FACULTAD DE INFORMATICA
UNLP

TESINA DE LICENCIATURA

TITULO: Derivación de código a partir de un lenguaje específico de dominio para sistemas colaborativos

AUTORES: Pablo Foresto

DIRECTOR: Luis Mariano Bibbó

CODIRECTOR: Claudia Pons

CARRERA: Licenciatura en Informática

Resumen

Este trabajo busca la generación de código a partir de un lenguaje específico de dominio para sistemas colaborativos llamado CSSL. Por un lado se trata de entender qué artefactos es posible generar a partir de un modelo CSSL, por otro lado se debe decidir una metodología para la generación de código, lo que incluye el análisis de múltiples herramientas. Eso requiere investigación en dos asuntos diferentes, el modelado específico de dominio y los sistemas colaborativos.

El modelado específico de dominio DSM (Domain-Specific Modeling en inglés) 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. Los problemas del desarrollo de software (productividad, calidad, mantenimiento y documentación, etc) son enfrentados por esta metodología, que a través del aumento de nivel de abstracción intenta producir mejoras en los aspectos mencionados anteriormente.

Por otro lado, los sistemas colaborativos son sistemas basados en computadoras que ayudan a un grupo de personas comprometidas en una tarea u objetivo en común, y que proveen una interfaz a un ambiente compartido. Existen características de estos sistemas que pueden ser modeladas a través de un lenguaje específico de dominio, llamado CSSL.

Finalmente, en este trabajo se generan un conjunto de clases Java a partir de un modelo CSSL, con el intento de mostrar como la generación automática de código puede ayudar al desarrollo de aplicaciones colaborativas.

Líneas de Investigación

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

-Ingeniería de Software
-Sistemas Colaborativos
-Modelado específico de Dominio
-Entorno de Desarrollo Eclipse
-Lenguajes específicos de dominio

Conclusiones

Los DSLs permiten un gran aumento de productividad en el desarrollo de software. Esto se logra a través de la automatización del desarrollo, basándose en modelos. CSSL es un lenguaje específico al dominio de los sistemas colaborativos, y podría mejorar la productividad en el desarrollo inicial de estos sistemas. En el trabajo se mostró cómo aprovechar la información contenida en un modelo para generar algunos artefactos, que pueden solucionar algunos problemas en el desarrollo de este tipo de sistemas, ayudados por un framework apropiado.

Trabajos Realizados

Como aportes del trabajo se pueden mencionar:
- Se estudiaron los mecanismos de definición y generación de código de un DSL.
- Se estudiaron herramientas de Eclipse para generar un plugin que recorra modelos y genere código
- Se estudiaron las características de los sistemas colaborativos, y su modelización mediante CSSL
- Se implementó un plugin que es capaz de generar código a partir de un modelo CSSL

Trabajos Futuros

Se plantean las siguientes líneas de trabajo:
- Se puede desarrollar un framework robusto que contenga los conceptos del dominio que describe CSSL, y que si es necesario, se apoye en algún framework existente.
-El dominio es demasiado extenso y CSSL no puede cubrirlo completamente. Esto lleva a dos caminos posibles: reducir el dominio al que apunta la generación o enriquecer el lenguaje.
La reducción del dominio podría permitir la generación de la mayor parte del código de la aplicación.
-Mezclar conceptos de otros dominios en modelos CSSL. Por ejemplo, modelado de interfaces gráficas.

Fecha de la presentación: Octubre de 2010

ÍNDICE

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

En este trabajo nos proponemos estudiar la generación de código en el dominio de los sistemas colaborativos. Se tomará como punto de partida un lenguaje específico de dominio (DSL) usado para describir sistemas colaborativos llamado CSSL ("Collaborative Software System Language"). Dicho lenguaje pretende enfrentar los problemas más importantes que esta tecnología presenta y diseñar ambientes colaborativos que cumplan con ciertas demandas (obtener sistemas groupware integradores, flexibles y altamente productivos). [23]

El desarrollo de código a partir del lenguaje se propone reducir la complejidad tecnológica que implica desarrollar ambientes con estas características. [23]

1.1.1 Sistemas colaborativos

El campo de investigación llamado "Trabajo Cooperativo Asistido por Computadora" (CSCW, "Computer-Supported Cooperative Work") está relacionado con el entendimiento de las interacciones sociales y el diseño, desarrollo y evaluación de sistemas técnicos que asisten la interacción social en equipos y comunidades. En esta definición y en otras cabe destacar que se mencionan dos aspectos: el aspecto tecnológico y el fenómeno social. [25]

Los sistemas resultantes de la investigación y desarrollo en esta materia son llamados "groupware" o sistemas colaborativos. Son sistemas basados en computadoras que ayudan a un grupo de personas comprometidas en una tarea u objetivo en común, y que proveen una interfaz a un ambiente compartido. [25] [26]

Según Koch [25] se distinguen cinco tipos de aplicaciones groupware: soporte para awareness (conciencia de grupo), comunicación, coordinación, equipo y comunidad.

1.1.1.1 Desarrollo de groupware

Según Borges y otros [27], la interacción armoniosa de un grupo depende del entendimiento mutuo, y dicho entendimiento requiere un soporte para:

1. Comunicación entre los participantes
2. Coordinación de sus actividades
3. Una memoria de grupo
4. Conciencia de grupo (awareness)

Existen varios frameworks para el desarrollo de groupware. Algunos de ellos intentan abarcar el soporte de las cuatro características mencionadas, y otros se concentran en una.

Por ejemplo, Borges y Dias [28] presentan un framework para el soporte de awareness en sistemas colaborativos. Dicho framework sólo ataca este tema. Lo hace a través de eventos. El código que lo utiliza debe registrar los eventos que pueden suceder, luego notificar cada una de las ocurrencias de cada evento, y por último el framework pondrá en aviso a los demás usuarios sobre los eventos que correspondan según los filtros aplicados.

Greenberg y Roseman, en su trabajo "Groupware Toolkits for Synchronous Work" [34], sostienen que el desarrollo de software para este tipo de sistema requiere poner atención en:

- El manejo de procesos distribuidos
- La comunicación Inter-Proceso
- La gestión de estado y sincronización de procesos
- El diseño de widgets para groupware
- La creación de administradores de sesión
- El control de concurrencia
- La seguridad.. etc

Basándose en lo anterior sostienen que los frameworks (o "Toolkits") deben proveer las siguientes características:

- **Arquitecturas de tiempo de ejecución**, que manejen automáticamente los procesos, sus interconexiones y las comunicaciones
- **Abstracciones de Programación para Groupware**, utilizables por los programadores para sincronizar los eventos de interacción y los modelos de datos entre procesos, y también las vistas presentadas en cada pantalla.
- **Widgets para Groupware**. Los widgets específicos pueden ser extensiones de los widgets usuales para ser utilizados en aplicaciones groupware o nuevos widgets diseñados específicamente para estos sistemas (por ejemplo la lista de contactos).
- **Administradores de sesión**, que permitan a los usuarios crear sesiones, unirse a ellas, o dejarlas.

El capítulo 2 provee un estudio algo más extenso sobre los desafíos presentes en el desarrollo de groupware.

1.1.2 Tendencias en el desarrollo de software

El trabajo "Un aporte gráfico a las herramientas para transformaciones entre modelos" ([9]), cita algunos problemas del desarrollo de software:

- *el problema de la productividad*: los documentos se producen en las primeras fases. Estos pueden ser requisitos escritos en lenguaje natural, dibujos, diagramas UML, etc. El problema radica en que, a medida que el sistema cambia, la congruencia entre el código y los documentos iniciales se pierde. Mantener actualizados dichos documentos es costoso. Pero los problemas comienzan cuando se cambia parte del equipo de desarrollo. Si los documentos iniciales no están actualizados las modificaciones son costosas.
- *El problema de la portabilidad*: cada año aparecen nuevas tecnologías. Las compañías necesitan aplicar estas nuevas tecnologías. El software cambia a una nueva tecnología y las anteriores pierden valor.
- *El problema de la interoperabilidad*: los sistemas necesitan comunicarse con otros, de otras tecnologías.

- *El problema del mantenimiento y la documentación*: en sistemas complejos la documentación en un nivel más alto de abstracción es imprescindible, y según lo ya explicado es difícil mantenerla actualizada con respecto al código.

1.1.2.1 MDD

El "Desarrollo Conducido por Modelos" (en inglés "Model Driven Development", MDD) promete mejorar la construcción de software basándose en un proceso guiado por modelos y soportado por potentes herramientas. La transformación entre modelos constituye el motor del MDD. Estos modelos son sucesivamente transformados, comenzando por modelos mas abstractos e independientes de la plataforma y con cada transformación se obtienen modelos más específicos, hasta llegar al código ejecutable. [9]

1.1.2.2 Lenguajes específicos de dominio

El libro "Domain-Specific Modeling: Enabling Full Code Generation" ([3]) subraya la importancia de que en el desarrollo manejado por modelos tanto el lenguaje como el generador sean específicos de dominio.

Para levantar el nivel de abstracción los lenguajes necesitan estar al tanto del dominio. De esa manera la generación total de código puede ser real. Con lenguajes de modelado de propósito general esto es casi imposible. Si fuera posible, la última década habría mostrado cientos de casos exitosos.

Existen ejemplos de lenguajes específicos de dominio que muestran que se mejora la producción con este enfoque. [3]

1.1.2.2.1 ¿Qué tan apropiado puede ser un DSM para sistemas colaborativos?

Sin embargo, se debe tener en cuenta que, por lo general, compartir la misma solución DSM con otra compañía en el mismo dominio no es posible. Los conceptos básicos son comunes, pero no los detalles. Además, los dominios no deben ser muy generales. Un banco completo, un auto, o una televisión son dominios demasiado extensos, y las soluciones DSM deben reducir esos dominios. De esto sale la primer gran pregunta de esta tesis: ¿Qué tan apropiado puede ser un DSM para sistemas colaborativos? ¿Es posible la generación total de código, o se necesitará algo de reescritura del código generado?

1.1.2.3 CSSL ("Collaborative Software System Language")

CSSL es un lenguaje específico de dominio, diseñado para representar conceptos del dominio de los sistemas colaborativos. La especificación de la sintaxis del lenguaje gráfico es el metamodelado. El metamodelo que describe a este lenguaje ([13]) está basado en un subconjunto de UML, al cual extiende con conceptos de este dominio. A continuación se describen los elementos modelados.

1.1.2.3.1 Elementos específicos del dominio

El dominio presenta ciertos elementos y hechos particulares. Por ejemplo existen:

- Herramientas
- Usuarios

- Roles
- Objetos compartidos: que pueden estar centralizados o distribuidos entre los sitios de los usuarios.
- Sesiones colaborativas
- Espacios de trabajo
- Relaciones de Ubicación
- Relaciones de Participación de Rol en Sesión
- Relaciones de Uso
- Operaciones

Algunas interacciones de estos elementos en un sistema colaborativo:

- Un usuario realiza operaciones. Dichas operaciones se envían a las sesiones a través de herramientas.
- Las sesiones contienen un conjunto de usuarios, que interactúan sobre algunos objetos compartidos a través de herramientas.
- Las herramientas solo se comunican con sesiones a través de operaciones.
- La sesión es quien, por lo general, contiene a los objetos compartidos.
- Si un objeto es compartido por mas de una sesión, tendría sentido que fuera referenciado por el espacio de trabajo.
- Los objetos compartidos son mostrados en las aplicaciones de los usuarios a través de las herramientas conectadas a la sesión.

1.1.2.3.2 ¿Que cosas son particulares a cada aplicación groupware?

Las operaciones realizadas sobre los objetos compartidos son totalmente distintas según la aplicación. Por esa razón resulta indispensable que la generación de código permita extender el código generado.

CSSL es demasiado genérico y es aplicable tanto a groupware sincrónico como asincrónico. Además, tampoco tiene manera de especificar la resolución de conflictos ante cambios concurrentes sobre los mismos objetos. Dada esta situación, la generación de código desarrollada en este trabajo, como sugiere [3] (3.1.1, "Narrow Focus"), opta por reducir el dominio. La generación de código sólo es válida para groupware sincrónico, la arquitectura es "hub-and-spoke" y la ordenación de eventos es por llegada al servidor. No hay operaciones concurrentes, existe un orden total. Para cambiar la arquitectura por una distribuida se puede escribir un segundo generador de código, que podría usar alguno de los algoritmos de transformaciones operacionales para ordenar los eventos y resolver conflictos.

1.1.3 Objetivo de la tesis

El objetivo principal de la tesis es la generación de código a partir de el metamodelo del lenguaje específico de dominio llamado "Collaborative Software System Language" (CSSL).

Entre las tareas requeridas para tal fin se encuentra la de investigar las alternativas y contingencias a tener en cuenta en la generación de código. Por un lado se trata de entender qué artefactos es posible generar a partir de un modelo CSSL, por otro lado se debe decidir una metodología para la generación de código, lo que incluye el análisis de múltiples herramientas. Finalmente está el objetivo de generar código a partir de un modelo.

Para ello se deben tomar dos aspectos de la generación:

1. Qué aspectos del sistema groupware serán simplificados con la herramienta.
2. Cómo se implementará

El metamodelo de CSSL está implementado usando EMF (Eclipse Modelling Framework). En pocas palabras, EMF es la implementación que provee la IDE Eclipse del estándar de metamodelado MOF (Meta Object Facility).

EMF provee la capacidad de leer, escribir e intercambiar modelos, ya sea mediante una API o su IDE genérica.

EMF nos permitirá generar código Java que represente a cada elemento del metamodelo. Se analizarán diferentes estrategias de generación.

Una de las estrategias será utilizar la capacidad de EMF para leer el metamodelo serializado en XMI y utilizar la herramienta JET (Java Emitting Templates) para la generación de código. Esto es lo que hace EMF para la generación de código de los metamodelos.

Otra alternativa a analizar es la generación de código usando Mofscript.

En una segunda generación de código, y para evitar sobreescribir posibles modificaciones sobre el código generado, se utilizará JMerge. Esto permite que existan en el código las llamadas "regiones protegidas", que no son otra cosa que regiones de código marcadas como "modificadas manualmente por el usuario".

Se espera que el código generado tenga resuelto:

- Las estructuras estáticas de la aplicación (Los objetos que representan a usuarios, grupos, sesiones, objetos compartidos).
- Funcionalidad básica (login, etc…)
- La comunicación entre los clientes de algunos eventos.

No se espera:

- Que genere interfaz gráfica. Si bien existirán algunas herramientas implementadas en el framework subyacente, y están representadas la herramientas que utiliza una sesión, no se espera que los objetos generados *sean* la interfaz gráfica. Sin embargo, luego del agregado de código manual, pueden ayudar a la interfaz gráfica escrita manualmente.

2 INTRODUCCIÓN A MDD

2.1 Visión General

La aplicación de modelos al desarrollo de software es una larga tradición, y se hizo aún más popular con el desarrollo de UML (Unified Modelling Language). Sin embargo, a menudo se trabaja con simple documentación, porque la relación entre el modelo y la implementación es simplemente una intención, no es formal. A este tipo de uso de modelos, en un proceso de desarrollo, lo llamamos "basado en modelos" ("model-based")[**12**].

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. Dentro del Desarrollo Dirigido por Modelos, existen varios enfoques. La OMG propone un grupo de estándares y tecnologías con su Arquitectura Dirigida por Modelos. Otro enfoque es el Modelado específico de Dominio.

MDA es el acrónimo de Model Driven Architecture (Arquitectura Dirigida por Modelos), un concepto promovido por la OMG que propone basar el desarrollo de software en modelos. De esta forma, a partir de esos modelos, se podrán realizar trasformaciones que generen código u otros modelos con características de una tecnología particular (o con menor nivel de abstracción). El punto clave en MDA es la importancia de los modelos en el proceso de desarrollo de software. Como se ve, esto suena parecido a MDD. Sin embargo, MDA tiende a ser más restrictivo, centrándose en lenguajes de modelado basados en UML. El primer objetivo de MDA es la interoperabilidad entre herramientas y la estandarización a largo plazo de modelos para dominios de aplicación populares. En cambio MDD apunta a la provisión de módulos para el desarrollo de software que son aplicables en la práctica, independientemente de la herramienta seleccionada o de la madurez del estándar de MDA.[**12**]

Por otro lado, los defensores de Modelado Específico de Dominio sostienen que para levantar el nivel de abstracción los lenguajes necesitan estar al tanto del dominio. De esa manera la generación total de código puede ser real. Con lenguajes de modelado de propósito general (por ejemplo UML) esto es casi imposible, sino imposible.

