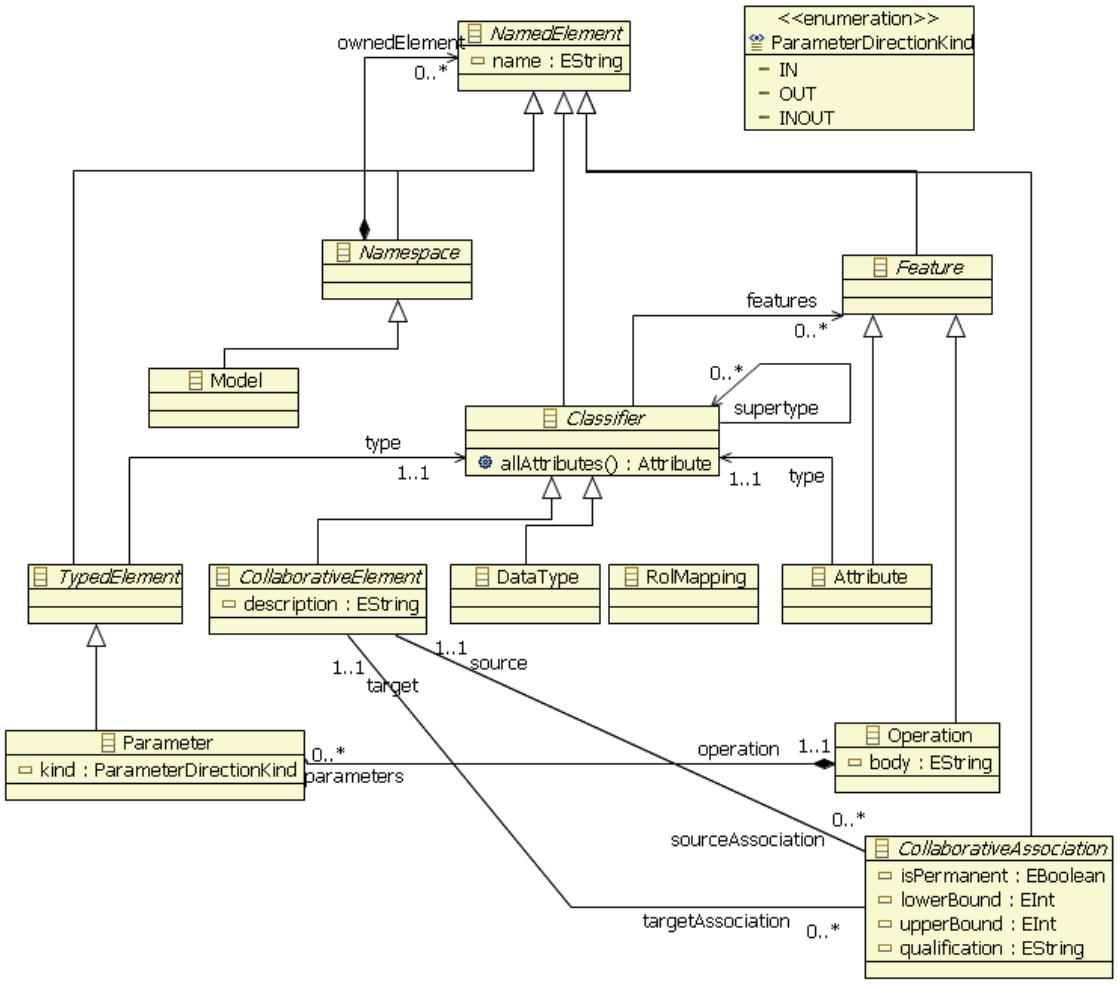


Figura 8-1 Diagrama Core

En el primer diagrama de la figura 8-1 se muestran las clases generales que proveen el soporte para las entidades del dominio. A continuación se detallan cada una de ellas.

Attribute
Descripción:
Un Attribute representa a un atributo de una clase

Generalizaciones:
Feature

Atributos:

- Type: referencia al tipo del atributo.

Classifier

Descripción:
Un classifier es una clasificación de instancias, las cuales tienen características en común. Un classifier es un espacio de nombres o namespace cuyos miembros pueden incluir features. Classifier es una metaclase abstracta.

Generalizaciones:
NamedElement

Atributos:

- features: Feature [*]. Especifica cada feature definido en el classifier, los cuales pueden ser atributos u operaciones.

Constraints:

[1] No pueden definirse atributos con el mismo nombre dentro de un Classifier

```
inv distinctAttributesName:
self.features -> select (a | a.oclIsKindOf(Attribute)) ->
forAll (p, q | p.name = q.name implies p = q)
```

Collaborative Association

Descripción:

Una asociación colaborativa es una clase abstracta que generaliza cualquier asociación o relación que forma parte de un modelo colaborativo. Es una conexión estructural simple entre instancias del sistema colaborativo. Declara que pueden existir links entre instancias de los tipos asociados. Un link es una tupla con un valor para cada final de asociación, donde el valor es una instancia del tipo declarado por el final de asociación.

Generalizaciones:
NamedElement

Atributos:

- isPermanent: es un booleano que indica si la conexión es permanente. El valor por defecto es false.
- lowerBound: es un número que indica la cardinalidad menor de la asociación.
- upperBound: es un número que indica la cardinalidad mayor de la asociación.

Constraints:

[1] El valor de lowerBound debe ser menor o igual a upperBound, a menos que alguno de estos tenga cardinalidad a muchos, lo cual se indica con el valor -1.

```
inv distinctAttributesName:
   self.lowerBound = -1 or self.upperBound = -1 or
(self.lowerBound <= self.upperBound)
```

Collaborative Element

Descripción:
Un elemento colaborativo es una clase abstracta que generaliza cualquier elemento que forma parte de un modelo colaborativo – describe un conjunto de instancias que tienen características comunes.

Generalizaciones:
Classifier

Atributos:

- description: es un string que indica la descripción de la clasificación

Datatype

Descripción:
Datatype actúa como una clase común de distintos tipos de datos. Representa la noción general de ser un tipo de datos, es decir, un tipo cuyas instancias son identificadas solo por su valor.

Generalizaciones:
Classifier

Atributos:

- No define atributos adicionales.

Feature

Descripción:
Un feature es una propiedad, es decir, una operación o un atributo el cual está definido dentro de un classifier.
Feature es una metaclase abstracta.

Generalizaciones:
NamedElement

Atributos:

- No define atributos adicionales.

Model

Descripción:
Un model representa a un modelo colaborativo.

Generalizaciones:
Namespace

Atributos:
- No define atributos adicionales.

Constraints:

[1] No pueden definirse modelos anidados

```
inv notNestedModels:
self.ownedElement -> select (e | e.oclIsKindOf(Model)) -> size() = 0
```

NamedElement

Descripción:
Un namedElement representa a los elementos con nombre. El nombre del elemento es opcional, si se especifica debe ser un string válido, incluyendo los string vacíos

Generalizaciones:
Element

Atributos:
- name: String [0..1]. El nombre del elemento.

Namespace

Descripción:
Un namespace contiene un conjunto de elementos donde cada uno de estos tiene un nombre único que lo identifica dentro del espacio de nombres.

Generalizaciones:
Element

Atributos:
- ownedElement: indica los elementos contenidos dentro del namespace.

Operation

Descripción:
Una operación es propiedad de una clase y puede ser invocada en el contexto de los objetos que son instancias de esa clase. Es posible invocar una operación sobre cualquier objeto que es directa o indirectamente instancia de la clase. Dentro de la invocación, el contexto de ejecución incluye al objeto y los valores de los parámetros.

Generalizaciones:
Feature

Atributos:

- class : Class [0..1]. La clase a la cual pertenece la operación.
- ownedParameter : Parameter [*] {ordered, composite }. Representa los parámetros de la operación.

Parameter

Descripción:
Un parámetro es un elemento con tipo que representa un parámetro de una operación. Cuando se invoca una operación, se pasa por cada parámetro un argumento. Cada parámetro tiene un tipo.

Generalizaciones:
NamedElement

Atributos:

- operation: Operation [0..1]. La operación a la cual pertenece el parámetro

ParameterDirectionKind

Descripción:
Un ParameterDirectionKind es un enumerativo que define literales usados para especificar la dirección de los parámetros. Los posibles valores son in, out e inout que indica que se trata de un parámetro de entrada, de salida o entrada y salida respectivamente.

Generalizaciones:

Atributos:

TypedElement

Descripción:
Un elemento tipado (typedElement) tiene un tipo, que sirve como restricción del tipo de valores que el elemento puede representar. TypedElement es una metaclase abstracta. Los valores representados por el typedElement deben ser instancias de su tipo. Un elemento sin tipo asociado puede representar valores de cualquier tipo.

Generalizaciones:
NamedElement

Atributos:

- type: Type [0..1] El tipo del TypedElement.

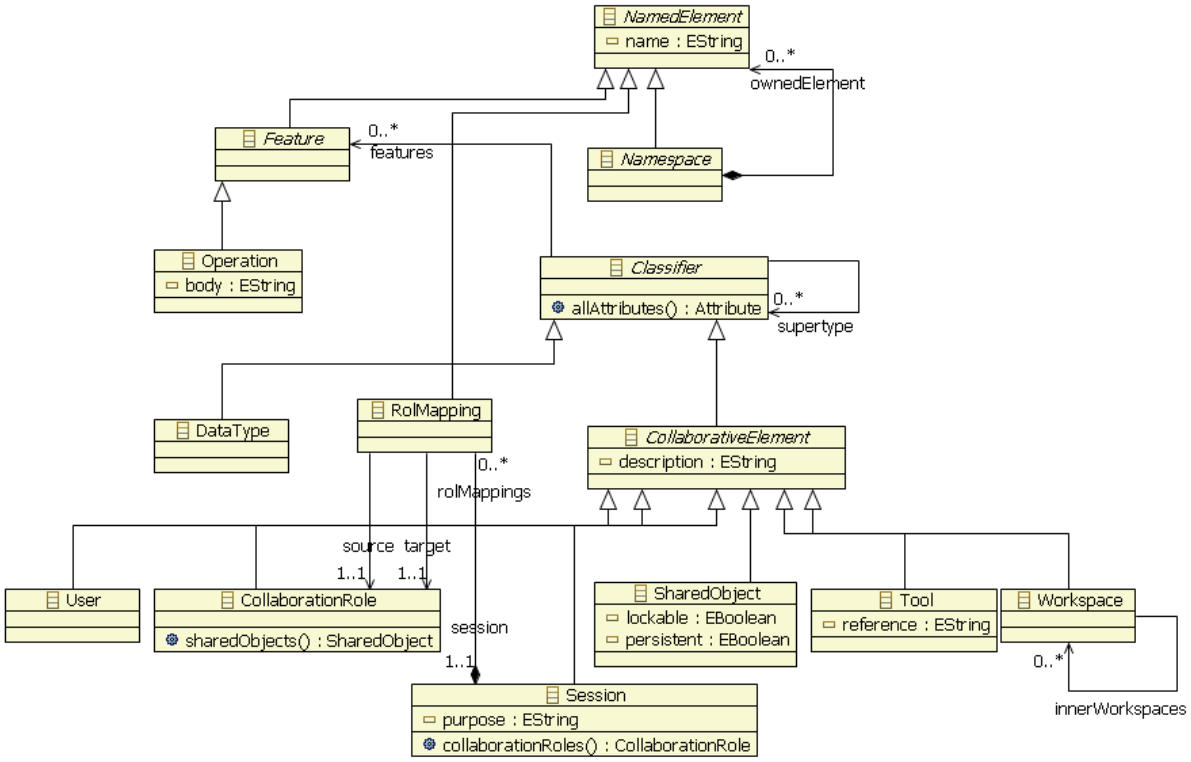


Figura 8-2 Diagrama Collaborative Elements

En el diagrama de la figura 8-2 se pueden ver las entidades que representan los diferentes elementos colaborativos. A continuación se detallan cada uno de ellos.