Estos enfoques se verán con mayor detalle más adelante.

Lo primero que se debe hacer es determinar las razones por las que surge esta tendencia en el desarrollo de software. En primer lugar plantearemos tres objetivos del desarrollo de software (calidad, productividad, longevidad) y veremos como

esas variables evolucionaron con el tiempo.
Luego analizaremos las distintas funciones de los modelos en el desarrollo de software, y su relación con el código generado.

2.1.1 Historia de los avances en el desarrollo de software

La viabilidad de la industria del software está determinada por hasta qué punto podemos producir sistemas cuya calidad y longevidad estén de acuerdo con su costo de producción. Construir software de alta calidad y duradero es caro. A raíz de ello, a veces nos vemos forzados a hacer concesiones a la calidad y la longevidad a favor del costo de producción.

Es difícil mejorar la viabilidad de un proyecto mejorando las tres variables antes mencionadas. Aunque, curiosamente, podemos ver la historia del desarrollo como una serie de mejoras que cambiaron la ecuación de la viabilidad. Frankel, en su libro 'Applying MDA to Enterprise Computing' ([30]), pone énfasis en los siguientes hechos y eras:

2.1.1.1 Computación con centro en la máquina

En los primeros tiempos los programadores codificaban las instrucciones en la computadora como unos y ceros, armando manualmente las instrucciones del procesador.

Luego, se invento el lenguaje ensamblador. Le permitió a los programadores usar nombres mnemotécnicos para las instrucciones y registros. Además, por primera vez, permitió asociar un nombre a una dirección de memoria.

Otra ventaja que trajo el ensamblador fue que los programas eran menos sensibles a pequeños cambios incrementales en los unos y ceros que componían una instrucción en una línea de procesadores.

Cabe subrayar que fue el **aumento del nivel de abstracción** lo que permitió el cambio en las tres variables (costo de producción, calidad, longevidad).

2.1.1.2 Computación con centro en la aplicación

El advenimiento de los **lenguajes de tercera generación (3GL)** disparó un gran salto en la productividad. Otra vez, los primeros 3GLs **aumentaron el nivel de abstracción** muy por encima de los conceptos del juego de instrucciones del procesador. Una simple instrucción podía reemplazar cientos de líneas de lenguaje ensamblador.

Esta vez, la transición entre tecnologías sucedió con algunas objeciones de los desarrolladores. Existían quejas porque los primeros compiladores introducían errores en el código y producían código menos óptimo que el ensamblador.

Los cambios en los procesadores dejaron de dejar obsoletos a los programas. Esto se conoció como *portabilidad*. Por lo tanto se aumenta la *longevidad* del programa.

También se mejora la *calidad*, porque escribir menos líneas de código para realizar una función la hace más manejable intelectualmente.

Al mismo tiempo que los 3GLs levantaban el nivel de abstracción del ambiente de programación, los sistemas operativos levantaron el nivel de abstracción de la plataforma de computación.

Otra vez las tres variables de la viabilidad aumentaron en la misma dirección, y eso posibilitó la aparición de nuevas clases de programas, que de repente resultaron viables.

Según Kelly y Tolvanen **([3]**), los lenguajes de tercera generación aumentaron la productividad en un sorprendente 450%. Dicho libro sostiene que luego de este gran cambio no hubo ningún otro avance en la industria que produjera tal aumento de productividad. Por supuesto, sin tomar en cuenta el modelado específico al dominio, un concepto similar a MDD y que es tratado por esa obra.

Se enfatiza en el aumento del nivel de abstracción como medio para aumentar la productividad y calidad del software porque el objetivo de esta tesis es lograr eso mismo a través de un lenguaje específico de dominio. Más adelante se tratará el tema.

2.1.1.2.1 Lenguajes Orientados a Objetos y Máquinas Virtuales

Los lenguajes estructurados de tercera generación evolucionaron hacia lenguajes de tercera generación Orientados a Objetos. Estos hicieron más fácil el reuso de código en distintos contextos. Algunos de estos lenguajes introdujeron un intérprete de código intermedio, llamado Máquina Virtual ("Virtual Machine", abreviado VM).

2.1.1.3 Computación con centro en la Empresa

En la era de la computación centrada en la empresa existían islas de automatización, que repetían y reimplementaban lógica de negocio. Para evitar esa reinvención constante de la rueda, se dieron algunos avances que se enumeran y describen a continuación.

Lo importante a notar en los avances que se tratan a continuación es el esfuerzo constante de la industria del software por elevar el nivel de abstracción.

2.1.1.3.1 Desarrollo basado en Componentes

Un componente es un módulo de software que puede ser empaquetado.

La componentización evita reinventar la misma solución para diferentes aplicaciones. Aumenta la productividad al reducir el costo de producción. También la calidad, ya que se actualiza y prueba la funcionalidad en un solo lugar.

2.1.1.3.2 Patrones de diseño

Son una importante contribución a la mejora de la productividad y calidad del desarrollo de software. Los programadores pueden reusar patrones de diseño comunes que ya fueron pensados y validados por otros.

2.1.1.3.3 Computación Distribuida

En los orígenes de la industria, el control total estaba a cargo de un único sistema operativo en una computadora central, llamada master ("amo"), mientras que los otros nodos de procesamiento eran llamados esclavos ("slaves").

Con el avance del tiempo, y especialmente con la llegada de las computadoras personales, se hizo claro que buena parte del procesamiento podía ser descargada hacia los esclavos.

Con el tiempo, la arquitectura cliente-servidor fue evolucionando gradualmente hacia el paradigma *peer-to-peer* ("par a par" o "punto a punto"), donde cualquiera de los nodos puede ser el servidor según el contexto.

2.1.1.3.4 Middleware

Inicialmente, la computación distribuida era manejada por código propietario o del cliente. Pero gradualmente se fue reemplazando por sistemas de propósito general, llamado *middleware*.

CORBA, J2EE, .NET y MOM ("Message Oriented Middleware") son ejemplos de plataformas middleware que **proveen servicios más potentes que aquellos de los sistemas operativos**.

El middleware aumenta el nivel de abstracción de la plataforma, y en algunos casos también de la programación. Proveen servicios independientes del sistema operativo, por ejemplo el sistema de concurrencia de CORBA.

2.1.1.3.5 Especificación Declarativa

Se trata de programar un sistema completando valores de algunas propiedades. Es decir, sin usar instrucciones procedurales, como en la programación imperativa.

Los ejemplos más usados son SQL y el diseño de GUIs. En SQL se define el formato de los registros que se necesitan, y es el motor de la base de datos quien se encarga de administrar los recursos para conseguir esos datos.

2.2 Desarrollo Dirigido por Modelos

Como se ejemplificó anteriormente, durante la historia del desarrollo de software, los desarrolladores siempre buscaron mejorar la productividad mejorando la abstracción.

Hoy en día, sin embargo, los lenguajes tradicionales de programación y los lenguajes de modelado contribuyen poco a la mejora de productividad, si se compara con el salto desde el lenguaje ensamblador hacia los lenguajes de tercera generación.

En ese sentido, tanto MDA como DSM buscan mejorar la productividad, aunque no son exactamente lo mismo.

2.2.1.1 Código y Modelo

Los desarrolladores por lo general diferencian entre el modelado y la codificación. Los modelos se usan para diseñar el sistema, entenderlo mejor, especificar funcionalidad y documentar. Para implementar el diseño, depurarlo, probarlo y mantenerlo se utiliza el código. Normalmente estos medios son vistos como desconectados. Pero existen varias maneras de alinear el código y los modelos.

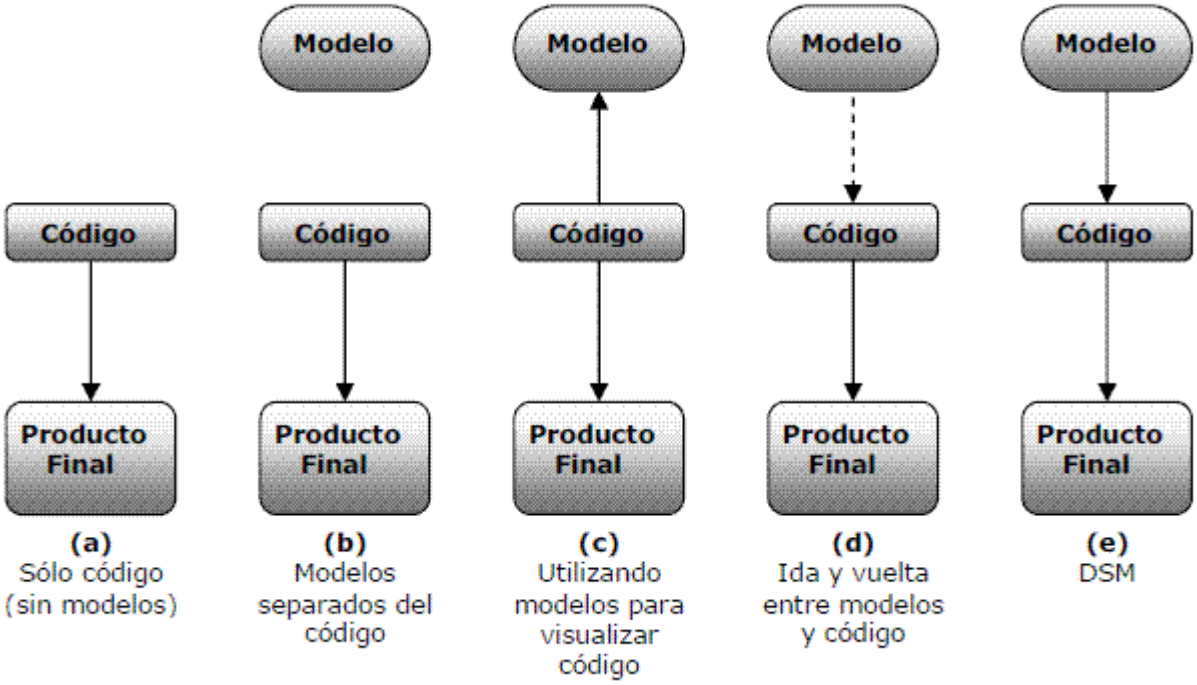


Ilustración 2-1.Relación entre código y modelo

En la figura se pueden ver las distintas relaciones entre código y modelo.

1. La primera posibilidad es ***no modelar***. Es aceptable para pequeños desarrollos.
2. La segunda opción es modelar, y luego usar esos modelos para basarse en la construcción del software. ***La conexión entre el los diagramas y el código se pierde gradualmente*** mientras se progresa en la fase de codificación. Hasta el punto en que se convierten en dibujos con poca o ninguna relación, en vez de ser una especificación exacta del código. Los programadores suelen hacer los cambios sólo en el código, porque no hay tiempo para actualizar los diagramas y documentos de alto nivel. De estos problemas surge la metodología "Extreme Programming" (XP)**[31]**. Esta metodología se volvió popular rápidamente, debido a que reconoce al código como la fuerza impulsora del desarrollo de software.
3. Volviendo a la figura, la tercer opción muestra como los modelos pueden ser usados en **ingeniería inversa**. Es decir, intentar entender el software luego de haberse diseñado y construido.
4. La cuarta posibilidad es **round-tripping** (que se puede traducir como "viaje de ida y vuelta"). Apunta a automatizar el trabajo de tener la misma información en dos lugares, modelo y código. Funciona cuando ambos formatos son similares y no hay pérdida de información en la traducción. Un

ejemplo son los diagramas de bases de datos. Cabe observar que en esta aproximación no hay una elevación del nivel de abstracción, no hay ocultamiento de información (no se ocultan los detalles menos importantes). Es simplemente una representación extra de la misma información.

5. En la quinta opción, correspondiente al **modelado específico de dominio**, parte del **desarrollo dirigido por modelos**, los modelos son los artefactos principales en el proceso de desarrollo. Se tienen modelos en lugar de código fuente. Una vez terminados los modelos, estos pueden ser llevados a código fuente a través de transformaciones. La situación ideal es que el código generado no necesite ser modificado antes de ser ejecutado. Esto será analizado con mayor detenimiento en la implementación del generador.

2.3 Resumen

Existen tres objetivos primordiales en el desarrollo de software (calidad, productividad, longevidad). Durante el desarrollo de la historia de la computación, pocas veces se logró un aumento significativo de estas tres variables al mismo tiempo. Y eso se logró de una única manera: **el aumento del nivel de abstracción**.

La función de los modelos es, justamente, aumentar el nivel de abstracción de un sistema. Pero muchas veces los modelos tienen un uso limitado en el desarrollo. Normalmente, los modelos y el código son entidades totalmente separadas. Muchas veces los modelos pierden validez al terminar el desarrollo, porque nadie los actualiza. Por eso comienzan a surgir alternativas para darle mayor importancia a los modelos en el desarrollo, que sean la fuente de información fundamental para derivar el código.

3 ARQUITECTURA DIRIGIDA POR MODELOS

3.1.1 Introducción

La Arquitectura Dirigida por Modelos (MDA) es un framework para desarrollo de software definido por el "Object Management Group" (OMG).

El ciclo de vida del desarrollo no es muy distinto al tradicional, pero se diferencia por los artefactos generados en cada fase.

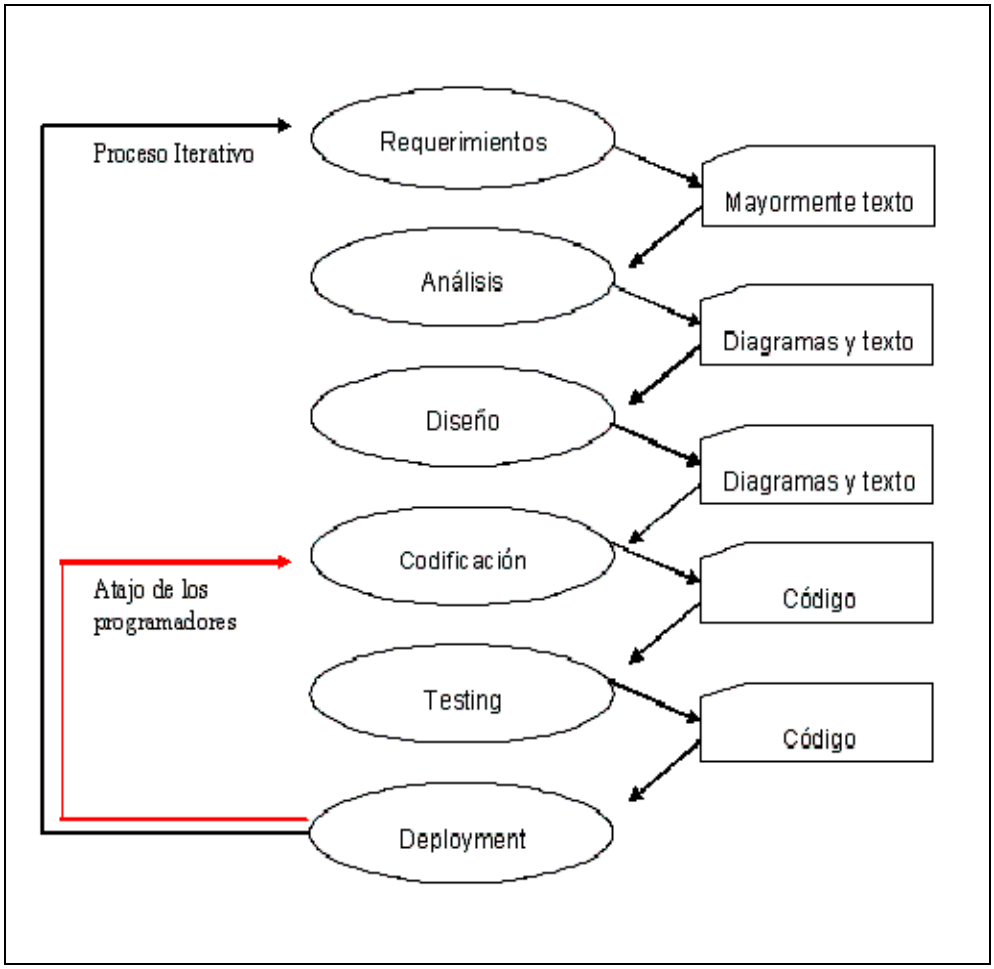


Ilustración 3-2. Ciclo de vida tradicional del desarrollo de software

1. **Modelo Independiente de la plataforma**: es un modelo con un alto nivel de abstracción que es independiente de cualquier tecnología o lenguaje de implementación. Se lo llama PIM o *"Platform Independent Model"*.

2. **Modelo específico de la plataforma**: Como siguiente paso, un PIM se transforma en uno o más PSM (*"Platform Specific Models"*). Un PIM representa la proyección de los PIMs en una plataforma específica.
3. **Código**: El paso final en el desarrollo es la transformación de cada PSM a código. Ya que el PSM habla de elementos del código de un lenguaje especifico, la transformación es relativamente directa.

Tradicionalmente, las transformaciones de modelo a modelo, o de modelo a código, son hechas, normalmente, con intervención humana. En contraste, las transformaciones MDA son siempre ejecutadas por herramientas.

Muchas herramientas pueden transformar un PSM a código; no hay nada nuevo en eso. Dado que un PSM es un modelo muy cercano al código, esta transformación no es demasiado compleja. Lo nuevo que propone MDA es que la transformación de un PIM a PSMs sea automatizada. Ese es el principal beneficio de MDA.

Los beneficios de la aplicación de MDA en el desarrollo de software son:

- **Productividad:** pues el foco del desarrollo está puesto en el desarrollo de un PIM. Los PSMs se derivan automáticamente a través de una transformación. Por lo tanto, los desarrolladores del PIM tienen menos trabajo que hacer, ya que los detalles específicos de la plataforma no necesitan ser diseñados. Además, durante el mantenimiento, introducir modificaciones sobre los modelos es mucho más rápido y seguro que hacerlo sobre el código final.
- **Portabilidad:** El mismo PIM puede ser automáticamente transformado en muchos PSMs para diferentes plataformas. Por lo tanto, todo lo que se especifica en el nivel de PIM es completamente portable.
- **Interoperabilidad:** Entre los distintos PSMs generados a partir del mismo PIM también se pueden generar automáticamente *bridges* (puentes) que permitan la interoperabilidad de esos PSMs.
- **Mantenimiento y documentación:** Con MDA los desarrolladores pueden enfocarse solamente en el PIM, el cual tiene un nivel más abstracto que el código. Como el código es generado luego de una serie de transformaciones, el modelo será una exacta representación del código. Por lo tanto, el PIM cumplirá la función de documentación de alto nivel.

3.1.2 La arquitectura de cuatro capas de modelado

Hace algunos años, los lenguajes se definían, normalmente, usando la gramática "Backus Naur Form" (BNF), en la cual se describe qué 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 necesariamente deben estar basados en texto, y por lo general no lo están (ya que tienen una sintaxis gráfica, como UML) se necesita un mecanismo diferente para definirlos. Este mecanismo de definición se llama metamodelado.

Usando un lenguaje de modelado, podemos crear modelos; un modelo especifica qué elementos pueden existir en un sistema. Por otro lado, la definición de un lenguaje de modelado especifica qué elementos pueden existir en un modelo. Debido a esta similitud, se puede describir un lenguaje por medio de un modelo,

usualmente llamado metamodelo. El metamodelo de un lenguaje describe qué elementos pueden ser usados en el lenguaje.

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, M0:

3.1.2.1 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.

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 se muestra un diagrama de objetos UML donde puede verse las distintas entidades que almacenan los datos necesarios para este sistema

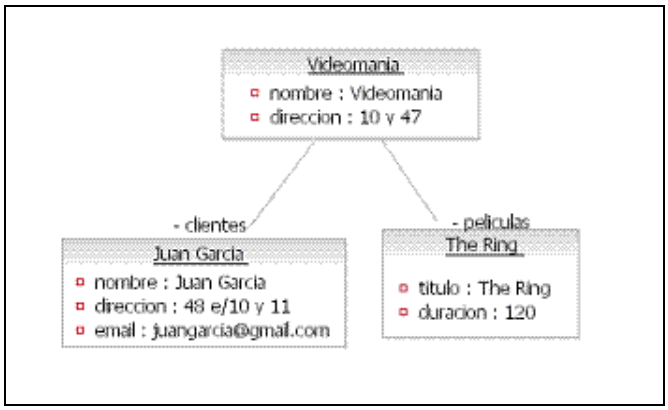


Ilustración 3-3. Entidades de la capa M0 del modelo de cuatro capas

3.1.2.2 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.

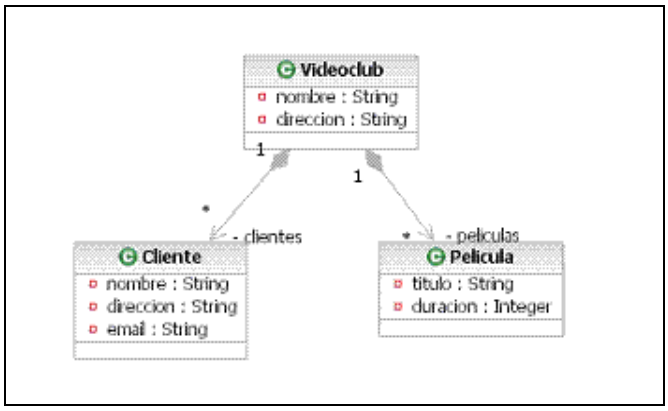


Ilustración 3-4. Modelo del sistema

Siguiendo el ejemplo, 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.

3.1.2.3 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 muestra una parte del metamodelo UML. En este nivel aparecen conceptos tales como Clase, Atributo o Asociación.

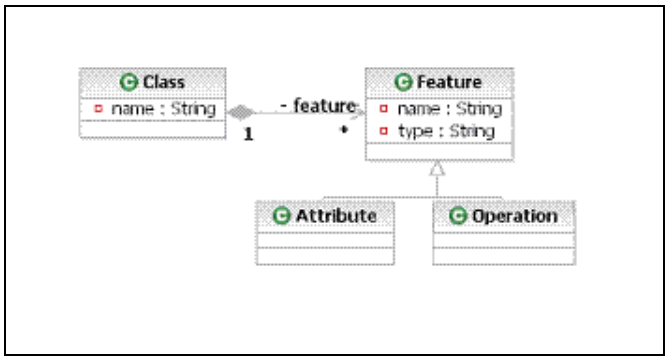


Ilustración 3-5. Parte del metamodelo UML

La siguiente figura muestra la relación entre los elementos del nivel M1 con los elementos del nivel M2.

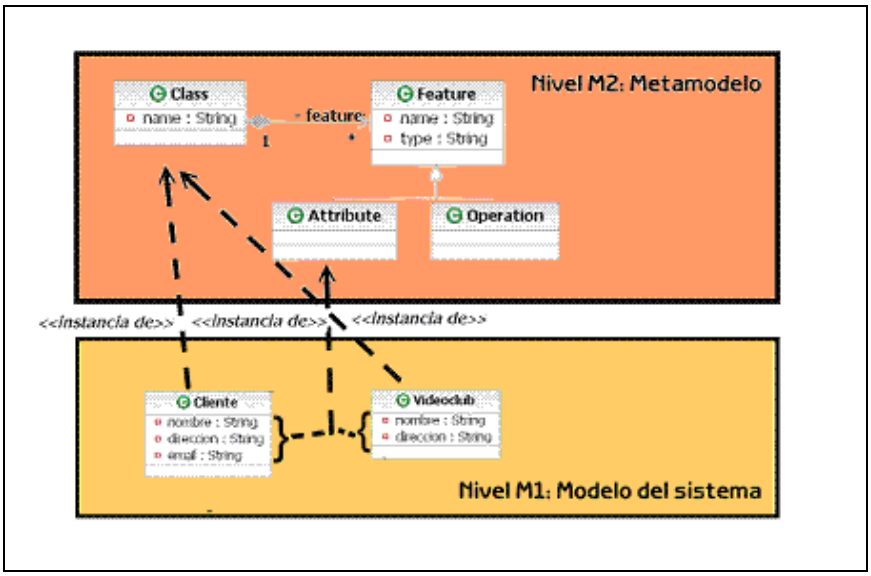


Ilustración 3-6. Relación entre los elementos del nivel M1 con los elementos del nivel M2

3.1.2.4 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. UML es un lenguaje definido a través de MOF, pero también se pueden definir metamodelos de lenguajes no orientados a objetos.

La figura 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.

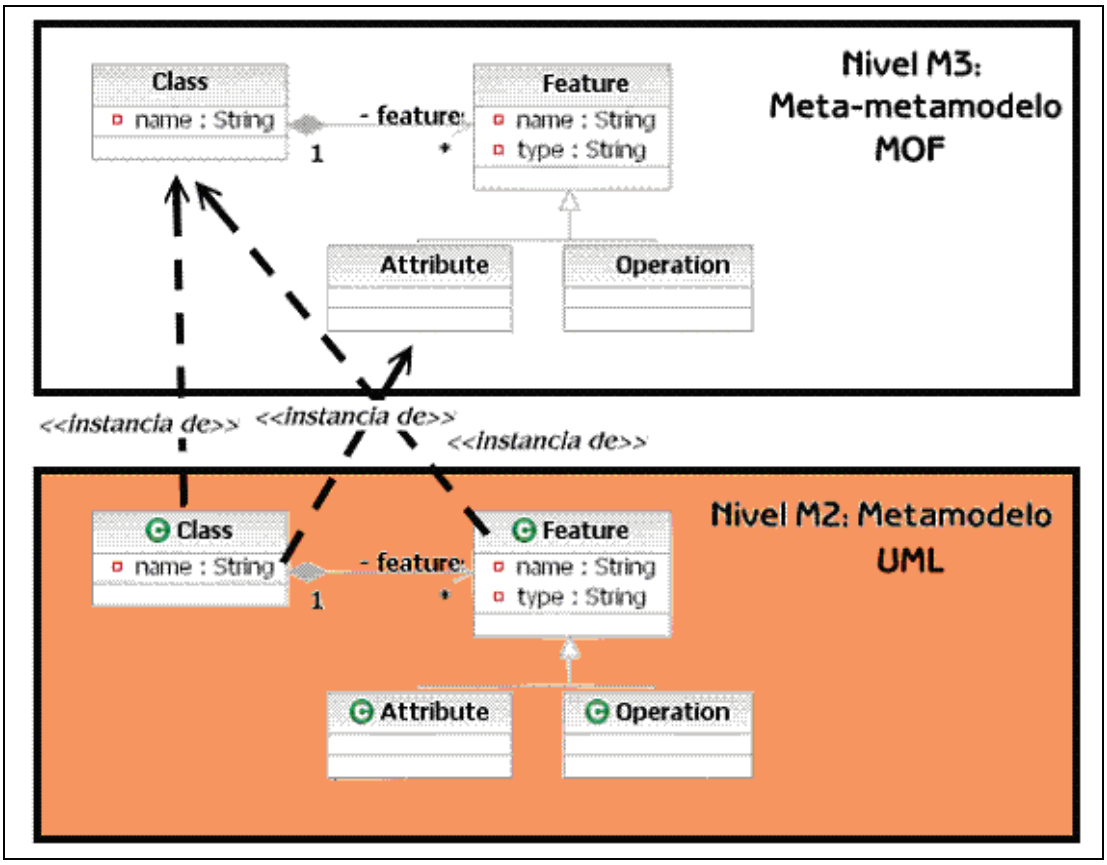


Ilustración 3-7. Relación entre el nivel M2 y el nivel M3

3.1.3 Estándares de la OMG

En el núcleo del concepto de MDA hay un número de importantes estándares de la OMG. Estos son:

- UML ("Unified Modeling Language")
- MOF ("Meta Object Facility")
- XMI ("XML Metadata Interchange")
- CWM ("Common Warehouse MetaModel")

Estos forman la base para la construcción de esquemas coherentes de composición, publicación y administración de modelos dentro de MDA.**[32]**

3.1.3.1 MOF

El lenguaje MOF, acrónimo de "Meta-Object Facility", es un estándar de la OMG para la ingeniería conducida por modelos. MOF se encuentra en la capa superior de la arquitectura de 4 capas. Provee un meta-meta lenguaje en la capa superior que permite definir metamodelos en la capa M2. El ejemplo más popular de un lenguaje en la capa M2 es el metamodelo UML, que describe el lenguaje UML.

Actualmente, la definición de MOF esta separada en dos partes fundamentales, EMOF ("Essential MOF") y CMOF ("Complete MOF"), y se espera que en el futuro se agregue SMOF ("Semantic MOF"). Ambos paquetes importan los elementos de un paquete en común, del cual utilizan los constructores básicos y lo extienden con los

elementos necesarios para definir metamodelos simples, en el caso de EMOF y metamodelos mas sofisticados, en el caso de CMOF.

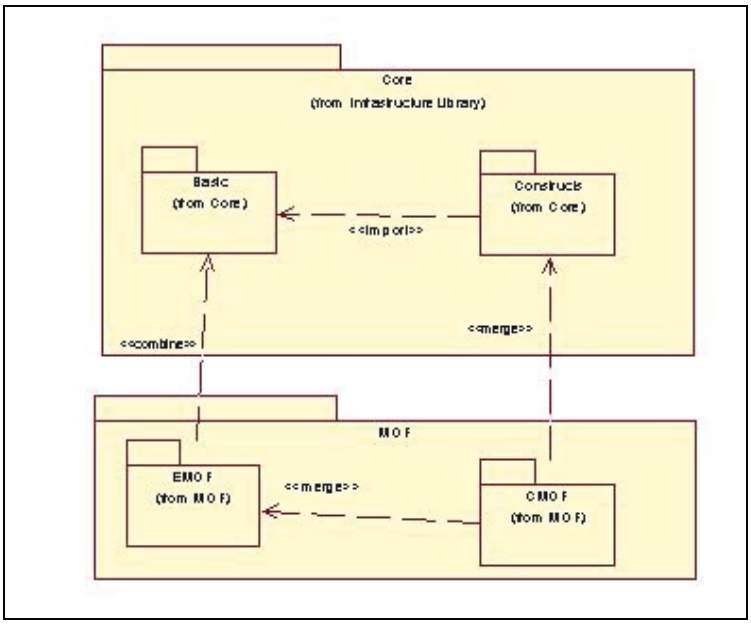


Ilustración 3-8 . Partes de la definición de MOF

En el caso del lenguaje utilizado en esta obra, CSSL, su metamodelo está definido usando MOF. En particular, se utiliza una implementación de EMOF para el entorno de desarrollo Eclipse llamada Ecore.

3.1.3.2 UML

Una definición simplificada e informal de UML podría ser: "una familia de notaciones gráficas, respaldadas por un único metamodelo, que ayuda a describir y diseñar software, particularmente software construido usando un estilo Orientado a Objetos." [6]

Existen varias *maneras de usar UML*:

- *Como un lenguaje para realizar bocetos* de lo que será el sistema. La definición dada anteriormente tiene alguna relación con este enfoque. De hecho, su autor, Fowler, propone utilizar UML de esta manera, y sin adherir a la formalidad del metamodelo del lenguaje.(Véase [6], en la sección del capitulo 1 intitulada *¿Qué es UML Legal?*)
- *Como un lenguaje de programación*. De ahí el término "Executable UML", que es algo similar a MDA y donde también se llega a un sistema a través de la compilación de un modelo.

La definición oficial expresa: "es un lenguaje visual para especificar, construir, y documentar los artefactos de un sistema. Es un lenguaje de propósito general que puede ser usado con los principales métodos de componentes y objetos, y que puede ser aplicado a todos los dominios."**[1]**

3.1.3.3 CWM

Common Warehouse Metamodel" define un metamodelo que representa la metadata tanto técnica como de negocio que se encuentra habitualmente en el dominio de "*Data Warehousing*" y el *análisis de negocios*. Es usado como base para el intercambio de metadata entre sistemas de diferentes proveedores. Por ejemplo, comunica herramientas de "Data Warehousing" con herramientas de análisis.

3.1.3.4 XMI

"XML Metadata Interchange" es un estándar de la OMG que relaciona MOF con XML. Define como usar etiquetas XML para serializar modelos compatibles con MOF. Los metamodelos basados en MOF son transformados a DTDs, y los modelos son traducidos a documentos XML que respetan el DTD de su respectivo metamodelo, según Poole **[32]**.

En la especificación de transformación entre MOF 2.0 y XMI (v. 2.1.1), con fecha 07-02-2002, se habla de XML Schemas, no de DTDs. *XML Schema* es un lenguaje de esquema más potente que DTD. **[35]**

3.1.4 Resumen

La Arquitectura Dirigida por Modelos (MDA) es un framework para desarrollo de software definido por el "Object Management Group" (OMG).