CollaborationRole

Descripción:
Un collaborationRole describe un conjunto de habilidades, competencias y responsabilidades de un participante o de un conjunto de participantes.

Generalizaciones:
CollaborativeElement

Atributos:

- no define atributos adicionales.

RoleMapping

Constraints:

[1] El collaborationRole destino debe estar relacionado a la session

```
inv targetInSession:
   self.session.collaborationRoles -> includes (self.target)
```

[2] El collaborationRole source no debe pertenecer a la session

```
inv sourceNotInSession:
   self.session.collaborationRoles -> not includes
(self.source)
```

Session

Descripción:
Una sesión es un período de interacción sincrónica soportada por un sistema colaborativo.

Generalizaciones:
CollaborativeElement

Atributos:

- purpose: es un string que especifica el propósito de la interacción

Constraints:

Operaciones Adicionales:

[1] La operación collaborationRoles retorna un conjunto conteniendo todos los collaborationRoles que están relacionados mediante una ParticipationRelationship con la sesión.

Shared Object

Descripción:

Un objeto compartido es cualquier cosa que los usuarios puedan crear. Los usuarios también pueden usar los objetos colocados por un tutor en su ambiente o sesión para aprender algo como ser papers, páginas web, etc.

Generalizaciones:

CollaborativeElement

Atributos:

- lockable: es un booleano que especifica si el objeto compartido se puede bloquear. El valor por defecto es falso.
- isPersistent: es un booleano que indica si el objeto compartido es persistente. El valor por defecto es falso.

Tool

Descripción:

La herramienta provee una forma de controlar la información. Instancias de herramientas son:

- Chat
- Pizarrón compartido o shared blackboard
- Text editors
- Graphical tools

Generalizaciones:

CollaborativeElement

Atributos:

- reference: descripción de la herramienta a la cual representa. Incluye también información sobre la versión e implementación de la misma.

User

Descripción:

Un usuario representa quién realiza o participa en una actividad.

Generalizaciones:

CollaborativeElement

Atributos:

- no define atributos adicionales

Workspace

Descripción:
El Workspace es el lugar donde se lleva a cabo la colaboración. Dentro de un workspace hay herramientas que son utilizadas para comunicarse y trabajar con los objetos compartidos. Los protocolos estructuran las interacciones de los roles en el workspace y el uso de las herramientas por ellos. Un workspaces o grupo de workspaces no son suficientes para definir una aplicación colaborativa. Los workspaces pueden ser compuestos en otros workspaces de manera tal de crear ambientes más complejos de colaboración.

Generalizaciones:
CollaborativeElement

Atributos:
-innerWorkspaces: conjunto de workspaces que forman parte del workpace representado.

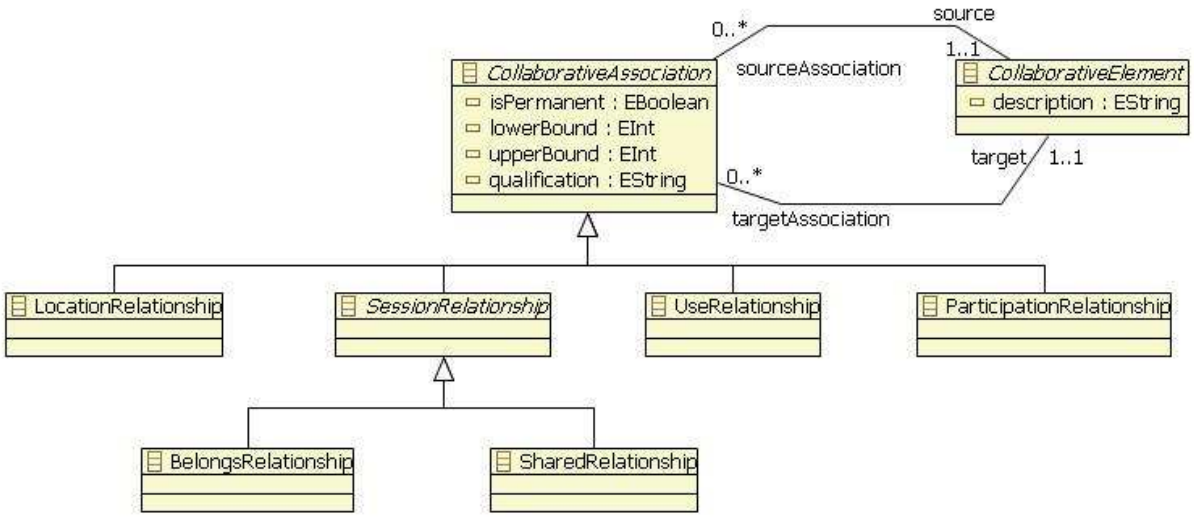


Figure 8-3 Asociaciones colaborativas

En el diagrama de la figura 8-3 se muestran las distintas asociaciones entre los elementos colaborativos.

BelongsRelationship

Descripción:
La relación de dependencia (BelongsRelationship) representa la pertenencia de la sesión en dicho espacio o contexto. De esta forma todas las sesiones que pertenezcan a un workspace están contextualizadas y por lo tanto

podrán tener acceso a las sesiones compartidas que hagan referencia a ese espacio

Generalizaciones:
SessionRelationship

Atributos:
- no define atributos adicionales

LocationRelationship

Descripción:
Una LocationRelationship se usa para especificar la ubicación de los objetos compartidos o herramientas dentro de los espacios de trabajo

Generalizaciones:
CollaborativeAssociation

Atributos:
- no define atributos adicionales

Constraints:
[1] Una LocationRelationship se define entre un workspace y una herramienta o un objeto compartido.

```
inv validLocationRelationship:
self.source.oclIsKindOf(Workspace) and
              (self.target.oclIsKindOf(SharedObject or
           self.target.oclIsKindOf(Tool))
```

ParticipationRelationship

Descripción:
Una ParticipationRelationship se usa para indicar la participación de un rol en una session.

Generalizaciones:
CollaborativeAssociation

Atributos:
- purpose: es un string que especifica el propósito de la interacción

Constraints:
[1] Los elementos conectados son instancias de CollaborationRole y Session

```
inv validParticipationRelationship:
```

```
self.source.oclIsKindOf(CollaborationRole) and
self.target.oclIsKindOf(Session)
```

SharedRelationship

Descripción:
La relación comparte (*SharedRelationship*) representa la posibilidad de que usuarios en una sesión perteneciente al mismo espacio de una sesión compartida tengan acceso a dicha sesión. Una sesión compartida es instanciada una única vez durante toda la ejecución de la aplicación Groupware. Por lo tanto, al ser activada una instancia de sesión en un espacio, activarán simultáneamente a las respectivas sesiones compartidas del mismo.

Generalizaciones:
SessionRelationship

Atributos:
- no define atributos adicionales

Constraints:

SessionRelationship

Descripción:
Las relaciones entre los espacios de trabajo y las sesiones están modeladas como una clase de la jerarquía encabezada por la clase abstracta *SessionRelationship*. Existen dos tipos de relaciones básicas para la contextualización de sesiones: La relación de pertenencia (BelongsRelationship) y la relación comparte (SharedRelationship)

Generalizaciones:
CollaborativeAssociation

Atributos:
- purpose: es un string que especifica el propósito de la interacción

Constraints:
[1] Los elementos conectados son instancias de espacios de trabajo y sesiones

```
inv validSessionRelationship:
self.source.oclIsKindOf(Workspace) and
self.target.oclIsKindOf(Session)
```

UseRelationship

Descripción:

Para expresar relaciones de uso entre los roles o sesiones, y las herramientas o los objetos compartidos.

Generalizaciones:
CollaborativeAssociation

Atributos:

Constraints:

[1] Una useRelationship se usa entre roles o sesiones, y las herramientas u objetos compartidos

```
inv validUseRelationship:
(self.source.oclIsKindOf(CollaborationRole) or
self.source.oclIsKindOf(Session) )
and
(  self.source.oclIsKindOf(Tool) or
self.source.oclIsKindOf(SharedObject) )
```

8.3. Implementación del metamodelo

La sintaxis abstracta se definió utilizando el plugin EMF. Para esto se utilizó el editor visual que provee para dibujar las metaclases y las relaciones entre ellas. Teniendo en cuenta que el editor permite separar varias vistas sobre el mismo modelo, se aprovechó esta característica creando 3 gráficos. El primero contiene los elementos generales del modelo, el siguiente la jerarquía de los elementos colaborativos y el último las jerarquías de las relaciones. Esta facilidad es clave para manejar modelos de gran complejidad. Las vistas definidas se corresponden con las figuras presentadas en la sección anterior. El modelo completo se encuentra en el archivo cm.ecore. Este archivo tiene formato xmi, lo cual permite su visualización en forma de árbol provista por EMF. En la figura 8-4 se puede ver el modelo en su totalidad.