MDA no modifica el ciclo de desarrollo de software, pero sí los entregables al finalizar cada etapa.

Se definen modelos, que se refinan gradualmente hasta llegar al código. El primer modelo es el "modelo independiente de plataforma". Mediante una transformación de modelo a modelo, se llega a uno o varios "modelos dependientes de plataforma". Estos, a través de una transformación de modelo a texto llegan a código ejecutable. Las transformaciones de MDA son realizadas siempre por herramientas.

4 LENGUAJES ESPECÍFICOS DE DOMINIO

El modelado específico de dominio DSM (Domain-Specific Modeling en inglés) 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) 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.

4.1 Características de los DSM

Kelly y Tolvanen [3] subrayan la importancia de que en el desarrollo dirigido por modelos tanto el lenguaje como el generador sean específicos al dominio.

Para levantar el nivel de abstracción los lenguajes necesitan estar al tanto del dominio. De esa manera la generación total de código puede ser real. Con lenguajes de modelado de propósito general (por ejemplo UML) esto es casi imposible, sino imposible. De ser posible, la última década habría mostrado cientos de casos exitosos.

Existen ejemplos de lenguajes específicos de dominio que muestran que se mejora la producción con este enfoque. [3]

Revisemos las características de los DSMs:

4.1.1 Concentrarse en un área de interés reducida

El modelado específico de dominio se propone automatizar el desarrollo de software en un área de interés reducido. Como su nombre lo indica, es específico a un dominio, no de propósito general. Cuanto más reducido es el centro de atención del lenguaje, más fácil resulta proveer soporte para la especificación de modelos y la automatización de la generación de código. Esto se debe a que el lenguaje y el generador de código conocen el dominio, y por lo tanto pueden completar la brecha semántica entre el modelo de entrada y el código de la aplicación.

Una solución particular DSM deja afuera todas las demás áreas de aplicación, no puede ser usada para desarrollar ningún otro tipo de características de las que los desarrolladores del DSM pretendieron. Por lo general compartir exactamente la misma solución DSM con otras compañías no es posible. Muchos de los detalles pueden ser aplicados, pero no necesariamente todos los detalles.

Usualmente, una solución de este tipo resuelve solo parte del dominio completo de la compañía. Por ejemplo, un banco completo es un dominio demasiado extenso, y el modelado específico de dominio requiere dominios más pequeños. La reducción del dominio podría enfocarse a sólo los productos de inversión del banco.

Los generadores de código también funcionan con un área de interés reducida. Sería imposible tener generadores de propósito general. El generador lee los modelos basado en el metamodelo del lenguaje y a partir de ellos genera el código.

4.1.2 Alto nivel de abstracción

El modelado específico de dominio levanta el nivel de abstracción al especificar la solución usando directamente los conceptos del dominio. Esto incrementa la productividad, no sólo en el tiempo y los recursos utilizados, sino también en el trabajo de mantenimiento.

Los conceptos del lenguaje se relacionan con el dominio del problema, y el generador traduce los modelos al dominio de la solución.

4.1.3 Generación completa del código

En DSM, la generación del código conviene que sea completa desde el punto de vista del desarrollador de la aplicación. La re-escritura manual del código no debería ser necesaria. Al igual que en los lenguajes de programación, el código generado puede ser enlazado con código preexistente, pero lo generado debería ser simplemente un producto intermedio, como los archivos *.o* en la compilación del lenguaje *C*.

Esto implica que, en lo posible, tanto el código estático (estructural) como el que implementa comportamiento debería ser generado.

La generación completa del código permite levantar el nivel de abstracción, y de esa manera aumentar la productividad. Claro que esto no siempre es posible. Un DSL puede ser ejecutable de varias maneras y en varios grados, aún si es no ejecutable. 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.

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.

4.1.4 Representaciones textuales, y otras

La mejor manera de representar una problema del dominio no es necesariamente el texto. El texto es usado habitualmente por los lenguajes de programación, pero es proclive a tener errores en la codificación y es difícil de manipular durante la generación.

DSM usa, además de texto, otras representaciones. Algunos ejemplos son: diagramas, matrices y tablas.

Estas escalan mejor que el texto puro, ofreciendo diferentes niveles de detalle. Se puede lograr con submodelos, ocultando información innecesaria, proveyendo diferentes vistas o vinculando especificaciones relacionadas. Es decir, en lugar de dar nueva información en el diseño, se puede forzar al desarrollador a elegir información ya especificada en otro submodelo, vista, etc.

4.1.5 Mayor cantidad de usuarios posibles

Estamos acostumbrados a pensar que los lenguajes de programación son usados por programadores. Pero los modelos creados con un DSM podrían, en algún caso, ser creados por personas que no trabajen como desarrolladores, y hasta podrían realizar una especificación completa del sistema.

4.2 Diferencia con otros enfoques de modelado

Existe una gran variedad de lenguajes de modelado. Sin embargo, la mayoría no se pensó para permitir la generación de código a partir del modelo. Esto es especialmente cierto para muchos lenguajes de modelado de propósito general.

4.2.1 ¿Como se diferencia DSM de UML?

El estándar UML ofrece muy poca ayuda en la automatización del desarrollo. UML no levanta el nivel de abstracción por encima del código. Sus conceptos centrales vienen del mundo del código: clases, métodos, atributos, etc..

El énfasis del lenguaje está puesto en "especificar, visualizar y documentar los artefactos", y no en la automatización del desarrollo con generadores de código.

Con UML no se puede forzar a los desarrolladores a seguir reglas de arquitectura, revisar la corrección del modelo con respecto a reglas del dominio, separar datos del modelo en diferentes vistas o aspectos relevantes en el dominio, ni saber como reusar información de los modelos.

En algunos casos, UML se usó para automatizar el desarrollo. Una inspección más profunda de esos casos nos muestra que UML no es seguido como en el estándar. El significado de los conceptos y estructura del lenguaje (metamodelo) fue alterado. Puesto de otro modo, se tomó el primer paso hacia el enfoque específico de dominio.

En este trabajo se estudia CSSL, que es, precisamente, lenguaje específico de dominio cuyo metamodelo está basado en UML. La diferencia es que CSSL toma algunos conceptos de UML, los extiende para adaptarlos al dominio, e ignora la gran parte de UML que no le interesa.

4.2.2 ¿Cómo se diferencia DSM de UML ejecutable?

De la misma manera, las iniciativas para usar UML como un lenguaje de programación, en una inspección más profunda, muestran que UML no es respetado completamente, sus conceptos son modificados y extendidos. Se crean lenguajes textuales adicionales, lenguajes de restricción como OCL y lenguajes de acción, o inclusive lenguajes de programación tradicionales. El objetivo de estos lenguajes es describir las transiciones de estados y otras acciones.

El UML ejecutable tampoco levanta el nivel de abstracción por encima del código. Sus conceptos centrales siguen perteneciendo al mundo del código: clases, métodos, atributos, etc.. Eso hace que algunos desarrolladores prefieran escribir las mismas estructuras que podrían escribir en UML directamente en código.

4.2.3 ¿Cómo se diferencia DSM de MDA?

MDA implica transformar un modelo (normalmente un modelo UML) en otro modelo UML, posiblemente en pasos sucesivos, hasta llegar a la generación de código. Cada uno de esos pasos los desarrolladores extienden el modelo generado con más detalles. En cambio, DSM pretende generar código directamente sin tener que modificar los modelos generados ni el código.

4.2.4 UML adaptado al dominio

Algunos partidarios de MDA visualizan mejoras en MDA incorporando elementos de DSM. En ellas, el UML base puede ser extendido usando "Profiles" para proveer información específica del dominio. "Profiles" es un mecanismo de extensión de UML, y será visto más adelante.

Algunas desventajas de esta aproximación son:

1. No se pueden agregar tipos totalmente diferentes al lenguaje
2. Están basados en conceptos de UML, y por lo tanto puede suceder que para acceder a los conceptos introducidos por los profiles se necesite pasar por elementos de UML, cuando eso no siempre es útil.

Sin embargo, para casos en los cuales la diferencia entre los conceptos de dominio y los conceptos básicos de UML no es tan grande, profiles es útil.

CSSL es un ejemplo de metamodelo que no se aleja demasiado de UML. La primer idea para el diseño del metamodelo fue el uso de un profile de UML. Desafortunadamente, con este mecanismo no se pueden agregar nuevos tipos al lenguaje, y además sólo se usaba un subconjunto muy pequeño del metamodelo de UML. De esa manera se llegó a pensar en la otra manera de describir un metamodelo, MOF.

4.2.5 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.

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.

4.2.6 Otras soluciones DSM

Los desarrolladores actuales ya disponen de soluciones DSM para algunas de sus actividades. Por ejemplo los *Diagramas de Entidad-Relación* (*DER*) permiten diseñar esquemas de bases de datos. Es un dominio relativamente pequeño. Lo mismo ocurre con el Diseño de interfaces gráficas y el diseño de protocolos. Las herramientas que generan esquemas e interfaces gráficas difieren de DSM en que ningún usuario de la compañía puede cambiar los generadores de código, ya que suelen ser soluciones propietarias.

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, etc. La figura enumera algunas soluciones:

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

Ilustración 4-9. Algunos de los DSLs más utilizados

4.3 Arquitectura de DSM

DSM propone una arquitectura de tres niveles para especificar como se debe realizar la automatización desde modelos a sistemas en funcionamiento.

- Un ***lenguaje específico de dominio*** provee un mecanismo de abstracción para abordar la complejidad de un determinado dominio. Los conceptos y reglas del lenguaje deben representar elementos del dominio, no del código.
- Un ***generador***, encargado de extraer la información de los modelos y transformarla en código.
- Un ***framework de dominio***, que provee la interfaz entre el código generado y la plataforma subyacente. En la mayor parte de los casos es necesario código extra, o componentes, que permitan que el código generado sea más simple.

El código generado es ejecutado junto con código adicional en un ambiente de ejecución (destino). Un producto puede usar parte de una plataforma más grande, como por ejemplo J2EE, puede utilizar completamente una plataforma, o puede,

incluso, usar varias.

4.3.1 División del trabajo de automatización

Las distintas asignaciones posibles del trabajo de automatización se ven en la figura (tomada de **[3]**):

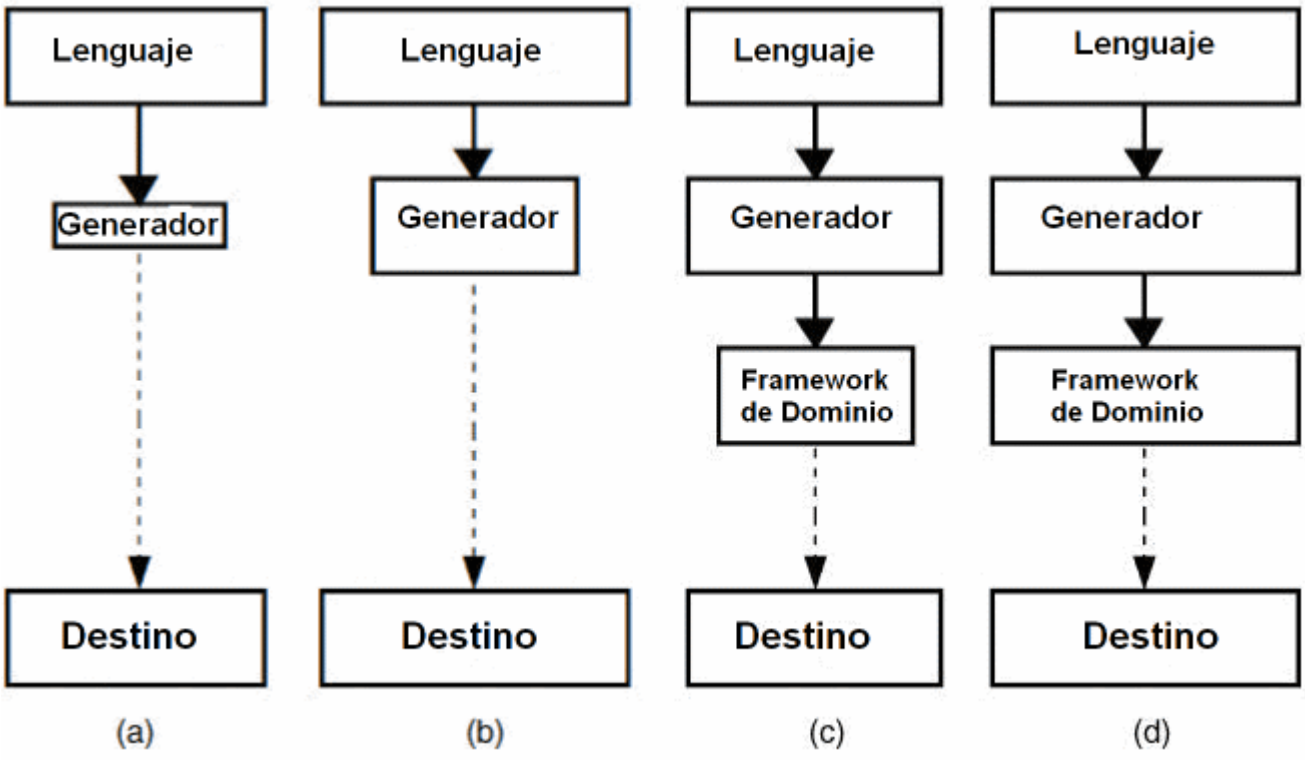


Ilustración 4-10 . Posibles asignaciones del trabajo de automatización

En los dos primeros casos, el framework de dominio puede ser muy reducido, o simplemente no estar. En el primer caso la mayor parte del trabajo de abstracción es realizado por el lenguaje de modelado y la generación es sencilla; y en el segundo el generador de código realiza más trabajo.

En los últimos dos casos, se escribe manualmente un framework de dominio, que será invocado por el código generado. También puede suceder que el código del framework sea emitido por el generador, en forma de trozos de código que son escritos cuando se los necesita. Por lo general, se intenta evitar que el generador tome la mayor carga de trabajo. Una razón para evitarlo es que el mantenimiento del generador se convertiría en el cuello de botella de la solución.

Dentro de esta arquitectura, cualquiera de los elementos puede cambiar si es necesario. Una forma de hacerlo es cambiar el generador para escribir código en otra plataforma, sin cambiar el lenguaje específico de dominio. También se pueden tener varios generadores, cada uno con un objetivo diferente. Puede existir uno que genere el código productivo, otro que genere prototipos y un tercero cuya salida permita la depuración del sistema.

4.4 Definición de un lenguaje

Para definir un nuevo lenguaje es necesario definir su sintaxis abstracta, su sintaxis concreta y su semántica.

4.4.1 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.

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.

4.4.2 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.

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.

4.4.3 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.

4.5 El proceso para el modelado de 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.

4.5.1 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.
- Frecuentemente ya tienen establecida una semántica.
- Establecen un mapeo natural con el problema que se quiere resolver con el lenguaje.
- Sólo deberían ser modelados los conceptos que permiten describir alguna variación, los conceptos comunes a todas las aplicaciones pueden implementarse como, por ejemplo, componentes.
- Los conceptos que pueden ser identificados a partir de una combinación de otros existentes deben ser excluidos.

4.5.2 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.

También existen otras fuentes típicas para encontrar conceptos, por ejemplo la descripción de la arquitectura, los productos existentes y sus manuales, especificaciónes disponibles, etc.

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.

4.6 Herramientas

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

4.7 Métodos para la definición de un lenguaje

Hace algunos años, los lenguajes se definían frecuentemente 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, y generalmente no lo están (ya que tienen una sintaxis gráfica, como UML) se necesita un mecanismo diferente para definirlos. Este mecanismo de definición se llama metamodelado.

Para definir un nuevo lenguaje existen varias alternativas:

- Si el lenguaje tiene puntos en común con UML, se puede ***extender el lenguaje UML*** a través de algunos de los mecanismos que este último provee.
- En algún otro caso, se puede **utilizar un *meta-meta lenguaje*** (un lenguaje de nivel M0 según las cuatro capas de MDA). Veremos más adelante un ejemplo utilizando el metalenguaje EMF, una implementación en Java de MOF.

4.7.1 Extensión de UML

Anteriormente se mencionó que algunos partidarios de UML favorecen la idea de utilizar UML en combinación con DSM. En el trabajo de Bruck y Hussey **[5]** se describen las diferentes opciones disponibles para extender UML e introducir conceptos específicos de un dominio.