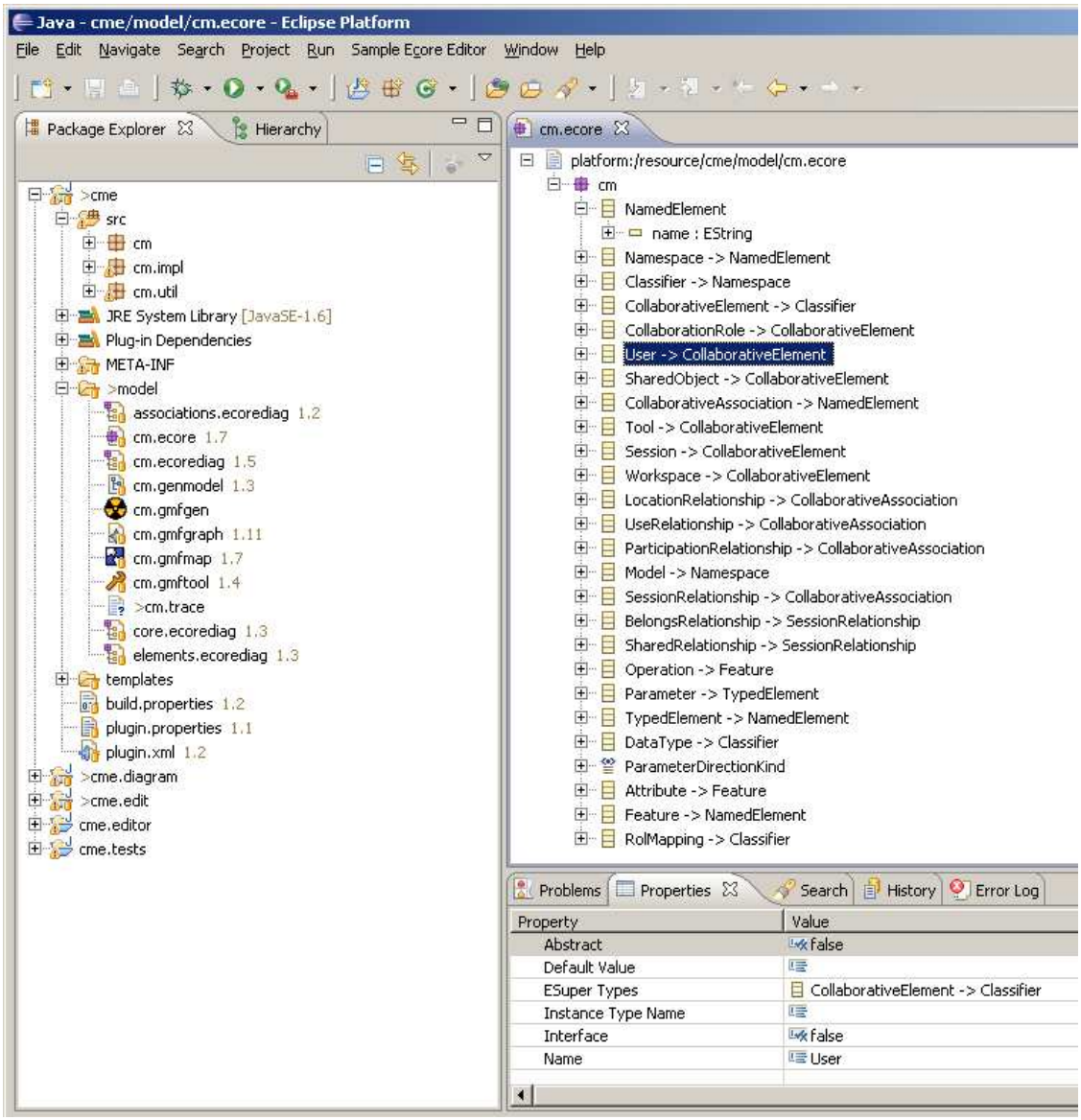


Figura 8-4 Archivo Ecore con el modelo instanciado

A continuación, se agregaron las restricciones de OCL utilizando el correspondiente plugin. Las mismas se agregan como anotaciones sobre los elementos que restringen. De modo similar se incluyeron las operaciones adicionales definidas en el metamodelo.

En la figura 8-5 se puede ver la implementación de la regla *distinctAttributesName* para la metaclase Classifier. También se ve la definición de una operación adicional llamada *allAttributes*.

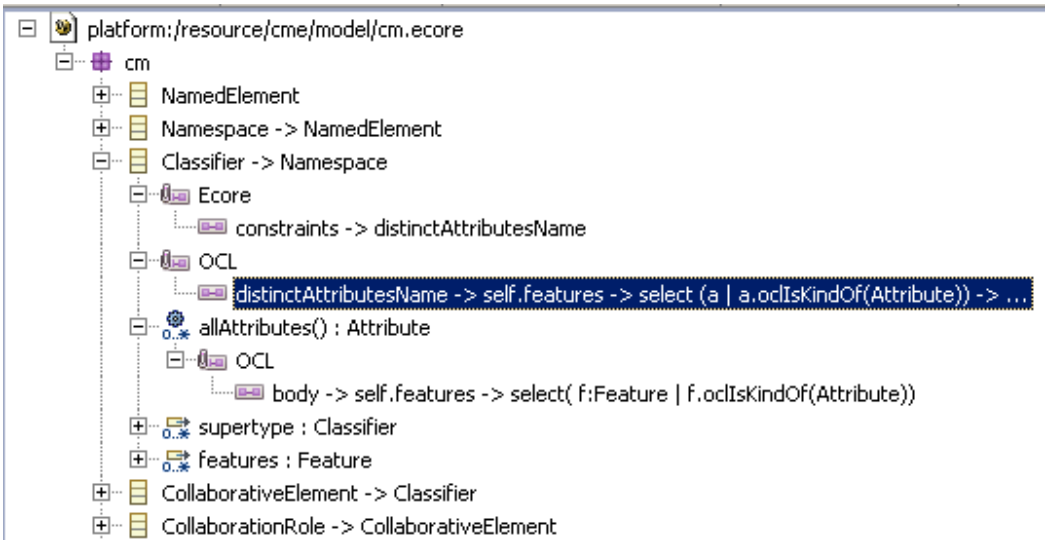


Figura 8-5 Reglas OCL

Una vez terminados los pasos anteriores se procedió a la generación de código. A continuación se muestra parte del código generado donde se invoca la validación mostrada.

```
public boolean validateClassifier_distinctAttributesName(
                  Classifier classifier, DiagnosticChain diagnostics,
                  Map<Object, Object> context) {
. . .
Query<EClassifier, ?, ?> query = OCL_ENV
            .createQuery(classifier_distinctAttributesNameInvOCL);

. . .
}
```

8.4. Definición de la sintaxis concreta

La sintaxis abstracta de un DSL debe ser presentada amigablemente por lo cual la mayoría de las veces es necesario desarrollar más de una sintaxis concreta. En este trabajo se presentan dos formas de sintaxis concreta, la primera visual, en forma de diagrama, y la segunda en forma de texto.

8.4.1. Definición de la sintaxis concreta en forma gráfica

Para definir la sintaxis concreta, y teniendo en cuenta el objetivo de construir un editor gráfico, se eligió un icono para cada entidad colaborativa. A su vez, las relaciones entre los mismos se representan por medio de flechas dirigidas.

La mayoría de las propiedades internas de las entidades no tienen representación gráfica, sino que se puede acceder a las mismas utilizando el componente de propiedades de Eclipse. Dentro de este grupo se encuentran los atributos, parámetros y operaciones.

Model

Un modelo es objeto raíz de la aplicación, es decir, es el contenedor de los demás elementos definidos en el diagrama.

CollaborationRole

Se identifica con el icono de un obrero. Debajo del mismo se visualiza el nombre asignado, el cual es editable.

RoleMapping:

Es un compartimiento dentro de la sesión. Desde la vista de propiedades se puede editar las relaciones entre los distintos CollaborationRole. Y en el dibujo principal, se muestra el nombre del rol origen, seguido de una flecha, finalizando con el nombre del rol destino.

Session

Se identifica con el icono de dos personas conversando. Debajo del mismo se visualiza el nombre asignado, el cual es editable.

Shared Object

Se identifica con el icono una hoja de papel que representa un objeto que es compartido dentro del ambiente. Debajo del mismo se visualiza el nombre asignado, el cual es editable.

Tool

Se identifica con el icono de las herramientas. Debajo del mismo se visualiza el nombre asignado, el cual es editable.

User

Se identifica con el icono de una figura humana. Debajo del mismo se visualiza el nombre asignado, el cual es editable.

Workspace

Se identifica con el icono de una casa. Debajo del mismo se visualiza el nombre asignado, el cual es editable.

Relatioships

Las relaciones involucradas en los modelos definidos (BelongsRelationship, LocationRelationship, ParticipationRelationship, SharedRelationship y UseRelationship) se representan siempre por medio de una flecha dirigida. Sobre la misma, y de forma similar a los estereotipos de

UML, se indica el tipo de relación. También se muestra la cardinalidad de las mismas, la cual se puede modificar desde el editor visual.

Para todos los casos, el editor sugiere como nombre de la relación, el nombre del componente origen y destino, separados por una flecha.

En la figura 8-6 se puede ver el editor gráfico con un modelo de ejemplo.

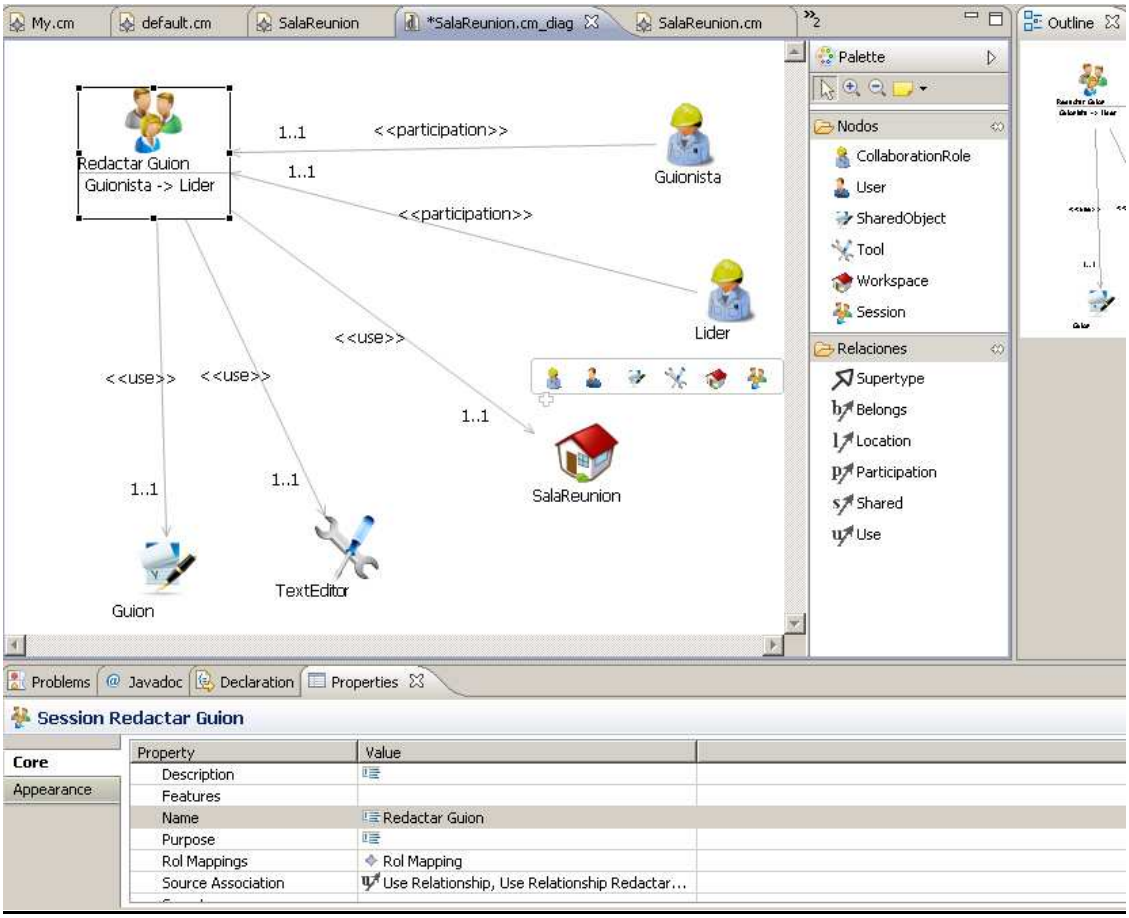


Figura 8-6 Editor gráfico

8.4.2. Implementación de la sintaxis concreta en forma gráfica

Para la implementación de la sintaxis concreta en forma gráfica se utilizó el plugin de GMF. Se definieron los tres archivos necesarios: cm.gmfgraph con la configuración gráfica de los elementos, cm.gmftool con la definición de la paleta del editor y cm.gmfmap que combina los anteriores y permite generar el código automáticamente.

Se hicieron adaptaciones con respecto a la forma en que se muestran los elementos del modelo. GMF permite dibujarlos como polígonos, pero no como imágenes. El beneficio principal de usar polígonos es la capacidad de escalarlos. Teniendo en cuanta que en nuestro caso no era necesaria esta característica, se optó por imágenes PNG para representar los nodos del

modelo. Para obtener esto fue necesario extender el código generado agregando el método *addImage* en cada clase EditPart del modelo. Por ejemplo para Session, este cambio se realizó en la clase SessionEditPart. Este método recupera la imagen a mostrar y la asocia al elemento del modelo, de tal manera que luego el editor la dibuje en su representación.

A continuación se puede ver el código del método *addImage* en la clase SessionEditPart.

```
private void addImage() {
      ImageDescriptor imageDescriptor;
      ImageFigure icon;
      imageDescriptor = ImageDescriptor.createFromFile(
                        SessionEditPart.class,
                        EditPartConstants.IMAGE_SESSION_FIGURE_PATH);
      icon = new ImageFigure((Image) getResourceManager().createImage(
                              imageDescriptor));
      this.add(icon);
}
```

8.4.3. Sintaxis concreta en forma de texto

Para la definición de la sintaxis concreta en forma de texto se utilizó el plugin EMFText. Se estableció una sintaxis textual cuya definición se plasmó en un archivo ".cs". Para ejemplificar el editor resultante, en la figura 8-7 se muestra el ejemplo que se vio en la sección previa pero utilizando la sintaxis textual.

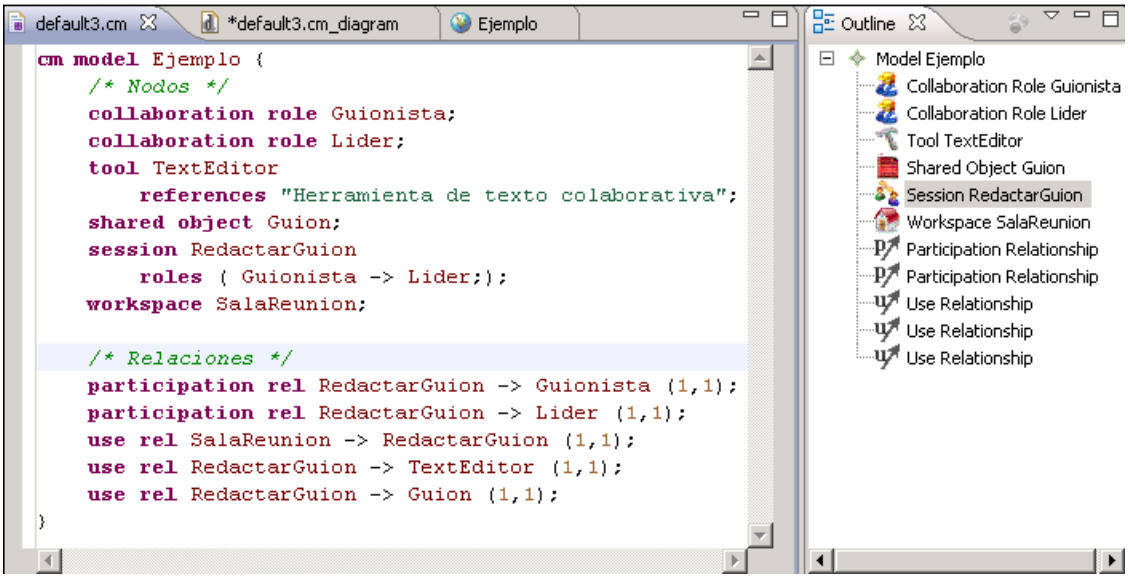


Figura 8-7 Editor textual

A continuación se muestra parte del archivo de definición creado. Para facilitar su lectura, solamente se puede ver la definición de la entidad Session y la relación Use, siendo las restantes análogas a estas.

```
SYNTAXDEF cm FOR  <http://cm/1.0> <cm.genmodel> START Model
TOKENS {
   DEFINE ML_COMMENT $'/*'.*'*/'$ COLLECT IN comments;
   DEFINE CARDINALITY_LITERAL $'-1'|'0'|'1'..'9''0'..'9'*$;
   DEFINE BOOLEAN $'true'|'false'$;
}
TOKENSTYLES {
   "TEXT" COLOR #770000;
}
RULES {
   // syntax definition for class 'Model'
   Model    ::= "cm" "model" name[]?
         "{"  (!1ownedElement ";")* !0"}"
;
   // syntax definition for Collaborative Elements
   Session ::= "session" name[]
         (!1 "supertype" supertype[])?
         (!1 "desc" description['"','"'])?
         (!1 "purpose" purpose['"','"'])?
         (!1 "roles" "("
               (!2 rolMappings ";")+ !1")")?
         ;
   // syntax definition for Collaborative Associations
   UseRelationship ::= "use" "rel" (name['"','"'])?
         source[] "->" target[]
         ("("#0 lowerBound[CARDINALITY_LITERAL] ","
               upperBound[CARDINALITY_LITERAL] #0 ")")?
         (!1 "qualification" qualification[])?
         (!1 "permanent" isPermanent[BOOLEAN])?
         ;
}
```

Este plugin está completamente integrado con EMF y GMF. La versión textual reemplaza el archivo xmi que provee EMF para manejarlo con su editor básico. De esta manera, a partir de un archivo con el formato de la sintaxis presentada, se puede obtener la vista gráfica y viceversa. También permite usarlo como base para las herramientas que trabajan sobre un archivo EMF, como es el caso de la herramienta de documentación que se presenta en la siguiente sección.

8.5. Semántica

En el momento en el que se comenzó a trabajar con la definición del lenguaje de elementos colaborativos, aún no se contaba con la especificación

de la semántica. Por ese motivo se resolvió generar, en su lugar, una documentación en forma de HTML que igualmente muestra la facilidad de Eclipse para realizar traducciones. Por ejemplo, el modelo se representará como una página HTML cuyo título se corresponderá con su nombre. Además listará las entidades que se defienen en él, permitiendo navegar a cada una de ellas. Esto se muestra en la figura 8-8.

Figura 8-8 Documentación del modelo

De la misma manera, la página que representa cada entidad tendrá como título su nombre. Dentro del cuerpo se podrán ver sus atributos y relaciones en las que participa, como se puede ver en la figura 8-9.

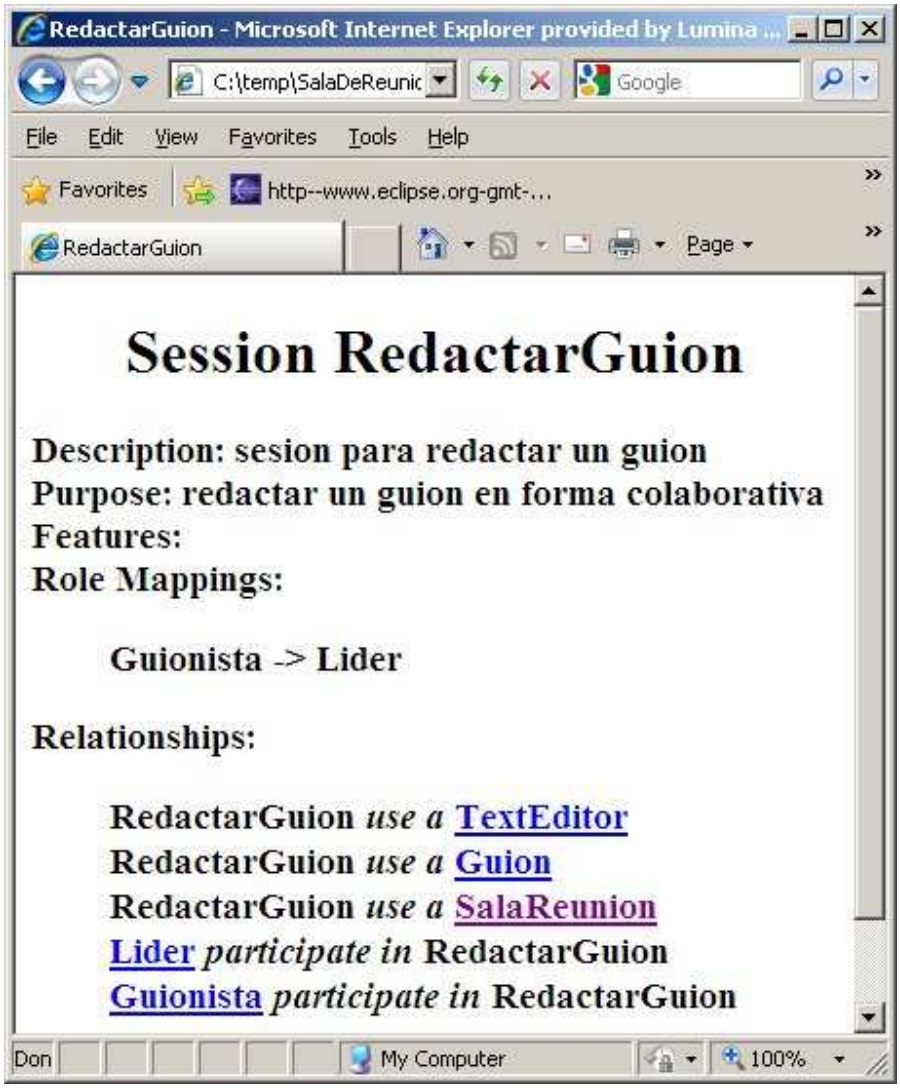


Figura 8-9 Documentación de un elemento

Para la generación de esta representación en forma de texto se utilizó el plugin de MOFScript, creando el archivo *transformation.m2t* donde se definieron las reglas necesarias parea la traducción de cada objeto del modelo. Allí se especificó que habría un archivo principal, el cual representaría al modelo, y un archivo secundario por cada entidad. Además se especificó como se mostrarían los atributos y relaciones entre las mismas. En la figura n se puede ver la porción de código del archivo que define *transformation.m2t* la representación del modelo.

```
cm.Model::main () {
file (self.name + ".html")
        self.putHTMLHeader(self.name)
        '<h1>Modelo ' self.name '</h1>\n'

        self.ownedElement->forEach(p:cm.NamedElement) {
```

```
                '<h3>\n'
                p.mapNamedElement(self.name)
                p.getName(self.name)
                '</h3>'
                      p.mapEntityToFile(self.name)
            }
            '\n'
            self.putHTMLFooter()
}
```

8.6. Resumen

En este capítulo se ejemplificó el desarrollo de un DSL para la definición de sistemas colaborativos. Se identificaron los conceptos propios del dominio y a partir de estos se definió una sintaxis abstracta por medio de un metamodelo. Para poder instanciarlo gráficamente, se utilizó GMF para construir un editor y se definieron íconos que representan cada uno de los conceptos encontrados.

Para su instanciación en forma textual, se definió una gramática representada con una sintaxis similar a BNF que se plasmó en un archivo de configuración de EMFText. Esto permitió obtener un editor de texto. Por último, se da una idea de cómo definir su semántica, utilizando el plugin MOFScript para generar una documentación del modelo por medio de una traducción a páginas HTML.

9 Conclusiones

El modelado específico de dominio tiene como propósito crear modelos para un dominio utilizando un lenguaje enfocado y especializado. Incluye la idea de generación automática de código ejecutable directamente desde los modelos, para que las aplicaciones finales puedan ser generadas a partir de estas especificaciones en alto nivel. De esta manera, se libera al desarrollador de las tareas de codificación y mantenimiento del código fuente y se incrementa significativamente su productividad. Asimismo, como el código no se escribe manualmente, se reducen los defectos en las aplicaciones resultantes y se mejora la calidad de estos productos. Por otro lado, da la oportunidad a los expertos del dominio de contribuir directamente con el desarrollo. Los elementos de DSLs no se encuentran demasiado distantes de los elementos de modelado de la filosofía MDD. MDD promete mejorar el proceso de construcción de software basándose en un proceso guiado por modelos y soportado por potentes herramientas. En este paradigma se asigna a los modelos un rol central y activo: son al menos tan importantes como el código fuente. Los modelos se van generando desde los más abstractos a los más concretos aplicando transformaciones y/o refinamientos, hasta finalmente llegar al código. Las transformaciones entre modelos constituye el motor principal de MDD.