4.7.1.1 Extensión "Featherweight" (muy liviana)

Este tipo de extensión involucra el uso de palabras clave ("Keywords"). Los keywords son palabras reservadas que normalmente aparecen como anotaciones de texto unidas a un elemento de UML. La especificación de Infraestructura de UML [2] describe el uso de keywords y mantiene una lista de keywords predefinidas, en el Anexo B.

Ejemplo:

El metatipo "uml::Interface" tiene una apariencia similar a "uml::Class". Por lo tanto se usa el keyword "<<*interface*>>" para distinguir las interfaces de otros clasificadores.

Los usos de los keywords son:

- Distinguir un concepto UML de otro que comparte su misma representación gráfica
- Distinguir un tipo de relación de otras relaciones que comparten la misma representación gráfica
- Especificar el valor de algún modificador unido a un concepto UML.
- Indicar un estereotipo estándar

4.7.1.2 Extensión Liviana

La extensión liviana ("lightweigt") involucra el uso de Profiles ("perfiles"). Se detalla en la especificación de superestructura de UML [2], capítulo 18.

Un Profile debe ser la primera opción al decidir como extender UML. Define extensiones limitadas al metamodelo de UML.

Este no es un mecanismo de extensión de primera clase (es decir, no permite modificar el metamodelo existente). En cambio, la intención de los profiles es proveer una manera rápida y directa para adaptar un metamodelo con construcciones que son específicas a un dominio en particular[2].

La extensibilidad de primera clase se logra a través de MOF, donde no hay restricciones en qué se puede hacer con un metamodelo: puede agregar o eliminar metaclases y relaciones si es necesario[2].

El mecanismo de extensión primario es el Estereotipo. Los estereotipos pueden ser usados para agregar: keywords, restricciones, imágenes, y propiedades (valores etiquetados) a los elementos del modelo.

4.7.1.3 Extensión de peso medio ("middleweight")

Este mecanismo de extensión extiende UML a través de la especialización de metatipos de UML. Puede ser considerado un mecanismo de extensión de primera clase y como tal expone conceptos tales como "sub-setting" y redefinición.

Crear una extensión de este tipo implica:

1. Extender referenciando *"UML.metamodel.uml"* (el metamodelo completo de UML, como se ve en la figura de esta sección)
2. A esos metatipos agregar los tipos específicos del dominio.

En la figura se ve una extensión *middleweight* a UML2 implementada sobre el plugin UML2 de eclipse. En la parte de abajo se pueden ver los archivos del metamodelo de UML, en la carpeta "*pathmap://UML_METAMODELS*". Arriba están los conceptos correspondientes al dominio de aplicaciones colaborativas. Existe un tutorial para realizar los tres tipos de extensiones (liviana, semi-pesada, pesada), y este ejemplo (claramente, si presta atención al nombre org.eclipse.uml2.tutorial.mw) está basado en uno de los ejemplos del tutorial.

Ilustración 4-11 . Ejemplo de extensión middleweight (CSSL)

Si se presta atención al ejemplo se pueden ver los problemas de este tipo de extensión.

- Se extiende **todo** el metamodelo de UML (en el archivo "*UML.metamodel.uml*")
- Se crea una dependencia a una versión de UML

- Las clases de implementación que se generan automáticamente a partir del archivo *"GroupwareProfile.metamodel.uml"* referencian clases internas de UML, y eso resulta en advertencias de compilación.

4.7.1.4 Extensión Pesada

Las extensiones "Heavyweight" (en castellano: "peso pesado") implican el reuso a través de la copia y la mezcla ("merge") de paquetes, en lugar del reuso mediante la especialización de tipos de un metamodelo como con las extensiones "middleweight".

Crear una extensión de este tipo implica:

- Seleccionar las unidades del lenguaje que se quiere extender y realizar un "merge" de dichas unidades.
- A esa conjunto de metatipos agregar los metatipos del dominio

4.7.1.5 El metamodelo de ejemplo

El gráfico de la sección de extensiones "middleweight" muestra una implementación en el plugin uml2 de eclipse de un DSL para sistemas colaborativos.

Además de los problemas ya descriptos, usar mecanismos de extensión de UML para un DSL implica entender las metaclases de UML, y además algunos conceptos relativos a las extensiones de primera clase.

Se usan conceptos como "Subconjuntos", "Subconjuntos derivados", "Subconjuntos no derivados", "Redefinición" y "Uniones derivadas". Se explican a continuación:

- **Subconjuntos**: representa al concepto de subconjunto tal cual en la teoría de conjuntos. Existen dos variantes:
    - *Subconjuntos derivados*: los usuarios no pueden agregar directamente a estas colecciones porque estas son calculadas a partir de otros elementos del metamodelo.
    - *Subconjuntos no derivados*: contienen valores que no pueden ser calculados directamente a partir de otros elementos del metamodelo.
- **Redefinición**: es una relación entre características de clases en una jerarquía de clases. Puede ser usado para cambiar la definición de una característica de clase
- **Unión derivada**: la colección de valores de la propiedad es derivada como la unión estricta de las propiedades definidas como subconjuntos de esa propiedad. Se puede trazar una analogía con los métodos abstractos en Java, porque los tipos concretos hacen útil a la unión derivada, contribuyendo subconjuntos. Esta propiedad es de sólo lectura y derivada.

En el ejemplo, en la metaclase de relación "ParticipationInSessionRelationship" la propiedad "session" podría ser un conjunto derivado de los miembros de la asociación (Association::memberEnd). Podrían filtrarse solo los elementos que cumplen con que su metaclase es "Session".

Otra manera de implementarlo sería declarar una propiedad normal, pero que el valor que se seleccione en esa propiedad también va a pertenecer a "Association::memberEnd". Dicho de otra manera, le aportaría sus elementos a la colección "Association::memberEnd".

En ese caso, "session" podría ser un subconjunto (no derivado) de "Association::memberEnd". En ese caso, al recorrer los elementos de "Namespace::member" también recorreríamos las sesiones, pues "Namespace::member" es una *unión derivada* que contiene a "Association::memberEnd".

En conclusión, la complejidad de los conceptos relacionados a este tipo de extensiones se ve claramente a partir de las explicaciones anteriores. Esto nos puede llevar a pensar que, para el caso de CSSL, la mejor implementación es utilizar MOF. CSSL usa un subconjunto muy pequeño de UML, y estudiar el metamodelo de UML a fondo no parece una buena idea.

4.7.2 MOF

El lenguaje MOF, acrónimo de "*Meta-Object Facility*", es un estándar de la OMG para la ingeniería conducida por modelos. Como ya se vio, MOF se encuentra en la capa superior de la arquitectura de 4 capas. Provee un meta-meta lenguaje en la capa superior que permite definir metamodelos en la capa M2. El ejemplo más popular de un lenguaje en la capa M2 es el metamodelo de UML, que describe el lenguaje UML.

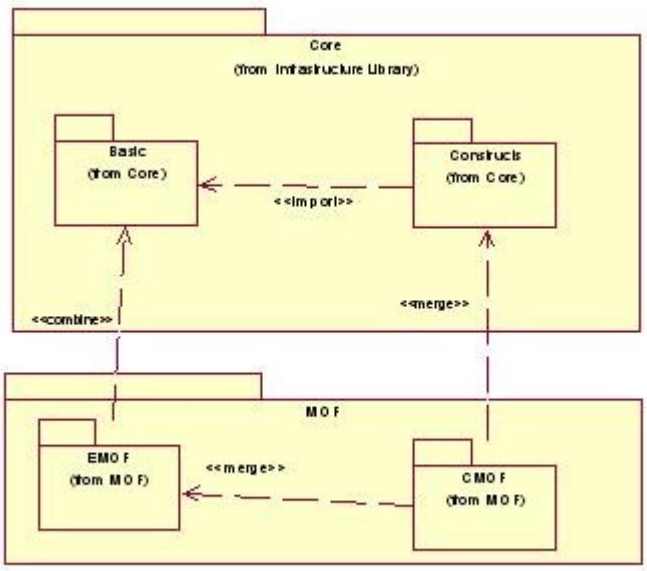


Ilustración 4-12 . Relación entre EMOF y CMOF

MOF es una arquitectura de metamodelado cerrada. Esto significa que el metamodelo MOF se define en términos de si mismo. MOF permite una arquitectura de metamodelado estricta, es decir, cada elemento de un modelo en cualquiera de

las capas tiene una correspondencia estricta con un elemento del modelo de la capa superior.

Actualmente, la definición de MOF esta separada en dos partes fundamentales, EMOF (Essential MOF) y CMOF (Complete MOF), y se espera que en el futuro se agregue SMOF (Semantic MOF). En la figura puede verse la relación entre EMOF y CMOF. Ambos paquetes importan los elementos de un paquete en común, del cual utilizan los constructores básicos y lo extienden con los elementos necesarios para definir metamodelos simples, en el caso de EMOF y metamodelos más sofisticados, en el caso de CMOF.

4.7.2.1 Implementación de MOF - Ecore

El metamodelo MOF esta implementado mediante un plugin para eclipse llamado Ecore. Este plugin respeta las metaclases definidas por MOF. Todas las metaclases 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.

4.7.2.2 Relación entre el meta-metamodelo MOF y el meta-metamodelo Ecore

La primera implementación de Ecore fue terminada en Junio del 2002. Esta primera versión usaba como meta-metamodelo la definición estándar de MOF (v1.4).

Gracias a las sucesivas implementaciones de Ecore y basado en la experiencia ganada con el uso de esta implementación en varias herramientas, el metalenguaje evolucionó. Como conclusión, se logró una implementación de Ecore realizada en Java eficiente, con sólo un subconjunto de MOF, y no con todos los elementos como manipulaban las implementaciones hechas hasta ese momento.

A partir de este conocimiento, el grupo de desarrollo de Ecore colaboró más activamente con la nueva definición de MOF, y se estableció un subconjunto de elementos como esenciales, llegando a la definición de EMOF, que fue incluida como parte de MOF (v2.0).

4.7.2.3 Diferencias entre MOF y Ecore

MOF y Ecore son conceptualmente muy similares, ambos basados en el concepto de clases con atributos tipados y operaciones con parámetros y excepciones. Además ambos soportan herencia múltiple y utilizan paquetes como mecanismo de agrupamiento.

La principal diferencia 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. 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. Dos *EReferences* pueden definirse como opuestas para establecer una relación navegable hacia ambos sentidos. Existen ventajas y desventajas para esta implementación. Como ventaja puede verse que las relaciones simétricas, como por ejemplo "esposoDe", implementadas con *Association*, son difíciles de mantener ya que debe hacerse consistentemente. En cambio con *Ecore*, al ser sólo una referencia, ella misma es su opuesto, es decir, se captura mejor la semántica de las

asociaciones simétricas, y no es necesario mantener la consistencia en el otro sentido.

Ambos, MOF y *Ecore*, soportan el reuso de conceptos en otros paquetes. MOF soporta varios mecanismos de reutilización de elementos como “import”, herencia de paquetes, no disponibles en *Ecore*. Sin embargo, *Ecore* permite tanto la definición recursiva entre paquetes, como que sean leídos conjuntamente, mientras que MOF prohíbe las dependencias cíclicas entre paquetes debido a los problemas prácticos para mantener la consistencia mutua en un ambiente distribuido.

MOF es mucho más rico y expresivo. Por ejemplo, define un concepto *Constant*, mientras que en *Ecore* se define como un *EAttribute* que no puede cambiar (*unchangeable*) y sólo puede ser definido dentro de un elemento *EClass*. Respecto a los elementos *Constraint* de MOF, *Ecore* no posee un concepto equivalente.

4.8 Resumen

El objetivo de este trabajo es la generación de código a partir de modelos especificados en el lenguaje CSSL. Por eso resulta conveniente entender los mecanismos para la definición de un lenguaje.

En este capítulo se estudiaron las características fundamentales de DSM ("Domain Specific Modelling", modelado específico al dominio). Es importante que un DSL aumente el nivel de abstracción. Para ello es necesario centrar la atención en un dominio reducido.

Tomando ese dominio reducido, se debe proceder a la definición del lenguaje. La definición de un lenguaje comprende tres partes esenciales: la sintaxis abstracta, la sintaxis concreta y la semántica.

Para definir los términos que formarán parte de la sintaxis abstracta se requieren varios pasos. Se estudió como encontrar los términos relevantes en el dominio. Y luego se vieron las opciones para definir el lenguaje, tanto a través de la extensión de UML y como del metamodelado usando EMF.

5 SISTEMAS COLABORATIVOS

El campo de investigación llamado "Trabajo Cooperativo asistido por Computadora" (CSCW, "Computer-Supported Cooperative Work") está relacionado al entendimiento de las interacciones sociales y el diseño, desarrollo y evaluación de sistemas técnicos que asisten la interacción social en equipos y comunidades. En esta definición y en otras cabe destacar que se mencionan dos aspectos: el aspecto tecnológico y el fenómeno social. [25]

Los sistemas resultantes de la investigación y desarrollo en esta materia es llamado "groupware" o sistemas colaborativos. Son sistemas basados en computadoras que ayudan a un grupo de personas comprometidas en una tarea u objetivo en común y que proveen una interfaz a un ambiente compartido. [25] [26]

Según Koch y Gross ([25]) se distinguen cinco tipos de aplicaciones: soporte para awareness (conciencia de grupo), comunicación, coordinación, equipo y comunidad.

5.1Conceptos Principales

Los siguientes términos forman un vocabulario conocido por los miembros de la comunidad que trabaja con tecnología groupware

Rol (CollaborationRole): Entendemos al rol como un conjunto de propiedades, conocimientos y responsabilidades que tendrá un usuario en un determinado momento. Permitirá entender cual será el papel que el usuario tendrá en todo momento cuando interactúe con otros usuarios en el sistema.

Usuario (User): Representa a un usuario dentro del sistema.

Objetos colaborativos (SharedObjects): Son los elementos creados por los propios usuarios que se pueden manipular en forma colaborativa gracias a las herramientas colaborativas que lo soportan. 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 (al mismo tiempo) o asincrónica.

Workspace (Workspace): Es el lugar en el que la *colaboración* se lleva a cabo y define, en parte, el estilo de colaboración que se va a llevar a cabo. En este trabajo se rescata a los workspaces como los elementos que contextualizan la colaboración. Usualmente un workspace es definido dentro de un entorno más grande que lo contiene que llamaremos escenario colaborativo

Sesión (Session): Una sesión es un período de interacción soportada por un sistema. En general un usuario ingresa (login) en la sesión identificándose a través de un nombre de usuario y password y sale de la misma (logout) en forma explícita o simplemente cerrando la aplicación (o ventana) que maneja la sesión.

Hacemos notar que desde el punto de vista de la implementación, la sesión se podría utilizar como un objeto para intercambiar información entre los usuarios conectados al sistema. Por ejemplo datos del usuario, nombre, rol, estado, etc. que se comunican para saber quien está conectado en ese momento en el sistema. En definitiva, la forma en que los sistemas groupware manejan la sesión es una de las principales diferencias con los sistemas monousuarios.

Herramientas colaborativas (Tool): 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.

Las herramientas pueden ser independientes del protocolo involucrado. Es decir que una misma herramienta puede ser utilizada con distintos protocolos.

Escenario Colaborativo (Setting): Es la integración de un conjunto de workspaces. A menudo, las aplicaciones groupware complejas necesitan más de un workspace, de una manera similar a una universidad virtual que estará compuesta por distintos workspaces como el aula, la biblioteca, etc. Los escenarios contienen asimismo los protocolos que estructuran el acceso y el uso de los diferentes workspaces por parte de los distintos roles existentes.

Asociación (CollaborativeAssociation): Permite asociar elementos del diseño. Con esta asociación vinculamos las herramientas que disponemos en un workspace, los roles que pueden participar de una sesión etc.

Contexto compartido (SharedRepository): Es el repositorio donde se alojan los objetos colaborativos. Los usuarios tendrán acceso a este repositorio para crear, leer y modificar los objetos colaborativos.