En la práctica, cada solución DSM diseña un lenguaje y un generador de código para una organización en particular. Se enfocan en dominios pequeños, porque son más fáciles de definir y permiten mejores posibilidades para su automatización. Como los conceptos del lenguaje son utilizados dentro de la organización, se reduce considerablemente el tiempo de aprendizaje del mismo.

Actualmente las discrepancias entre DSM y MDD se han reducido. Se puede comparar el uso de modelos así como la construcción de la infraestructura respectiva en DSM y en MDD. En general, DSM usa los conceptos dominio, modelo, metamodelo y meta-metamodelo como MDD sin mayores cambios y propone la automatización del ciclo de vida del software. Ambas propuestas anhelan la construcción de software más confiable, focalizándola en el desarrollo de modelos y dejando a un lado la escritura manual de código ejecutable.

Existe una familia importante de herramientas para crear soluciones en DSM que permiten diseñar un lenguaje específico de dominio, para luego construir herramientas de modelado y generadores de código para ese lenguaje.

- Microsoft con las "Software Factories".
- Metacase con su producto MetaEdit+.
- Eclipse con su proyecto llamado Eclipse Modeling Project.

En este trabajo hicimos enfoque en la propuesta de Eclipse, ya que es la única de código abierto entre las mencionadas. Los proyectos en los que trabaja Eclipse se enfocan principalmente en construir una plataforma de desarrollo abierta, la cual está compuesta por frameworks extensibles y otras herramientas que permiten construir y administrar software. También permite a los usuarios mejorar el ambiente y extender su funcionalidad a través de la creación de plugins. Se profundizó en el análisis de los distintos pasos en el desarrollo de un DSL: la definición de la sintaxis abstracta, la sintaxis concreta y su semántica. Para la implementación de cada uno de estos pasos se presentaron algunos plugins de Eclipse.

Para la plataforma Eclipse existen una gran variedad de plugins para la definición de DSLs. Día a día crecen el número de proyectos y se producen actualizaciones sobre los ya existentes. Por esta razón, no fue posible presentarlos a todos ni cubrirlos en su totalidad. Para la sintaxis abstracta se detallaron los plugins EMF y OCL. Para la sintaxis concreta, se presentaron GMF, EuGENia, EMFText, Xtext y TCS. Para la definición de la semántica se presentó MOFScript y JET.

Para finalizar el trabajo se presentó un caso de estudio en donde se aplicaron las técnicas detalladas en los capítulos previos. El mismo consistió en la implementación de un DSL para modelar sistemas colaborativos.

Luego de analizar y profundizar en la implementación, identificamos aspectos positivos y negativos en el uso de estas herramientas. Dentro de los positivos, se encuentran los siguientes:

1. Se puede escribir un DLS, junto con componentes de Eclipse que lo soporten, sin la necesidad de escribir código.
2. Para introducir cambios específicos, es posible editar el código fuente con la ventaja de que no se perderá en posteriores invocaciones al generador automático de código.
3. Existen varias fuentes de documentación y soporte para utilizar los plugins usados. Dentro de los mismos se pueden mencionar tutoriales en internet, foros, libros publicados, asistentes y videos on line.
4. El plugin resultante provee un gran número de funcionalidades sin que sea necesario implementarlas. EMF provee, para el código generado, mecanismos de persistencia, validaciones y dependencia. De la misma manera, GMF nos brinda un editor visual que soporta comandos, hacer y deshacer, zoom y diferentes vistas de los elementos creados.

Dentro de los negativos, se puede mencionar:

1. La evolución de los metamodelos es difícil de aplicar a los modelos ya implementados.
2. La gran variedad de herramientas existentes dificulta la elección de las mismas, así como también el aprendizaje sobre ellas.

3. El desarrollo de un DSL es considerado una tarea difícil, ya que requiere un profundo conocimiento del dominio y mucha experiencia en el tema. Pocas personas tienen ambas habilidades, y debido a eso, la decisión de desarrollar un DSL es pospuesta muchas veces indefinidamente.

En resumen, como aportes de este trabajo se puede mencionar que se presentó el proceso de creación de un DSL, detallando los pasos necesarios para su definición. También se enunciaron los beneficios de definir y utilizar un DLS en lugar de utilizar otras técnicas de desarrollo. Luego se expusieron diferentes plugins actuales de Eclipse vinculados a la implementación de un DSL. Para cada uno de estos se indicó su objetivo, su forma de uso y las ventajas y desventajas que poseen. Por último, se implementó un DSL para modelar sistemas colaborativos donde se integraron todos los conceptos vistos en este trabajo. Como conclusión podemos destacar que a pesar que construir un DSL es una tarea muy costosa, las facilidades que proveen las herramientas existentes para Eclipse agilizan mucho esta tarea quedando la complejidad centrada en la definición del propio lenguaje.

Los posibles trabajos a futuro se plantean en las siguientes líneas:

1. Implementar una herramienta integradora que simplifique la tarea de definir y construir un DSL. Esto mismo se puede lograr construyendo un asistente que guíe al usuario durante todo el proceso.
2. Continuar con la implementación del DSL para ambientes colaborativos definido como caso de estudio. El mismo puede ser extendido para que genere un esqueleto de su implementación en algún lenguaje de programación utilizando los diagramas dinámicos presentados en el trabajo [1].
3. Definir e implementar un plugin que facilite la construcción de figuras para los editores que genera GMF.
4. Investigar la forma de solucionar el problema que se presenta debido a la evolución de los metamodelos y su consecuente desincronización respecto a los modelos ya generados. También se debería atacar este inconveniente en los pasos que se realizan durante la implementación del plugin que soporta al DSL creado. Por ejemplo, al realizar una modificación en el metamodelo, se debe invocar la generación de código que realiza EMF y la generación del plugin de GMF.

Bibliografía

[1] Bibbo Luis Mariano, García Diego, Pons Claudia, "A Domain Specific Language for the Development of Collaborative Systems" International Conference of the Chilean Computer Science Society, 2008.

[2] Budinsky Frank, Steinberg David, Merks Ed, Ellersick Ray, Grose Timothy. "Eclipse Modeling Framework (EMF)". Addison Wesley Professional.

[3] Clark A, Evans Andy, Sammut Paul, Willans James. "Applied Metamodelling A Foundation for Language Driven Development". Xactium, 2004

[4] Cook Steve, Jones Gareth, Kent Stuart, Wills Alan Cameron, "Domain-Specific Development with Visual Studio DSL Tools", Addison-Wesley, 2007.

[5] Ellis, C.A., Gibbs, S.J., Rein, G.L., "Groupware: some issues and experiences", Communications of the ACM, 34(1) (1991).

[6] The Eclipse Project. Home Page. Copyright IBM Corp, 2000-2009. http://www.eclipse.org/.

[7] EMF, The Eclipse Foundation, http://www.eclipse.com/emf/.

[8] "EMFText How To: Developing an Organigram DSL using EMFText", disponible en http://sites.google.com/site/luismiguelpedro/curriculum-vitae/technical-reports/emftextReport.pdf.

[9] Fowler Martin, "Language Workbenches: The Killer-App for Domain Specific Languages?", 2005.

[10] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Professional Computing Series, 1994.

[11] Garcia Miguel, Shidqie A. J, "Ocl tools: Status and perspectives", Technische Universität Hamburg-Harburg, 2007.

[12] GEF, The Eclipse Foundation, http://www.eclipse.com/gef/.

[13] GMF, The Eclipse Foundation, http://www.eclipse.com/gmf/.

[14] Gronback Richard, "A Domain-Specific Language Toolkit", Addison-Wesley, 1 edición, 2009.

[15] Heidenreich Florian et al, "Derivation and Refinement of Textual Syntax for Models", In Proceedings of the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009).

[16] Hudak P, "Modular domain specific languages and tools". In Proceedings of the 5th International Conference on Software Reuse (JCSR '98), pages 134–142. IEEE Computer Society, 1998.

[17] Kelly Steven, Tolvanen Juha-Pekka, "Domain Specific Modeling - Enabling Full Code Generation", John Wiley & Sons, Inc., 2008.

[18] Kleppe Anneke, "Software Language Engineering", Addison Wesley Professional, 1 edición, 2009.

[19] MDA Guide, v1.0.1, omg/03-06-01, June 2003. http://www.omg.org.

[20] "Meta Object Facility (MOF) 2.0 Core Specification. OMG adopted specification", en http://www.omg.org, 2003.

[21] MetaEdit + en http://www.metacase.com/MetaEdit.html

[22] "MOFScript User Guide, version 0.6 (MOFScript v 1.1.11)", disponible en http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf.

[23] OMG (Object Management Group), en http://www.omg.org

[24] Software Factories, en http://msdn.microsoft.com/en-us/library/aa480032.aspx

[25] UML 2.0. "The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification", en http://www.omg.org, 2003.

[26] "UML 2.0 OCL Specification", disponible en http://www.omg.org/docs/ptc/03-10-14.pdf.

Glosario de siglas y términos

En este Glosario se describen siglas y términos utilizados en el desarrollo de este trabajo. El contenido que sigue es simplemente, una ayuda rápida para ilustrar algunos conceptos.

Eclipse

Eclipse es una plataforma de software de código abierto independiente de otras plataformas. Esta plataforma, típicamente ha sido usada para desarrollar entornos integrados de desarrollo (del inglés IDE), como el IDE de Java llamado Java Development Toolkit (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse (y que son usados también para desarrollar el mismo Eclipse). Sin embargo, también se puede usar para otros tipos de aplicaciones cliente. Eclipse es también una comunidad de usuarios, extendiendo constantemente las áreas de aplicación cubiertas. Un ejemplo es el recientemente creado Eclipse Modeling Project, cubriendo casi todas las áreas de Model Driven Engineering. Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente no lucrativa, que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

EMF - Eclipse Modeling Framework Project (EMF)

El proyecto EMF es un framework de modelado y facilidades de generación de código para la creación de herramientas de instalación y otras aplicaciones basadas en un modelo de datos estructurados. Desde una especificación de modelo descrito en XMI, EMF ofrece herramientas y soporte en tiempo de ejecución para producir un conjunto de clases Java para el modelo, junto con un conjunto de clases adaptadoras que permiten la visualización y la edición basada en comandos, del modelo, y un editor básico.

GEF

GEF es el acrónimo de Graphical Editing Framework. Es un framework implementado para la plataforma Eclipse que ayuda en el desarrollo de componentes gráficos. Consiste de tres componentes principales, *draw2d* usado para los componentes visuales, Request/Commands usado durante la edición para crear pedidos y comandos capaces de rehacerse y deshacerse, y por último una paleta de herramientas para mostrar las opciones al usuario.

GMF

GMF es el acrónimo de Graphical Modeling Framework, que se traduce como Framework para modelados gráficos. Es un framework implementado

para la plataforma Eclipse. Permite el desarrollo de editores gráficos y es un plugin que depende principalmente de EMF y GEF.

HTML

Es el acrónimo inglés de HyperText Markup Language, que se traduce al español como Lenguaje de Marcas Hipertextuales. Es un lenguaje de marcación diseñado para estructurar textos y presentarlos en forma de hipertexto, que es el formato estándar de las páginas web. Gracias a Internet y sus navegadores, el HTML se ha convertido en uno de los formatos más populares y fáciles de aprender que existen para la elaboración de documentos para web.

JSP

JavaServer Pages (JSP) es una tecnología Java que permite generar contenido dinámico para web, en forma de documentos HTML, XML o de otro tipo. Esta tecnología es un desarrollo de la compañía Sun Microsystems. La Especificación JSP 1.2 fue la primera que ser liberó y en la actualidad está disponible la Especificación JSP 2.1. Las JSPs permiten la utilización de código Java mediante scripts. Además es posible utilizar algunas acciones JSP predefinidas mediante etiquetas. Estas etiquetas pueden ser enriquecidas mediante la utilización de Librerías de Etiquetas (Tag Libraries) externas e incluso personalizadas.

KM3

KM3 o Kernel Meta Meta Model es un lenguaje neutral para escribir motamodelos, y para definir la sintaxis abstracta de DSLs. Su sintaxis es muy simple y tiene algunas similitudes con la del lenguaje Java. Un archivo KM3 (.km3) puede ser transformado a un metamodelo y puede serializarse como un archivo XMI.

MDA

En español, Arquitectura conducida por modelos. Han surgido varios enfoques dentro del ámbito de MDE, pero sin duda la iniciativa más conocida y extendida es la MDA, acrónimo de Model Driven Architecture, presentada por el consorcio OMG (Object Management Group) en noviembre de 2000 con el objetivo de abordar los desafíos de integración de aplicaciones y los continuos cambios tecnológicos. MDA propone el uso de un conjunto de estándares (descriptos en este Glosario) como MOF, UML, JMI o XMI. Su objetivo es separar la especificación de la funcionalidad del sistema de su implementación sobre una plataforma concreta, por lo que se hace una distinción entre modelos PIM y modelos PSM.

MDD

Otro acrónimo relacionado a MDE es Model-Driven Development (MDD), que en español se traduce como Desarrollo de Software Conducida por Modelos. Es visto como un sinónimo de MDE, ambos describen la misma metodología de desarrollo de Software.

MDE

Acrónimo inglés de Model Driven Engeenering, en español se traduce como Ingeniería de Software Conducida por Modelos.

El paradigma MDE tiene dos ejes principales: - por un lado hace énfasis en la separación entre la especificación de la funcionalidad esencial del sistema y la implementación de dicha funcionalidad usando plataformas tecnológicas específicas. Por otro lado, en MDE los modelos son considerados los conductores primarios en todos los aspectos del desarrollo de software. MDE identifica dos tipos principales de modelos: modelos con alto nivel de abstracción e independientes de cualquier tecnología de implementación, llamados PIM y modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica, conocidos como PSM. Un PIM es transformado en uno o más PSMs, es decir que para cada plataforma tecnológica específica se genera un PSM específico.

MOF

El Meta Object Facility (MOF), es un estándar de OMG para MDD. La página oficial de referencia se puede encontrar en OMG's Meta Object Facility. MOF se originó en el Lenguaje Unificado de Modelado (UML); OMG tenía la necesidad de contar con una arquitectura de Metamodelado para definir el UML. MOF está diseñado como el nivel más abstracto de una arquitectura de cuatro capas o niveles. Proporciona un meta-metamodelo en la capa superior, denominado nivel M3. Este modelo M3 es el lenguaje utilizado por MOF para construir metamodelos, denominados modelos M2. El ejemplo más destacado de un modelo MOF de nivel M2, es el metamodelo UML, es decir el modelo que describe a UML. Estos modelos M2 describen los elementos del nivel M1, y por lo tanto describen modelos M1. Estas serían, por ejemplo, modelos escritos en UML. La última capa es el nivel M0 o capa de datos. Se utiliza para describir el mundo real (instancias de elementos M1).

OCL

Lenguaje de Restricciones para Objetos (OCL, por su sigla en inglés, Object Constraint Language) es un lenguaje declarativo para describir reglas que se aplican a metamodelos MOF, y a los modelos UML, desarrollado en IBM y en la actualidad parte del estándar UML. OCL inicialmente era sólo un lenguaje de especificación formal integrado a UML. Sin embargo, OCL puede

ser usado con cualquier metamodelo MOF de OMG, incluyendo UML. El Object Constraint Language es un lenguaje de texto preciso que permite definir restricciones y consultas sobre expresiones de objetos de cualquier modelo o metamodelo MOF que de otra manera no pueden ser expresadas mediante la notación gráfica. OCL es un componente clave de la nueva propuesta estándar OMG para la transformación de los modelos, la especificación QVT. Muchos otros lenguajes de transformación de modelos como ATL, también están construidos utilizando OCL.

OMG

El Object Management Group u OMG (de su sigla en inglés Grupo de Gestión de Objetos) es un consorcio dedicado a la gestión y el establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como UML, XMI, CORBA. Es una organización no lucrativa que promueve el uso de tecnología orientada a objetos mediante guías y documentos de especificación de estándares independientes de las implementaciones de las diferentes empresas. El grupo está formado por compañías y organizaciones de software como lo son: Hewlett-Packard (HP), IBM, Sun Microsystems, Apple Computer.

PIM

Es el acrónimo inglés de Platform Independent Model, que se traduce al español como Modelo Independiente de la Plataforma. MDE identifica dos tipos principales de modelos: modelos con alto nivel de abstracción e independientes de cualquier tecnología de implementación, llamados PIM.

PSM

Es el acrónimo inglés de Platform Specific Model, que se traduce al español como Modelo específico de la Plataforma modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica, conocidos como PSM. En MDE un PIM es transformado en uno o más PSMs, es decir que para cada plataforma tecnológica específica se genera un PSM específico.

QVT

En MDD, QVT (Query/ View/ Transformation) es un estándar para transformación de modelos definido por el OMG (Object Management Group). La especificación del lenguaje QVT tiene una naturaleza híbrida, declarativa/imperativa, con la parte declarativa dividida en una arquitectura de dos niveles. Esta especificación define tres paquetes principales, uno por cada lenguaje definido: QVTCore, QVTRelation y QVTOperational. Estos paquetes principales se comunican entre sí y comparten otros paquetes intermedios.

RCP

Se le llama Rich Client Platform al conjunto mínimo de plugins de Eclipse necesario para construir una aplicación cliente rica. Esta aplicación estará basada en el modelo de plugins dinámicos, y su interfase visual estará hecha con el mismo conjunto de herramientas y puntos de extensión que la plataforma Eclipse.

SDO

Service Data Object fue diseñado para unificar y simplificar la manera en que las aplicaciones manejan datos. Utilizando SDO, los desarrolladores de aplicaciones pueden acceder y manipular datos uniformenmente, sin importar la fuente de los mismos (bases de datos relacionales, XMLs, Web services, etc).

UML

Lenguaje Unificado de Modelado (UML, por su sigla en inglés, Unified Modeling Language) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad. Es el lenguaje estándar oficial, respaldado por el OMG (Object Management Group). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software. UML ofrece un estándar para describir la estructura y el comportamiento del sistema, incluyendo aspectos conceptuales tales como procesos de negocios y funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de software reutilizables.

WTP

Web Tools Platform es un conjunto de herramientas para extender la plataforma de Eclipse, orientado a aplicaciones Web. Incluye facilidades para el desarrollo de este tipo aplicaciones por medio de asistentes y editores de código y de gráficos. Además provee un conjunto de herramientas que ayudan en la instalación, ejecución y prueba.

XML

Es el acrónimo inglés de eXtensible Markup Language («lenguaje de marcas extensible»), es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C). Permite definir la gramática de lenguajes específicos. Por lo tanto XML no es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades. Algunos de estos lenguajes que usan XML para su definición son XHTML, SVG, MathML.

XMI

XMI o XML Metadata Interchange (XML de Intercambio de Metadatos) es una especificación para el Intercambio de Diagramas. La especificación para el intercambio de diagramas fue escrita para proveer una manera de compartir modelos UML entre diferentes herramientas de modelado. En versiones anteriores de UML se utilizaba un esquema XML para capturar los elementos utilizados en el diagrama; pero este esquema no decía nada acerca de cómo el modelo debía graficarse. Para solucionar este problema la nueva Especificación para el Intercambio de Diagramas fue desarrollada mediante un nuevo esquema XML que permite construir una representación SVG (Scalable Vector Graphics).

XSD

XML Schema Definition es un lenguaje utilizado para describir la estructura y las restricciones de los contenidos en documentos XML. Con este lenguaje de esquema se pueden definir restricciones de forma muy precisa y más allá de las normas sintácticas que impone el propio XML. Se consigue así una percepción del tipo de documento con un nivel alto de abstracción.

XSD fue desarrollado por el World Wide Web Consortium (W3C) y alcanzó el nivel de recomendación en mayo de 2001.

Anexo I

Gramática del editor textual

A continuación se muestra la definición que se empleó con el plugin EMFText para generar el editor textual, el parser asociado y el módulo de impresión de modelos.

```
SYNTAXDEF cm
FOR  <http://cm/1.0> <cm.genmodel>
START Model

OPTIONS {
      reloadGeneratorModel = "true";
      generateCodeFromGeneratorModel = "true";
}

TOKENS {
      DEFINE SL_COMMENT $'//'(~('\n'|'\r'|'\uffff'))* $ COLLECT IN
comments;
      DEFINE ML_COMMENT $'/*'.*'*/'$ COLLECT IN comments;
      DEFINE CARDINALITY_LITERAL $'-1'|'0'|'1'..'9''0'..'9'*$;
      DEFINE BOOLEAN $'true'|'false'$;
}

TOKENSTYLES {
   "ML_COMMENT" COLOR #008000, ITALIC;
   "SL_COMMENT" COLOR #000080, ITALIC;
   "TEXT" COLOR #770000;
   "CARDINALITY_LITERAL" COLOR #996633;
   "QUOTED_34_34" COLOR #236743;
   "BOOLEAN" COLOR #330099;
}

RULES {
      // syntax definition for class 'Model'
      Model    ::= "cm" "model" name[]?
            "{"  (!1ownedElement ";")* !0"}";

      // syntax definition for Collaborative Elements
      CollaborationRole ::= "collaboration" "role" name[]
                  (!1 "supertype" supertype[])?
                  (!1 "desc" description['"','"'])?
                  ;

      User ::= "user" name[]
                  (!1 "supertype" supertype[])?
                  (!1 "desc" description['"','"'])?
                  ;

      Session ::= "session" name[]
                  (!1 "supertype" supertype[])?
```

```
                    (!1 "desc" description['"','"'])?
                    (!1 "purpose" purpose['"','"'])?
                    (!1 "roles" "("
                          (!2 rolMappings ";")+ !1")")?
                    ;

      SharedObject ::= "shared" "object" name[]
                    (!1 "supertype" supertype[])?
                    (!1 "desc" description['"','"'])?
                    (!1 "lockable" lockable[BOOLEAN])?
                    (!1 "persistent" persistent[BOOLEAN])?
                    ;

      Tool ::= "tool" name[]
                    (!1 "supertype" supertype[])?
                    (!1 "desc" description['"','"'])?
                    !1 "references" reference['"','"']
                    ;

      Workspace ::= "workspace" name[]
                    (!1 "supertype" supertype[])?
                    (!1 "desc" description['"','"'])?
                    (!1 "inners" innerWorkspaces[] ("," innerWorkspaces[])*)?
                    ;

      RolMapping ::= source[] "->" target[]
                    ;
      // syntax definition for Collaborative Associations
      BelongsRelationship ::= "belongs" "rel" (name['"','"'])?
                              source[] "->" target[]
                              ("("#0 lowerBound[CARDINALITY_LITERAL] "," upperBound[CARDINALITY_LITERAL] #0 ")")?
                              (!1 "qualification" qualification[])?
                              (!1 "permanent" isPermanent[BOOLEAN])?
                              ;

      SharedRelationship ::= "shared" "rel" (name['"','"'])?
                              source[] "->" target[]
                              ("("#0 lowerBound[CARDINALITY_LITERAL] "," upperBound[CARDINALITY_LITERAL] #0 ")")?
                              (!1 "qualification" qualification[])?
                              (!1 "permanent" isPermanent[BOOLEAN])?
                              ;

      LocationRelationship ::= "location" "rel" (name['"','"'])?
                              source[] "->" target[]
                              ("("#0 lowerBound[CARDINALITY_LITERAL] "," upperBound[CARDINALITY_LITERAL] #0 ")")?
                              (!1 "qualification" qualification[])?
                              (!1 "permanent" isPermanent[BOOLEAN])?
                              ;

      UseRelationship ::= "use" "rel" (name['"','"'])?
                              source[] "->" target[]
                              ("("#0 lowerBound[CARDINALITY_LITERAL] "," upperBound[CARDINALITY_LITERAL] #0 ")")?
```

```
                                (!1 "qualification" qualification[])?
                                (!1 "permanent" isPermanent[BOOLEAN])?
                                ;

      ParticipationRelationship ::= "participation" "rel"
(name['"','"'])?
                                source[] "->" target[]
                                ("("#0 lowerBound[CARDINALITY_LITERAL] ","
upperBound[CARDINALITY_LITERAL] #0 ")")?
                                (!1 "qualification" qualification[])?
                                (!1 "permanent" isPermanent[BOOLEAN])?

                                ;
}
```