El contexto compartido puede ser único (es decir que todos los usuarios de la aplicación comparten a todos los objetos colaborativos en un lugar). En otros casos, puede haber varios contextos compartidos, asociados a alguna componente del sistema. Por ejemplo cada usuario podrá tener un espacio en el que coloca los documentos que quiere compartir. Asimismo, puede determinar quién puede acceder a su contexto compartido y qué acciones puede realizar dentro del mismo. En otros casos, los contextos compartidos pueden estar asociados a determinados espacios. Por ejemplo, cada aula podría tener un contexto compartido donde los usuarios comparten sus trabajos. Finalmente los contextos compartidos podrían generarse dinámicamente cuando un usuario decide compartir algún documento con otro usuario.

Telepuntero (Telepointer): Es el cursor del mouse de cada uno de los usuarios que está conectado a un objeto colaborativo y que puede ser movido por cada usuario. En algunos casos los cursores de los usuarios pueden ser visualizados todos al mismo tiempo.

Protocolo (Protocol): Un factor importante en el trabajo en grupo es el *proceso* social que se lleva a cabo. Sin gente interactuando, el sistema groupware está muerto. Los protocolos sirven para modelar, guiar y estructurar el proceso social que se lleva a cabo dentro del grupo. Los protocolos definen qué herramientas y objetos colaborativos pueden ser utilizados por los distintos roles o usuarios. Un aspecto importante que define un protocolo es en qué momento puede participar cada usuario.

Vista (View): Es una porción del contexto compartido que puede ser visualizada por un usuario. En algunos casos, los usuarios pueden visualizar todo el contexto compartido, y en ese caso la vista es una representación del contexto compartido. En otros casos, por limitaciones de los dispositivos, la vista representa una porción de la información.

Meta protocolo (Meta-Protocol): En algunas ocasiones es necesario contemplar una forma de cambiar de protocolo. En síntesis el Meta protocolo es el

protocolo que administra los cambios de protocolo. Los meta protocolos controlan el cambio y la transición entre los protocolos y proveen una forma de hacer que los workspaces sean más flexibles.

Acoplamiento: El acoplamiento es una medida que determina el grado de ensamble que tiene el mismo componente en los puestos de trabajo de un sistema groupware. Las componentes pueden estar acopladas o no y el acoplamiento lo mencionaremos como un acoplamiento fuerte o débil. Si decimos que el sistema tiene la vista fuertemente acoplada, significa que los usuarios tienen la misma vista. Si tienen el workspace débilmente acoplado (también podemos usar el término desacoplado), estamos diciendo que los usuarios pueden estar en distintos workspaces. El mismo concepto puede aplicarse a otros conceptos como el protocolo, la herramienta, el telepointer, etc. Esta medida determina en cierta forma la complejidad del sistema y de la colaboración. En general el sistema va a ser más simple (mas simple para desarrollarlo y también mas simple para operarlo) cuando sus componentes estén fuertemente acopladas.

Awareness: Es la información que el sistema provee sobre el estado de la colaboración. En las reuniones presenciales estar al tanto de los otros ("staying aware of others") es algo natural. Se puede percibir dónde está ubicado cada uno, cuál es su estado, qué actividad está desarrollando y con qué objeto. Por el contrario, mantener actualizada esta información en sistemas groupware es bastante difícil. El awareness del workspace involucra mantener constantemente actualizada la información de los otros usuarios en relación al espacio compartido indicando al menos la identidad y la presencia de los usuarios. Conjuntamente con esta información suele aparecer otra información de awareness como la actividad que están desarrollando, su ubicación dentro del sistema, su estado, qué acciones va a desarrollar, qué cambios está realizando, objetos que se utilizan, etc.

Avatar: Es la representación de un usuario dentro del sistema. Puede ser una pequeña imagen, un gráfico o un ícono. En algunos casos, como en los ambientes virtuales, el avatar se mueve dentro del ambiente y sirve para iniciar colaboraciones con ese usuario. El avatar puede servir como información básica de awareness y en muchos sistemas colaborativos indican la presencia y la ubicación de los usuarios dentro de workspaces.

5.2 Espectro de sistemas colaborativos

Los sistemas que soportan tareas comunes y ambientes compartidos varían ampliamente; es, por tanto, apropiado pensarlos más en términos de un espectro con múltiples dimensiones.

5.2.1 1.1.1. Según su nivel de integración:

Se describen aquí dos sub-dimensiones:

Subdimensión de tarea común: En un nivel de integración *bajo* los usuarios realizan tareas en forma independiente. Cada usuario utiliza algún recurso del sistema pero el sistema no fomenta la integración de los usuarios.En un nivel de integración *alto* los usuarios realizan sus tareas en forma conjunta, dependen unos de otros y hay una constante percepción de las actividades que realizan los demás colaboradores.

Subdimensión de Ambiente Compartido: En un nivel de integración bajo los usuarios colaboran en ambientes independientes. En un nivel de integración alto los usuarios colaboran en un ambiente compartido donde comparten información y se comunican

5.2.2 Según el tiempo y el espacio

Los sistemas groupware pueden ser concebidos tanto para ayudar a grupos en los cuales las personas se encuentren en un mismo lugar, como para grupos en los cuales sus integrantes estén distribuidos en distintas ubicaciones. A su vez, un sistema de groupware puede ser concebido para reforzar la comunicación y la colaboración dentro de una interacción en tiempo real (sincrónica), o de una interacción que no se lleva a cabo en tiempo real (asincrónica).

Estas consideraciones de tiempo y espacio sugieren cuatro categorías de sistemas de groupware representados en el siguiente cuadro de doble entrada:

	Mismo Tiempo	**Diferente Tiempo**
Mismo Lugar	*Interacción cara a cara*	*Interacción asincrónica*
Diferente Lugar	*Interacción sincrónica distribuida*	*Interacción asincrónica distribuida*

Tabla 5-1. Clasificación según tiempo y espacio

5.2.3 Restrictivo contra Permisivo

Otra dimensión del espectro que estamos analizando es el grado de libertad que tienen los usuarios para desarrollar sus tareas. Algunos sistemas colaborativos como los sistemas de *workflow*, restringen o dirigen el comportamiento de los usuarios. Este tipo de sistemas son considerados restrictivos y típicamente mantienen un curso de acción posible o deseable que restringe las posibilidades de acción de los usuarios.

Otros sistemas, tales como los sistemas de pizarrón compartido o sistemas de escritura colaborativa permiten que los usuarios puedan libremente realizar sus actividades (escribir, navegar, dibujar, entrar o salir, etc.) sin control del sistema.

Los sistemas permisivos son más complejos de desarrollar. Por un lado, requieren soportar las alternativas de acción de los usuarios brindando la funcionalidad que corresponda. Y por otro, requieren elementos de awareness que informen qué actividades los usuarios están realizando.

5.3 Desarrollo de Groupware

Según Borges y Pinheiro [27], la interacción armoniosa de un grupo depende del entendimiento mutuo, y dicho entendimiento requiere un soporte para:

1. Comunicación entre los participantes
2. Coordinación de sus actividades
3. Una memoria de grupo
4. Conciencia de grupo (awareness)

Existen varios frameworks para el desarrollo de groupware. Algunos de ellos intentan abarcar el soporte de las cuatro características mencionadas, y otros se concentran en una.

Por ejemplo, un trabajo de Dias y Borges[28] presenta un framework para el soporte de awareness en sistemas colaborativos. Dicho framework sólo abarca este tema. Lo hace a través de eventos. El código que lo utiliza debe registrar los eventos que pueden suceder, luego notificar cada una de las ocurrencias de cada evento, y por último el framework pondrá en aviso a los demás usuarios sobre los eventos que correspondan según los filtros aplicados.

Por otro lado, Agora, DyCE y Coast [19][20][18] son frameworks que intentan facilitar la implementación de todas las características de un sistema groupware, o al menos una gran parte de dichas características.

Greenberg y Roseman, en su trabajo "Groupware Toolkits for Synchronous Work" [34], el desarrollo de software para este tipo de sistema requiere poner atención en:

- El manejo de procesos distribuidos
- Comunicación Inter-Proceso
- Gestión de estado y sincrónización de procesos
- diseño de widgets para groupware
- creación de administradores de sesión
- control de concurrencia
- seguridad.. etc

A partir de esa información sostienen que los frameworks (o "Toolkits") deben proveer las siguientes características:

- **Arquitecturas de tiempo de ejecución**, que manejen automáticamente los procesos, sus interconexiones y las comunicaciones
- **Abstracciones de Programación para Groupware**, utilizables por los programadores para sincronizar los eventos de interacción y los modelos de datos entre procesos, y también las vistas presentadas en cada pantalla.
- **Widgets para Groupware**. Los widgets específicos pueden ser extensiones de los widgets usuales para ser utilizados en aplicaciones groupware o nuevos widgets diseñados específicamente para estos sistemas (por ejemplo la lista de contactos).
- **Administradores de sesión**, que permitan a los usuarios crear sesiones, unirse a ellas, o dejarlas.

Las necesidades del programador deberían estar cubiertas con las características propuestas. En las siguientes secciones se explicarán cada uno de los desafíos en el desarrollo de groupware.

5.3.1 Arquitecturas de tiempo de ejecución

Los sistemas groupware sincrónicos están compuestos, por lo general, por múltiples procesos comunicados por una red. Los toolkits no deben ser sólo librerías de programación, sino que deben proveer una arquitectura de tiempo de ejecución, de la misma manera que en las aplicaciones empresariales existe middleware como los "servidores de aplicación".

Estas arquitecturas deben ayudar al programador en la tarea de implementar la comunicación, el control de concurrencia, la sincronización y la tolerancia a fallos.

Pero estos últimos cuatro temas son influidos profundamente por la distribución del sistema. Por esa razón, se tratarán primero las posibilidades de distribución del procesamiento y datos de un sistema, para luego ver las variantes que deben simplificar los frameworks en cuanto al control de concurrencia. Luego se revisarán los otros temas concernientes a la arquitectura de tiempo de ejecución, es decir, comunicación, tolerancia a fallos y sincronización.

5.3.1.1 Arquitecturas centralizadas vs Distribuidas

Las arquitecturas centralizadas usan una única aplicación, que reside en el servidor central y controla toda la Entrada/Salida de los participantes distribuidos. Los programas cliente sólo envían peticiones al servidor, y muestran los resultados devueltos por este. La principal ventaja es que no se necesita sincronización, la consistencia de los datos compartidos es sencilla porque existe una única copia, y esos datos son manipulados por un único programa que puede, si es necesario, serializar el acceso concurrente para que no ocurran inconsistencias.

Las arquitecturas replicadas ejecutan una copia del programa en cada sitio. El programa debe sincronizar cada acción (local y remota) con sus pares. Y debe garantizar que todas las copias de los datos sean consistentes.

Las arquitecturas centralizadas son simples para manejar la concurrencia. Pero pueden traer problemas de latencia, cuellos de botella y ambientes heterogéneos. La latencia es el tiempo aparentemente inactivo, mientras el pedido se envía al servidor, este lo procesa y envía las respuesta o actualiza las pantallas. Si ese tiempo es alto será un problema, pues hará lenta la interacción del usuario. El cuello de botella puede ubicarse en el procesador central o la red. El tercer problema mencionado, los ambientes heterogéneos, significa que a menudo es complicado que un proceso pueda actualizar correctamente diferentes clientes remotos, dado que cada uno de ellos podría tener un "Look & Feel" diferente.

Las arquitecturas replicadas solucionan estos problemas manejando las interacciones y la actualización de pantalla en cada replicación. Las acciones de cada usuario se pueden realizar primero en la copia local y luego distribuirse, resultando en una latencia muy baja. La performance mejora al intercambiar sólo la información mínimamente necesaria para mantener el estado local de la réplica consistente con el de los demás. Y cada réplica es responsable de dibujar sólo su interfaz gráfica local, lo que permite que en cada sitio la interfaz pueda ser distinta, por ejemplo usando el "Look & Feel" nativo.

Pero todo esto se da al costo de una mayor complejidad. Se deben tratar asuntos tales como el control de concurrencia. Y cada framework tiene sus variantes para ayudar al programador. Por ejemplo, Share-Kit no tiene control de concurrencia, y si es necesario, lo debe implementar el programador desde el principio. DistEdit usa broadcasts atómicos. ObjectWorlds usa objetos compartidos, y tiene la habilidad de detectar mensajes que no llegaron en orden para hacer un bloqueo no optimista. Groupkit puede forzar la serialización de ciertas acciones.

Existen arquitecturas intermedias (semi-replicadas) que contienen tanto componentes replicados como centralizados. Por ejemplo, se puede disponer de un servidor central utilizado únicamente para mantener el estado y distribuir los cambios de estado emitidos por un sitio a todos los demás. Se arma así un modelo

de distribución *hub-and-spoke*. Queda a cargo de las replicas decidir cómo será la interfaz gráfica, y actualizar la pantalla de acuerdo a las notificaciones recibidas.

5.3.1.2 Control de Concurrencia

Los frameworks o toolkits deben proveer una variedad de esquemas de control de concurrencia, para que luego cada programador pueda decidir explícitamente cuál usar.

El control de concurrencia es necesario dentro de sistemas colaborativos para ayudar a resolver conflictos entre participantes, y para permitir que participen en actividades altamente acopladas. Esto sucede, por ejemplo, cuando en un editor colaborativo un usuario elimina una oración que otro está editando. [21]

El control de concurrencia debe asegurar que el estado de un documento sea consistente en una arquitectura replicada. Existen dos aspectos de concurrencia a tener en cuenta[33]:

- consistencia del ***estado de varias copias*** distribuidas del documento, asumiendo una arquitectura distribuida
- consistencia del ***estado resultante*** *con el* ***resultado esperado*** *por el usuario*. Si el control de concurrencia garantiza que las copias de los datos son iguales en todos los sitios, pero el estado en el que queda el documento no es el esperado por los usuarios cuando hay un conflicto, estaríamos ante este tipo de inconsistencia.

Antes de proseguir, se analizarán algunos asuntos y/o términos relacionados al control de concurrencia:

- ***WYSIWIS:*** Las pantallas WYSIWIS ("what you see is what I see") tienen dos implicaciones en el control de concurrencia. En primer lugar el tiempo necesario para acceder a los datos, modificar datos y notificar cambios a otros usuarios debe ser lo más corto posible. Segundo, si el uso de un esquema de concurrencia lleva a que los cambios hechos por un usuario no sean vistos inmediatamente por otros usuarios, se deben considerar las consecuencias de dicho esquema en la dinámica del grupo.
- ***Distribución en área amplia:*** se debe considerar que la transmisión de datos en redes WAN suele ser más lenta que en las redes LAN. Se debe tomar en cuenta el tiempo de respuesta.
- ***Replicación:*** como el tiempo de respuesta es tan importante en estas aplicaciones, se suelen replicar localmente el estado de ciertos datos. La replicación de datos será explorada más profundamente en otra sección.
- ***Robustez:*** normalmente se refiere a la recuperación en circunstancias no usuales, por ejemplo fallas. Pero los sistemas colaborativos también deben recuperarse de acciones no esperadas de los usuarios (dejar la sesión abruptamente, la entrada de un nuevo usuario a la sesión, inactividad del usuario).

5.3.1.3 Esquemas de Control de Concurrencia

Las siguientes estrategias de control de concurrencia son utilizadas para permitir una modificación consistente de los datos compartidos. Se las puede clasificar, en general, en pesimistas y optimistas. Las técnicas **pesimistas** aseguran que no habrá inconsistencias entre copias adquiriendo bloqueos para evitar modificaciones

conflictivas sobre los mismos datos. Las técnicas **optimistas** no impiden que ocurran las inconsistencias, pero las detectan y corrigen si es necesario.

5.3.1.3.1 *Bloqueo*

Se trata simplemente de bloquear los objetos que serán modificados. Es lo que en bases de datos se llama bloqueo pesimista.

Conlleva tres problemas principales.

- El overhead del pedido del bloqueo. Eso incluye una potencial espera y degradación del tiempo de respuesta.
- Se debe analizar la granularidad del bloqueo. Es decir, qué parte del objeto es necesario bloquear.
- En qué momento deben ser pedidos y liberados los bloqueos.

A estas consideraciones se puede agregar algún problema de robustez si es que un usuario bloquea elementos que nunca libera, y el sistema no tiene medios para solucionarlo.

Una variante es el esquema de bloqueo **basado en token**. Cuando un sitio necesita editar una parte del documento, obtiene un token que representa el bloqueo de esa parte del documento. Cuando libera el bloqueo, sólo lo marca como disponible para otros usuarios. Si el mismo usuario desea obtener el mismo bloqueo, sólo debe volver a marcar el token como no disponible, sin necesidad de comunicarse con los otros nodos de la red. Esto es útil en aplicaciones donde no hay varios usuarios interactuando sobre los mismos objetos en exactamente el mismo momento. Es algo normal en editores donde un usuario realiza toda la actividad y los otros sólo observan.