Anexo II

El CD que se adjunta contiene los siguientes archivos:

- jdk.1.6_b14.exe, que contiene el ambiente de desarrollo de Java
- eclipse.zip, que contiene el ambiente de desarrollo Eclipse versión Ganymede (3.4.1) junto con los plugins implementados.

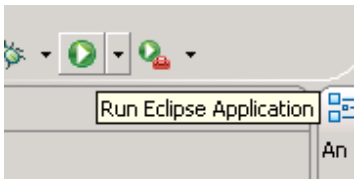
1- Como instalar el ambiente entregado

a. Como primer paso, hay que instalar la máquina Java provista, ejecutando la aplicación jdk.1.6_b14.exe.

b. Como segundo paso, hay que descomprimir el archivo eclipse.zip.

c. Como último paso, hay que ejecutar el archivo eclipse.exe que se encuentra en la carpeta "eclipse". Dentro del workspace se encuentran los plugins desarrollados para el editor colaborativo.

d. Para utilizar el editor colaborativo es necesario arrancar el ambiente de ejecución como se muestra en la siguiente figura.



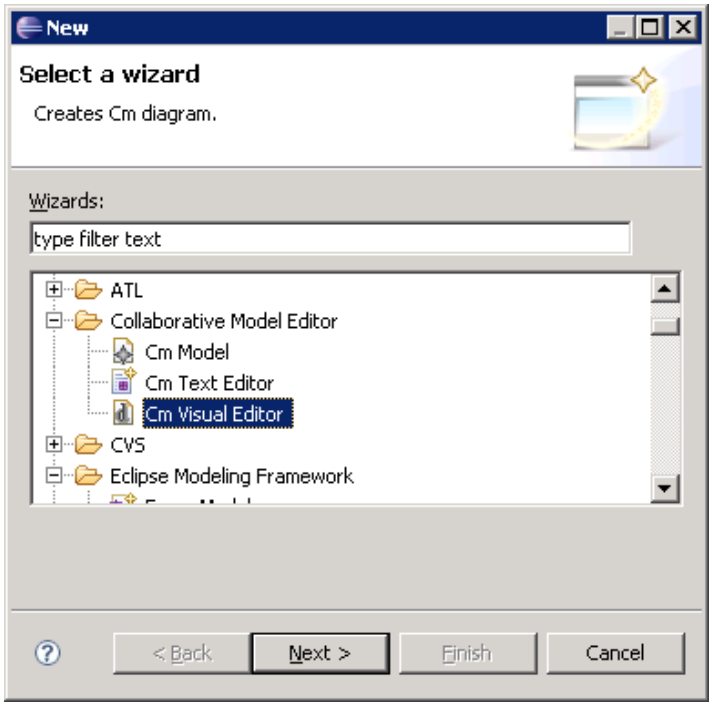
2- Como usar el editor implementado

Para mostrar como se usa el editor, se explicará con un ejemplo. Antes que nada se debe crear un proyecto Java llamado Ejemplo, una vez instanciado el ambiente de ejecución.

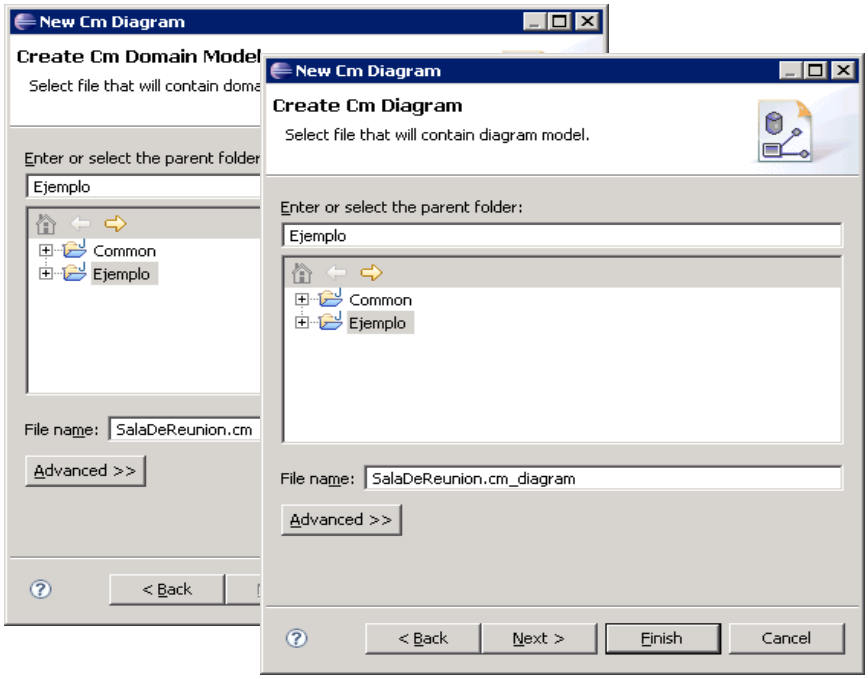
Un modelo colaborativo se puede crear de dos maneras.

a. De forma visual

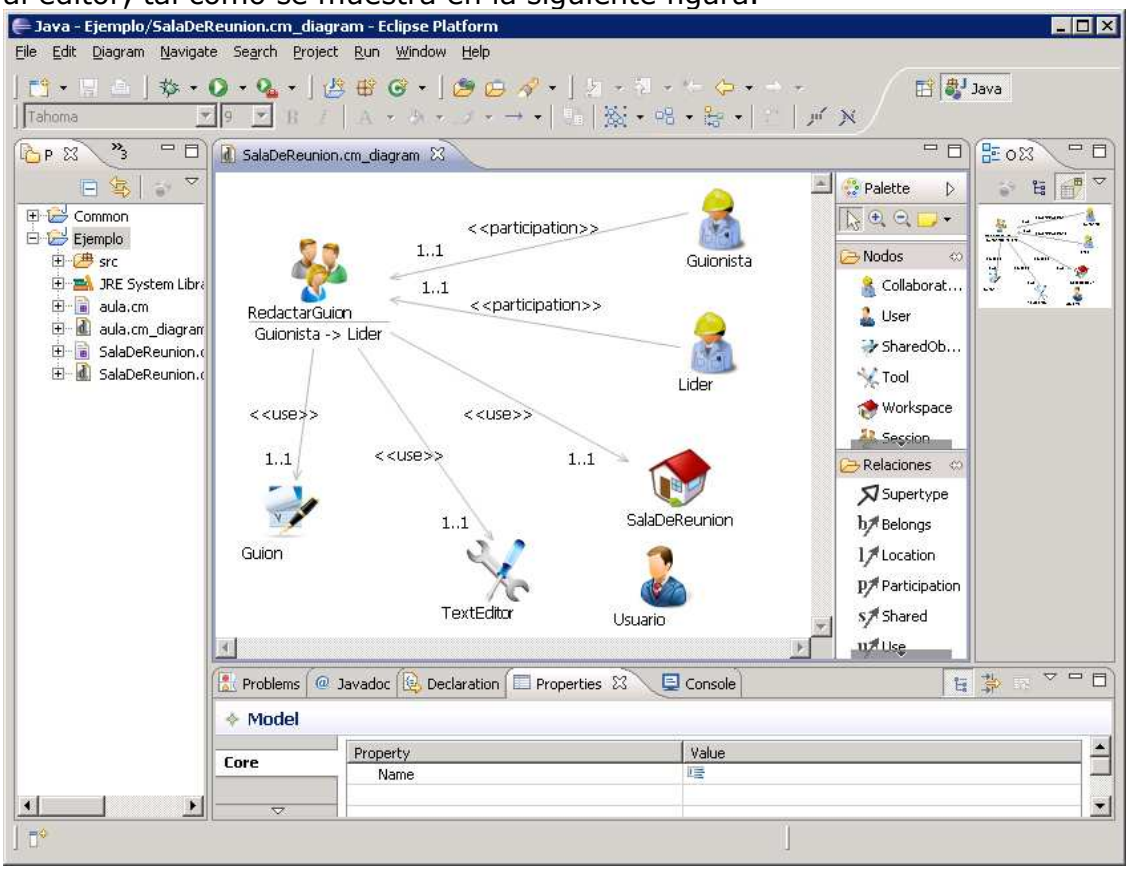
Hay que crear un nuevo diagrama colaborativo. Para hacer esto, el asistente de creación se encuentra en el menú File -> New -> Collaborative Model Editor. Dentro de ese menú, elegir el ítem "Cm Visual Editor" como se ve en la siguiente figura



Al crear un nuevo modelo colaborativo, se generan automáticamente dos archivos. El primero, con extensión *cm_diagram* contendrá la vista del modelo y el segundo, con extensión *cm* contendrá el modelo. Siguiendo el asistente es necesario ingresar los nombres para cada uno de estos archivos, como se muestra en la siguiente figura.



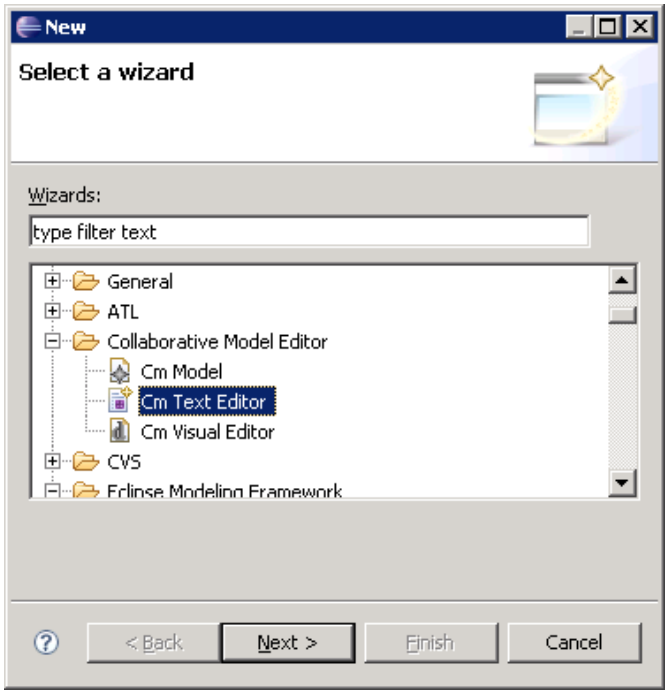
Una vez creado el modelo colaborativo, se abre un editor para instanciar los elementos. Para hacerlo, se deben arrastrar los elementos desde la paleta al editor, tal como se muestra en la siguiente figura.