Una complicación con este enfoque es la tolerancia a fallos. Se debe implementar un algoritmo de recuperación de un token, para lidiar con situaciones en las que un sitio cae cuando todavía tenía un token.

Otra técnica para reducir el impacto en el tiempo de respuesta es soportar muchos bloqueos de granularidad fina en el documento. Muchos usuarios esperarán en muchos bloqueos distintos, por lo tanto se reduce la probabilidad de una espera.

5.3.1.3.2 *Transacciones*

Una transacción es generalmente definida como una unidad de trabajo que se hace a nombre de una aplicación o componente. Cada transacción puede estar compuesta de múltiples operaciones realizadas en datos que están dispersos en uno o varios procesos o en una o varias máquinas.

Tiene algunos problemas en sistemas groupware. En primer lugar, las transacciones distribuidas consumen muchos recursos. Además, si la transaccionalidad está implementada en base a bloqueos, surgen los problemas inherentes a la estrategia de bloqueos. Si se implementa con timestamps (también conocido como bloqueo optimista), los cambios pueden ser revertidos debido a la interferencia con el trabajo de otro usuario. Eso, que en una Base de Datos es poco usual y de un impacto bajo, es problemático en un sistema colaborativo, donde no suele ser apropiado que dos personas vean un estado distinto gracias al aislamiento de las transacciones que impide ver los estados intermedios.

La gran diferencia filosófica entre las Bases de Datos y los sistemas colaborativos es que los últimos se esfuerzan por mostrarle a los otros usuarios las acciones del usuario que está interactuando, mientras que las transacciones de Bases de Datos intentan dar la sensación de ser la única transacción en el sistema. La literatura del tema habla de niveles de aislamiento de transacciones, y los motores de Bases de Datos, como mínimo, proveen el nivel de "Read-Commited", en otras palabras nunca se leen datos que no hayan sido comprometidos ("commited") por otra transacción.

En el caso de los editores colaborativos, es posible que sea necesario considerar un conjunto de operaciones como una única operación[33]. Por ejemplo, la operación "buscar y reemplazar", ejecutada por un usuario y seleccionando por cada coincidencia si debe reemplazarse o no. Al ejecutar "Deshacer", debería deshacerse toda la operación.

El problema es que las operaciones parciales deben ser ejecutadas para proveer feedback de las acciones. Una solución podría ser ejecutar la operación localmente, y cuando la operación se sabe confirmada, enviar las operaciones a las réplicas. Los otros usuarios sólo verían la operación terminada.

5.3.1.3.3 Unico Participante Activo

Sólo un participante a la vez posee el "Floor Control" (control del piso). El acceso al "Floor Control" puede administrarse a través de un protocolo implementado en el software o mediante un protocolo externo (por ej: acuerdo verbal entre usuarios).

El problema con este esquema es que no es apropiado para sesiones con un nivel alto de paralelismo entre los participantes. Además, si el cambio en el control del suelo es dejado a un protocolo externo, existe la posibilidad de que los usuarios no se pongan de acuerdo en quién tiene el control y emitan operaciones conflictivas entre sí.

5.3.1.3.4 Detección de dependencias

Se basa en el uso de timestamps para detectar operaciones conflictivas. Dichos conflictos se resuelven manualmente. No se necesita sincronización, y las operaciones no conflictivas pueden ser ejecutadas inmediatamente, resultando en un buen tiempo de respuesta.

5.3.1.3.5 Ejecución reversible

Se ejecuta la operación inmediatamente, pero se guarda información que permite volver al estado anterior. Se define un orden global para las operaciones, y si se detecta que en algún sitio las operaciones fueron ejecutadas en un orden incorrecto, se deshacen los cambios y se reejecuta en el orden correcto. Como desventaja se puede mencionar que al usuario puede ver cambios que luego son revertidos. Por otro lado, un orden total en un sistema distribuido no es algo sencillo de implementar ([24]).

Obviamente todas las operaciones deben tener manera de ser revertidas (una operación de "undo"). Dado que los editores suelen tener soporte para "Deshacer" y "Rehacer", esto no es un problema. Se puede mejorar la concurrencia de este esquema con esta optimización: si dos operaciones que son conmutables son ejecutadas de manera desordenada, no realizar la corrección. Esta optimización está implementada en COAST y Dolphin.

Un problema con este y otras técnicas optimistas es que el sistema puede permanecer mostrando un estado inconsistente durante un lapso pequeño de tiempo. Sucede, por ejemplo, si dos operaciones son emitidas al mismo tiempo, hasta que se realice el "deshacer", y nuevamente la ejecución de las operaciones. El estado inconsistente es visible al usuario, porque, al contrario de las Bases de Datos, la prioridad es el tiempo de respuesta. La mayoría de los editores no soluciona este problema, simplemente se asume que no sucede muy a menudo.

5.3.1.3.6 Transformaciones operacionales

Es una alternativa para mantener la consistencia en ambientes totalmente distribuidos. Con este método no es necesario ejecutar las operaciones en un orden total, ni utilizar bloqueos.

En los algoritmos de OT los cambios son encapsulados en operaciones. Cada máquina genera operaciones que serán ejecutadas localmente y luego distribuidas en todas las demás máquinas. Las operaciones recibidas desde otros sitios son ejecutadas siguiendo algunas reglas estrictas. Todos los algoritmos de OT transforman las operaciones antes de ejecutarlas, para incluir o excluir los efectos de otras operaciones. En el trabajo de Ellis [21] se da el ejemplo de un editor, en el cual un usuario inserta una letra en una posición y un segundo usuario en una posición posterior al del primero. La operación del segundo usuario (digamos Insert('r',7)), debe ser tranformada para contemplar el desplazamiento de un caracter introducido por la primer operación, por lo tanto debería insertar el caracter en la posición 8 (y no 7 como pretendía originalmente).

5.3.1.4 Sincronización

Una arquitectura específica lleva a diferentes modos de separar el modelo abstracto de datos de la aplicación de las vistas generadas a partir de dicho modelo. En un sistema centralizado, las vistas y el modelo están en el mismo lugar, por lo tanto la sincronización es trivial. En contraste, las arquitecturas replicadas mantienen copias de los datos y las vistas en todas las ubicaciones. En el medio está la posibilidad de tener un servidor central de notificaciones, que mantiene los datos, y replicas que deciden como mostrar los datos cuando se le envían notificaciones de cambio.

A nivel de framework, esta separación de modelo y vistas suele ser visible para el programador. La manera que tiene un framework para procesar los eventos del usuario y sincronizar el modelo y las vistas usualmente depende de como es la distribución de vistas y modelos en la arquitectura.

5.3.1.5 Comunicación

El framework debe facilitar la comunicación. El programador no necesita tratar con los problemas de la creación de una conexión entre máquinas. Pero debe tener la libertad de decidir qué se debe comunicar y con qué prioridad.

Modelos centralizados son vulnerables a los cuellos de botella en la comunicación, pues el servidor central debe manejar tanto la entrada como la actualización de todas las pantallas. Los sistemas distribuidos son más eficientes, pues sólo es necesario enviar mensajes cortos que contienen la mínima información indispensable para realizar un cambio de estado.

5.3.1.6 Tolerancia a Fallos

El framework debe proveer un mecanismo de notificación que permita manejar ciertas fallas. Tales fallas pueden ser la pérdida de conexión, retrasos excesivos, etc. Esto obliga al programador a estar consciente de las fallas que son inherentes a un determinado tipo de arquitectura.

5.3.2 Abstracciones de Programación

Los frameworks deben proveer abstracciones para coordinar los diferentes procesos distribuidos, modificar el modelo de datos abstracto común, y controlar y actualizar las vistas. La abstracción suele depender de la arquitectura de tiempo de ejecución elegida.

Existen diferentes arquitecturas de datos compartidos:

3. *Sistema no compartido*, donde ni la vista ni los datos son compartidos por los procesos. El programador debe encargarse de mantener los objetos compartidos, las vistas y la relación entre ellos.
4. *Sistema con modelo compartido:* el modelo está compartido por el sistema. La abstracción del framework especifica cómo acceder al modelo compartido y modificarlo. Las vistas, posiblemente no compartidas, reaccionarán ante cambios del modelo.
5. *Sistema con vistas y modelo compartidos,* donde los cambios se propagan automáticamente de uno al otro.

En las siguientes secciones se describen las abstracciones más comunes.

5.3.2.1 Llamada a Procedimientos Remotos por Difusión (Multicast)

RPC ("Remote Procedure Calls") via multicast es simplemente RPC, pero ejecutando el mismo procedimiento en varias réplicas. Permite a los diferentes procesos distribuidos comunicarse y compartir datos.

Algunos frameworks permiten ocultar detalles de asignación de ruta y comunicación entre los procesos. Por ejemplo, en el caso del código generado a partir de CSSL que se presenta en este trabajo, envía los mensajes a los demás integrantes de la sesión, y es el framework subyacente quien mantiene el registro de las direcciones de red de cada persona.

5.3.2.2 Eventos y notificadores

En esta abstracción el programador puede especificar eventos de interés, y los otros procesos son notificados cuando estos eventos suceden.

Estos eventos pueden ser generados automáticamente (por la arquitectura de tiempo de ejecución) o por el usuario (explícitamente). Al evento se le pueden adherir notificadores (también conocidos como *"observers"* o *"listeners"*).

Se podrían usar, por ejemplo, para notificar cambios en el modelo compartido.

5.3.2.3 Modelos compartidos y vistas

Está basado en la idea de separar los datos de la vista, originada en el Model-View-Controller de Smalltalk (MVC). El framework mantiene un modelo de datos compartido consistente, manejando concurrencia y sincronización. Y puede notificar los cambios en los datos a los procesos o actualizar automáticamente la vista. COAST, Dyce y GroupKit soportan esta abstracción.

En particular, COAST provee un espacio de objetos compartidos, que pueden variar desde simples documentos hasta espacios de información muy complejos. Los programadores pueden crear, acceder, modificar objetos compartidos como si fueran locales. El framework se ocupa de la replicación, sincronización, control de concurrencia y almacenamiento persistente de dichos objetos. Provee un mecanismo de transacciones, pero no mediante un bloqueo pesimista, que sería demasiado caro para este tipo de sistemas. Como ya se mencionó, usa operaciones reversibles. Se ejecuta la operación localmente y se la envía a un mediador, que decidirá si la transacción sera completada o no. Si es rechazada, se deshacen los cambios.

5.3.3 Widgets para Groupware

Otra facilidad que pueden proveer los toolkits groupware son los widgets.

Dado que la intención de este trabajo es en realidad la generación de código a partir de CSSL, y no tiene dentro de su alcance la generación de código para interfaces gráficas, estos serán explicados brevemente aquí. CSSL contiene el concepto de herramienta, pero la especificación de las características de la herramientas debería hacerse con otro tipo de DSL, algo más cercano a los diseñadores de interfaces gráficas que a UML. Eso no impediría que en un futuro dos modelos puedan ser combinables, y CSSL incluya construcciones para soportar este tipo de componentes.

Existen dos clases de widget para groupware: versiones groupware de componentes gráficos monousuario, y componentes gráficos específicos para groupware, que facilitan actividades habituales del trabajo en grupo.

Durante el rediseño de un componente para groupware se presentan nuevos problemas. Según cómo se comparte la interacción de los usuarios se habla de acoplamiento débil o fuerte. Se habla de control de acceso cuando sólo un usuario de un grupo puede usar un componente. Existe bibliografía que trata estos temas, que está fuera del alcance de este trabajo. [34]

5.3.3.1 Acoplamiento

El acoplamiento es definido como los medios por los cuales los componentes de la interfaz comparten el estado de la interacción entre diferentes usuarios. En un acoplamiento fuerte, las acciones en una pantalla se ven reflejadas inmediatamente en las demás ventanas. En un acoplamiento débil, sólo algunos eventos causan esa actualización. En este mismo capítulo existe una definición algo más amplia del término.

En la introducción al framework DyCE [20] se habla de tres métodos de acoplamiento del estado de la aplicación:

- *aplicación compartida* (tambien llamada ventana compartida): la interfaz gráfica ubicada en un servidor centralizado es distribuida a todos los usuarios. A su vez, los eventos de cada uno de los usuarios son enviados a dicho servidor. Esto implica un uso elevado de ancho de banda. La aplicación queda restringida al un estricto WYSIWIS ("What You See Is What I See") y una colaboración sincrónica.
- *Eventos de Interfaz compartidos:* en este caso cada usuario ejecuta una instancia de la aplicación en su computadora. Los eventos de la interfaz gráfica son capturados y distribuidos a todos los demás usuarios. Claramente las aplicaciones que ejecutan cada uno de los usuarios son similares, y el procesamiento de los mismo eventos debe llevar al mismo resultado visual. Este enfoque es similar al anterior, pero reduce el ancho de banda.
- *Datos compartidos:* en este caso también los usuarios ejecutan su propia instancia de la aplicación. Cada una de estas instancias accede a un conjunto de datos compartidos. Esos datos pueden estar en un servidor central o replicados. Permite que varias herramientas accedan al mismo modelo compartido. La colaboración puede ser sincrónica o asincrónica.

Las dos primeras son implementaciones que llevan a un sistema de acoplamiento fuerte. La última permite un acoplamiento débil.

5.3.4 Administración de Sesiones

Es importante que el framework soporte un conjunto de funcionalidades y diferentes modalidades para gestionar sesiones. No es bueno que se fuerce una única modalidad para el manejo de sesiones en la aplicación.

Un gestor de sesiones debe:

- crear nuevas conferencias
- nombrar conferencias
- eliminar conferencias
- encontrar conferencias en actividad
- determinar quién está participando en una conferencia
- unir gente a la conferencia
- controlar el acceso a la conferencia
- permitir el ingreso tardío (latecoming)
- permitir retirarse de la conferencia
- decidir si la conferencia persiste al retirarse sus participantes

Los gestores de sesión pueden implementar alguna política, por ejemplo:

- **Política rudimentaria:** donde el usuario es quién debe decidir con quién conectarse, a veces especificando información de bajo nivel, como direcciones de red.
- **Política de puerta abierta:** es una política permisiva que permite pensar en términos de conferencias y participantes, en lugar de direcciones de red. Se puede ingresar a una conferencia de varios modos: simplemente unirse, ser invitado o crear uno mismo la conferencia.
- **Puntos de encuentro**: provee puntos de encuentro. La gente va hacia un lugar, y se conecta a la sesión correspondiente a ese lugar. Es la metáfora basada en "Habitaciones".
- **Otros:** por ejemplo, basado en documentos (dos personas que abren el mismo documento forman una sesión). Otro ejemplo: una sesión donde el director de la misma puede invitar y eliminar participantes.

5.4 Google Wave

En el momento de finalizar este trabajo encuentro que Google está realizando esfuerzos por establecer su "Google Wave Federation Protocol" [36].

Google wave es una plataforma de comunicación y colaboración basada en documentos almacenados en sus servidores (llamados "waves"), que proveen modificaciones concurrentes y modificaciones con latencia muy baja.

Pretende ser una plataforma abierta, donde cualquiera podría ejecutar servidores de "waves" y convertirse en proveedores de "waves".

Los datos en la plataforma se organizan de esta manera. Existen waves, wavelets e identificadores. Un wave está conformado por varios wavelets. Cuando un usuario tiene acceso a un wavelet se dice que es un participante del wavelet. Se mantienen copias del wavelet en todos los proveedores que tengan un usuario participando en el wavelet. Una de los proveedores contiene la "copia definitiva" del wavelet, y se dice que esta "hosteando" el wavelet.

Un proveedor opera un servicio de wave en uno o mas servidores. Las piezas centrales del servicio de wave son el almacenamiento de waves y el servidor de waves, y este último es quien resuelve las operaciones sobre los wavelets a través de transformaciones operacionales.

Parece una plataforma interesante para trabajos futuros en esta materia, y como continuación de este trabajo. Su modelo de datos y su mecanismo de interacción parecen compatibles con los implementados en el plugin desarrollado y con el **lenguaje CSSL**.

5.5 Conclusión

En este capítulo se introdujo el concepto de sistema colaborativo, para seguir luego con un conjunto de conceptos básicos de la materia.