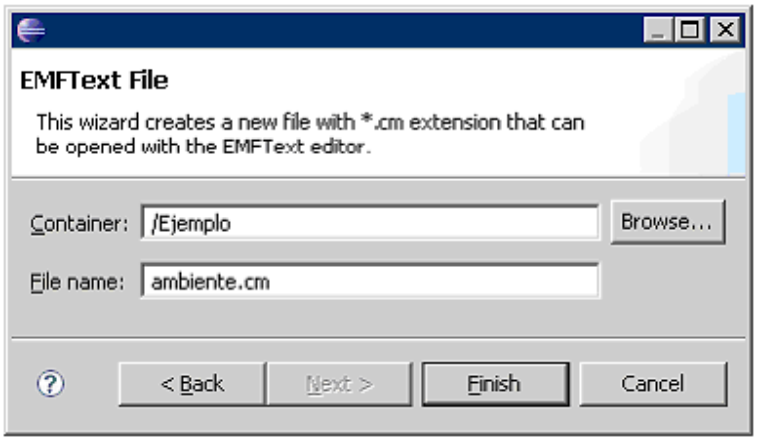
b. De forma textual

Otra manera de instanciar el modelo es utilizando la sintaxis textual. Para abrir el editor de texto hay que seguir los siguientes pasos:

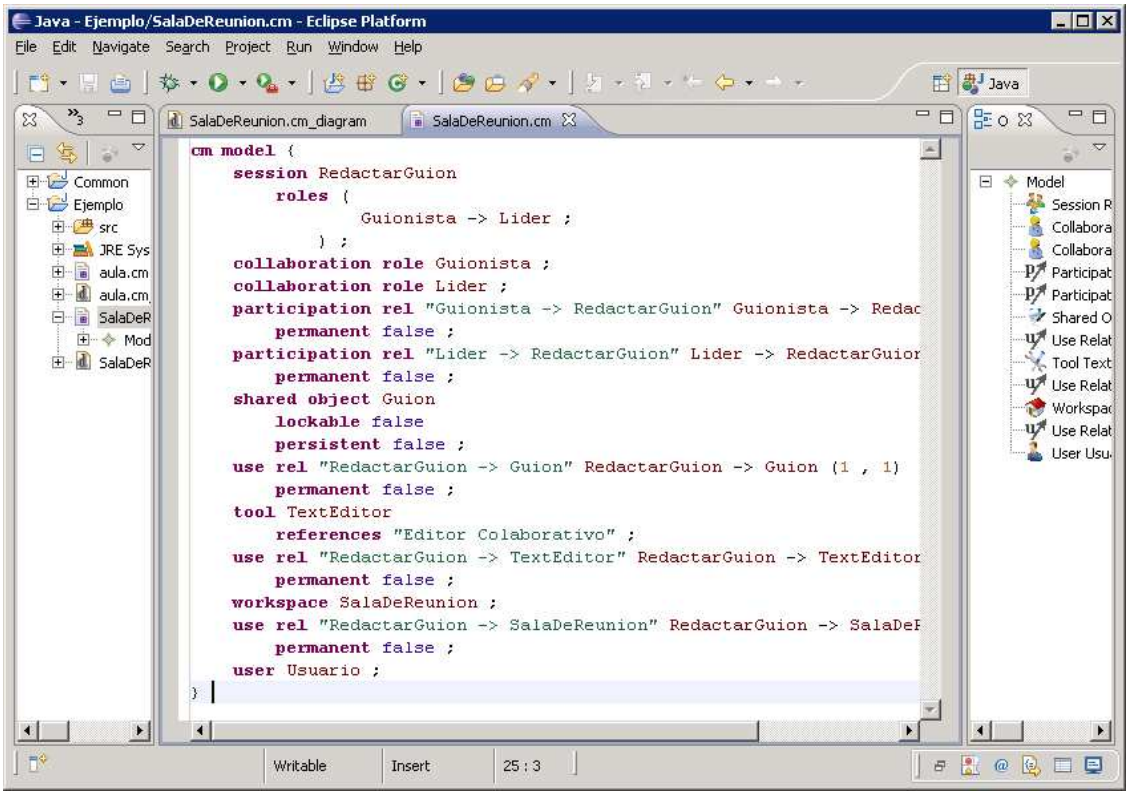
- Crear un nuevo modelo colaborativo. Para hacer eso, el asistente de creación se encuentra en el menú File -> New -> Collaborative Model Editor.
- Elegir el item "Cm Text Editor" como se ve en la siguiente figura



Una vez seleccionado el tipo de archivo, hay que indicar su nombre en el siguiente paso del asistente.



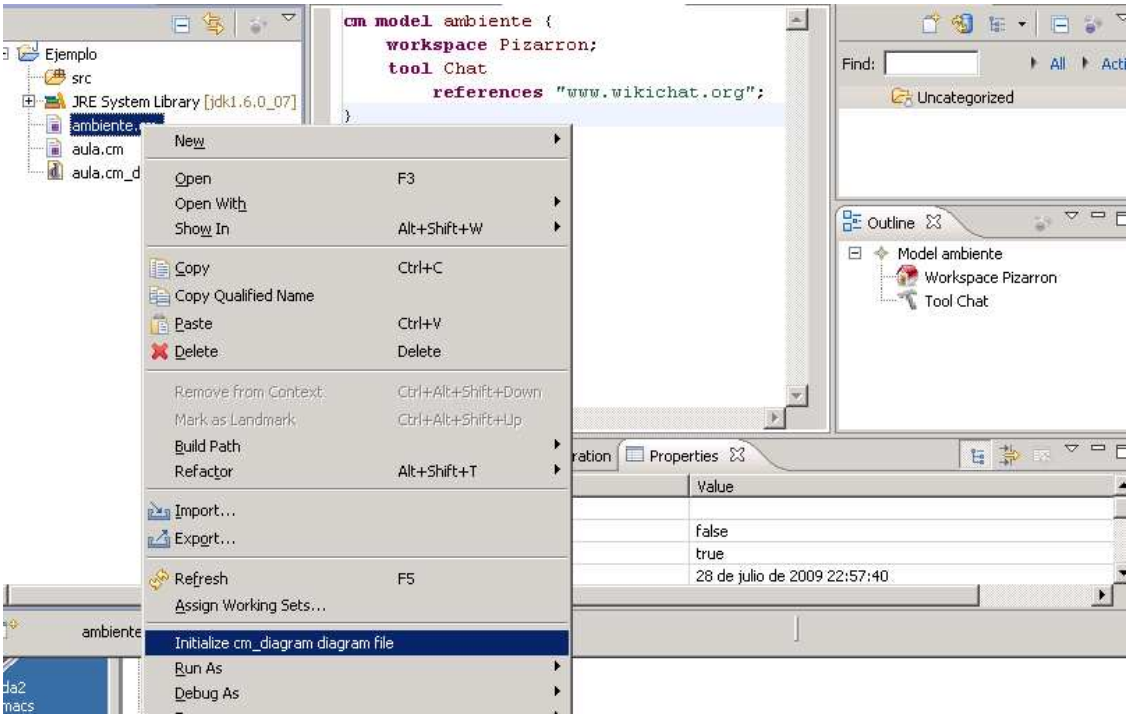
El ejemplo mostrado en el editor visual se podría haber escrito de la siguiente forma



3- Otras funcionalidades

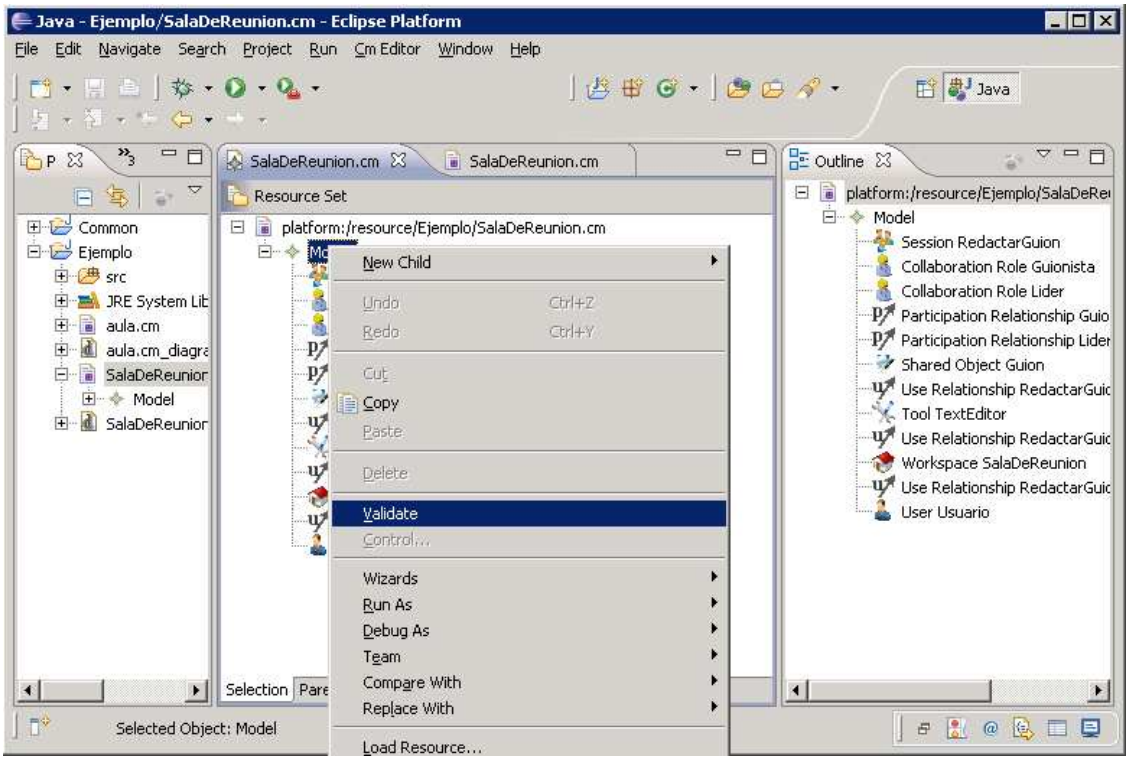
a. Crear una vista a partir del modelo

A partir de un modelo, es posible generar la vista para ese modelo. Para lograr esto, sobre el modelo hay que elegir la opción "Initialize cm_diagram diagram file" que se muestra en el menú contextual que aparece al hacer clic con el botón derecho del mouse, como lo muestra la siguiente figura.



b. Validar la correctitud del modelo

También existe la vista en forma de árbol que muestra los elementos del modelo. A la misma se accede abriendo el archivo cm con la opción "Cm Model Editor" del menú contextual "Open with". En la siguiente imagen se muestra el diagrama, así como la opción para invocar la validación de las reglas OCL definidas.



c. Generar documentación a partir del modelo

Para generar la documentación en formato HTML, hay que ejecutar el archivo de transformación "transformation.m2t" ubicado en el proyecto Common.

Este paso se realiza invocando la opción Execute de la opción MOFScript del menú contextual.