De ahí en más, se comenzó a analizar el desarrollo de groupware. Y se concentró la atención en dos puntos fundamentales para este trabajo: qué cosas le preocupan al desarrollador de sistemas colaborativos y como los frameworks pueden ayudarlo.

Se analizaron los problemas inherentes al desarrollo de estos sistemas, especialmente los mas complejos como el control de concurrencia. También se le dio importancia a las abstracciones que proveen los frameworks para el manejo de objetos compartidos.

La pregunta del lector será entonces: ¿Por qué esos temas y no otros?

Este trabajo trata de la generación de código a partir de CSSL, un lenguaje gráfico que permite representar un sistema colaborativo. CSSL es un lenguaje específico de dominio para groupware. La generación de código a partir de un DSL suele realizarse manteniendo varias capas: un generador de código, un framework, una plataforma, y la máquina, en ese orden.

Esta generación de código, por lo tanto, necesita un framework. Y existen muchos frameworks para sistemas colaborativos.

Una manera de encarar el tema sería investigar varios de ellos por separado, elegir uno, y generar código que use ese framework para alivianar la tarea del generador. El inconveniente de esto es que CSSL contiene ciertas abstracciones que no son soportadas en forma exacta por ningún framework. Entonces deberíamos escribir un pequeño framework que actúe como un wrapper alrededor del framework destino, y que el código generado utilice las interfaces provistas por este nuevo framework. Investigar en profundidad un único framework, olvidándose de los demás, podría hacer que el diseño del nuevo framework pase a depender fuertemente de características del framework destino.

La segunda alternativa es apoyar la generación de código sobre la interfaz de un framework que considere ampliamente los problemas de este dominio, y que eventualmente delegue funcionalidad en framework ya escrito.

En la primera implementación de este generador de código, se escribieron funcionalidades básicas para el manejo de usuarios, manejos de sesiones, multicast de operaciones. Todo está basado en las investigaciones de este capítulo. La implementación inicial no utiliza un framework ya escrito (COAST, Dyce, GroupKit, Habanero, etc..), sino que simplemente implementa la funcionalidad. Por supuesto que eso lo hace menos potente, confiable y robusto. Pero el objetivo del trabajo no es el desarrollo de un framework que cubra exhaustivamente las necesidades de CSSL de manera robusta.

Si cotejamos la implementación del framework con los conceptos vertidos en estas secciones, podremos ver que algunos módulos sugeridos por el texto están implementados. Se puede ver, módulo por módulo y funcionalidad por funcionalidad, qué es lo que está implementado y cómo se puede expandir el framework para soportar funcionalidad no soportada. Pero esta no es una tarea a analizar en la conclusión de este capítulo, sino en un capítulo siguiente.

6 GENERACIÓN DE CÓDIGO

6.1Sobre la Generación de Código

Para la realización del trabajo se evaluaron varias herramientas. Y se generó código de CSSL utilizando tanto Mofscript como JET, dos herramientas de transformación de modelo a texto. Esta sección comienza con un estudio de las transformaciones de modelo a texto. Se repasan los tipos de generadores, y luego las posibles maneras de ingresar código manual al código generado, permitiendo una posterior regeneración sin la pérdida de los cambios manuales.

Al terminar con esta visión general, se describen brevemente Mofscript y JET.

Luego, se relata la experiencia del desarrollo con cada una de estas herramientas, incluyendo los inconvenientes y ventajas.

6.2 Transformaciones

Como ya se mencionó, MDA propone realizar modelos en las fases iniciales del proyecto de desarrollo de software, e ir transformando dichos modelos automáticamente en cada una de las etapas de desarrollo.

Se define, en primer lugar, el PIM ("Platform Independent Model"), que luego será transformado a uno o más Modelos Específicos de Plataforma (PSM,"Platform Specific Model"). Para realizar esta transformación existen muchas *herramientas de transformación de modelo a modelo*, que ayudan a la conversión de los conceptos del metamodelo origen al metamodelo destino.

Finalmente, se puede tranformar un PSM a código. Para dicha transformación existen las *herramientas de transformación de modelo a texto*.

6.2.1 Herramientas de Transformación de Modelo a Modelo

Repasamos algunas herramientas de transformación:

VIATRA [5] (VIsual Automated model TRAnsformations) framework: Provee un lenguaje textual para describir modelos y metamodelos, y transformaciones llamados VTML y VTCL respectivamente. La naturaleza del lenguaje es declarativa y está basada en técnicas de descripción de patrones, sin embargo es posible utilizar secciones de código imperativo.

Epsilon: es una plataforma desarrollada como un conjunto de plug-ins (editores, asistentes, pantallas de configuración, etc.) sobre Eclipse. Define el lenguaje *"Epsilon Transformation Language"* (ETL), para transformación de modelos.

Kent Model Transformation language:desarrolladas en la Universidad de Kent. La semántica del lenguaje de transformación esta basada en el lenguaje *"Relations"* de QVT.

Tefkat: define un lenguaje propio con una sintaxis concreta parecida a SQL

ATL (Atlas Transformation Language): forma parte del framework de gestión de modelos AMMA. Utiliza un lenguaje propietario llamado ATLAS para definir transformaciones.

UMLX: es una sintaxis concreta gráfica para complementar el lenguaje de transformación de modelos OMG QVT.

AToM3 (A Tool for Multi-formalism and Meta-Modeling): Las dos tareas principales de AToM3 son metamodelado y transformación de modelos. Los formalismos y modelos están descriptos como grafos. Las transformaciones de modelos están ejecutadas por la reescritura de grafos. Las transformaciones pueden entonces ser expresadas declarativamente como modelos basados en grafos.

MOLA (MOdel transformation LAnguage): consiste de un lenguaje de transformación de modelos y de una herramienta para la definición y ejecución de transformaciones. El lenguaje para la definición de la transformación Mola es un lenguaje gráfico.

Kermeta: abreviatura de "Kernel Metamodeling". Fue diseñado para escribir modelos, para escribir transformaciones entre modelos y para escribir restricciones sobre estos modelos y ejecutarlos.

6.2.2 Herramientas de Transformación de Modelo a Texto

Dado que el objetivo primordial de este trabajo es la generación de código a partir de un DSL, estamos especialmente interesados en las herramientas de tranformación de un modelo a Texto.

En ese mismo sentido, al saber que CSSL está definido con EMF, una implementación de MOF, nos interesan particularmente los lenguajes que permitan manipular modelos definidos a través de MOF. De ahi nuestro interés por Mofscript y JET.

Según Kelly y Tolvanen ([3]), se distinguen principalmente dos maneras de transformar un modelo a código: basada en Templates (plantillas) y basada en Visitors (Visitante, en referencia al patrón de diseño OO). Aunque también existen alternativas, como simplemente acceder al modelo de manera programática o los generadores del tipo "Crawler".

Cuando se trata de generar código a partir de la lectura del modelo, conviene que exista una API para leer el modelo, y así evitar ensuciar el generador con los detalles de implementación del acceso al modelo.

El tipo más simple de generador es el "model visitor". En este caso, se mapean elementos en el modelo origen a elementos en el modelo de salida. El generador

"visita" cada elemento del modelo, llamando a un determinado código de generación para cada clase de elemento, como en el patrón de diseño "Visitor".

Una variante de lo anterior sería hacer el mapeo en tres fases. El primer paso convertiría los elementos del modelo a conceptos intermedios, correspondientes a un conjunto de lenguajes. El segundo paso convertiría esos objetos en sus representaciones en un determinado lenguaje. Finalmente el tercer paso generaría el código en ese lenguaje. Para los generadores de DSM, por lo general, un generador de un sólo paso no es suficiente.

Una plantilla es el código que se quiere como salida, pero con partes que varían según el modelo, y que son reemplazadas por comandos encerrados por un par de caracteres de "escape". Por ejemplo, JSP ("Java Server Pages") es una tecnología de templates. Los generadores basados en plantillas son los mas conocidos. Algunos ejemplos son: *JET* ("Java Emmiter Templates"), el motor *DSLTools T4* de Microsoft, y el lenguaje de script *CodeWorker*.

Un generador DSM tiene dos tareas: navegar y leer el modelo, y generar texto basado en el modelo. Usar "visitors" requiere que los elementos del modelo origen y destino coincidan uno a uno. Las plantillas fuerzan a que la navegación esté subordinada al formato de salida. Un tercer tipo de generador nos da más libertad: los de tipo "Crawler".

Un crawler mantiene y opera sobre dos pilas en la generación: una para el elemento del modelo actual y otra para los streams de salida. En cada momento existe un elemento actual y un archivo actual.

6.2.3 Integrando código escrito a mano

La situación ideal en un DSL es la generación completa de código. Nunca se debería editar el código generado. Se debería modificar el lenguaje de modelado para evitarlo. Pero a veces no es posible. En ese caso, los cambios realizados sobre el código generado, no pueden ser perdidos en una regeneración.

El código escrito a mano puede ser integrado de tres maneras:

5. **Regiones protegidas**: es una sección de un archivo generado que el generador sabe que se puede editar a mano. En una regeneración, el generador lee el archivo generado previamente, y si está modificado manualmente no vuelve a generar esa sección. Las regiones son, por lo general, delimitadas por comentarios especiales. A veces los comentarios tienen el checksum del código que contienen para saber si fueron modificados. Provocan que el desarrollador deba versionar el modelo y el código generado. Hay mejores maneras de obtener el mismo resultado.
6. **Código escrito a mano en el modelo:** debe ser evitado, pero cuando el código escrito a mano es una necesidad, ponerlo en el modelo deja la información versionable del sistema en un sólo lugar. Si los textos son muy largos o demasiados, tienden a reducir la eficiencia de un DSL. Además, una mayor cantidad de código se beneficiaría con el uso de una IDE. Para eso existe la siguiente opción.
7. **Archivos referenciados en el modelo:** cada pieza de codigo se externaliza en su propio archivo y se referencia desde el modelo. Por lo tanto, el código puede ser editado en una IDE.

6.2.4 Mofscript

La herramienta MOFScript permite la transformación de cualquier modelo MOF a texto. Es decir, permite la generación de código Java, EJB, JSP, C#, SQL Scripts, HTML o documentación a partir de los modelos. MOFScript es un lenguaje de transformación basado en QVT. La herramienta está desarrollada como un plugin de Eclipse, el cual soporta el análisis sintáctico, la validación y la ejecución de scripts escritos en dicho lenguaje.

Algunas características de MOFScript son:

- Generación de texto a partir de modelos basados en MOF. La habilidad de generar texto a partir de cualquier modelo basado en un metamodelo definido usando MOF, como por ejemplo, modelos UML.
- Sentencias de control de flujo: la habilidad de especificar mecanismos de control básicos como loops y sentencias condicionales.
- Manipulación de strings con operaciones básicas.
- Permite producir recursos (archivos) como salida. Permite especificar el archivo de salida para la generación de texto.
- Permitirá hacer *traceability* entre los modelos y el texto generado. Esta propiedad permite hacer una regeneración.
- El editor de scripts posee ayuda para completar el código.
- La ingeniería inversa todavía no es parte de la herramienta.

Los archivos que definen la transformación tienen como extensión '.m2t'.

El lenguaje de transformación MOFScript esta basado en el lenguaje QVT. MOFScript es un refinamiento del lenguaje operacional QVT. Es un lenguaje textual, basado en objetos y usa OCL para la navegación de los atributos. Además, presenta algunas características avanzadas, como la jerarquía de transformaciones.

Otra característica avanzada es "traceability", útil para saber que partes de código fueron generadas a partir de qué elementos del modelo, y, por ejemplo, no sobreescribir código modificado manualmente. Desafortunadamente la guía del usuario de Mofscript (Version 0.6), declara, en sus secciones 2.4 y 4.6, que la implementación no está finalizada. Habilitar esta característica solo hará que se impriman unos mensajes de depuración. Todavía no hay soporte para regiones protegidas. No encuentro una versión más actual de la guía de usuario, que tiene fecha dos de Noviembre de 2006, y por lo tanto tengo que suponer que a la fecha aún no fue implementado, aunque la guía pertenece a la versión 1.1.11 y la versión actual es 1.3.4.

Dado que este trabajo no preve la generación total de código a partir del metamodelo, pero sí la regeneración sin sobreescribir los cambios del programador, esto es un punto en contra muy importante para este lenguaje. Y si esta capacidad estuviera implementada y no documentada, eso dificultaría su uso y nos llevaría a usar algún otro mecanismo para la regeneración.

6.2.5 JET

JET es el acrónimo de "Java Emmiter Templates".

JET opera sobre una entrada consistente en un único objeto Java, y sus sentencias son escritas en Java. Sólo puede ejecutarse dentro de Eclipse, y fue

diseñado para generar clases Java. Usa "<%", "%>", "<%!" como caracteres de escape, y estos encierran expresiones Java que son añadidas a la salida.

JET corre en dos fases: traducción y generación. La primera fase traduce las plantillas a un programa Java. La segunda fase invoca el programa de la primera fase para generar la salida.

Según "EMF: a Developer's Guide" [7], en su capítulo 11, el framework utilizado por EMF para la generación de código es JET. En ese mismo capítulo se da la siguiente plantilla de ejemplo:

```
<%@ jet
    imports="java.util.* org.eclipse.emf.codegen.ecore.genmodel.*"%>
<%@ include file="../Header.javajet"%>
<%GenClass genClass = (GenClass)argument;
  GenPackage genPackage = genClass.getGenPackage();%>

package <%=genPackage.getInterfacePackageName()%>.validation;
import org.eclipse.emf.validation.EMFValidator;

public interface <%=genClass.getInterfaceName()%>Validator
  extends EMFValidator
{
<%for (Iterator i=genClass.getGenFeatures().iterator();
       i.hasNext();) {
  GenFeature f = (GenFeature)i.next();%>
<%if (f.isChangeable()) {%>
    boolean validate<%=f.getCapName()%>(<%=f.getType()%> value);
<%}%>
<%}%>
}
```

6.3 Resumen

El objetivo de este capítulo fue introducir, de manera teórica, algunos conceptos que se utilizarán en la generación de código en el próximo capítulo.

En la primera parte de este capítulo se vieron conceptos generales sobre la generación de código. Se clasificaron las herramientas para generar código según cómo recorren el modelo y van generando los artefactos. También se vieron las distintas maneras de mezclar el código escrito a mano con el código generado, de manera tal que el segundo no sobreescriba el primero y obligue al desarrollador a reescribirlo.

También se discutieron las caraterísticas principales de algunas herramientas, y se indagó con algo más de profundidad sobre las herramientas Mofscript y JET.

7 GENERACIÓN DE CÓDIGO DE CSSL

Anteriormente se vieron conceptos generales sobre la generación de código. Se clasificaron las herramientas para generar código y se realizó una breve introducción sobre Mofscript y JET.

En lo que resta de este capítulo se tratará **el diseño de la generación de código**. Se explicará el funcionamiento y la estructura de los principales módulos que componen el plugin. También se tratará sobre la interacción con las librerías, frameworks y herramientas necesarias para la generación. En particular, Mofscript y JET son las herramientas para asistir a la generación, EMF es el framework para leer el modelo y PDE es el "ambiente de desarrollo de plugins" de Eclipse.

7.1 Visión General

La generación está escrita como un **plugin de Eclipse**. Como consecuencia, está desarrollada en el lenguaje Java, utilizando el framework provisto por Eclipse para el desarrollo de plugins, llamado PDE.

Su utilización es simple. A partir de un archivo "cm" que contiene un modelo CSSL, el plugin genera tres proyectos dentro del workspace del usuario. Estos tres proyectos contendrán el código para iniciar el desarrollo de la aplicación colaborativa. Un proyecto contendrá las clases del servidor, otro las clases del cliente, y un tercero tendrá las clases en común entre cliente y servidor. Los detalles del funcionamiento son explicados en un capítulo posterior.

Para la generación, el plugin se apoya en dos **herramientas**: **Mofscript y JET**. En el caso de la generación de las operaciones de una sesión, se utiliza Mofscript. En todo el resto JET. Mofscript lee elementos del modelo, y en los elementos de una determinada clase comienza la ejecución, siguiendo la regla "main". Las demás reglas se siguen por invocaciones desde las reglas principales. JET, en cambio, puede generar código a partir de cualquier objeto que se le pase por parámetro. La versión 2 de JET también puede leer modelos en EMF y XML. Sin embargo, en el código se usa la primera versión de esta librería. JET está basado en plantillas, y es muy similar a JSP; esa es su principal ventaja, prácticamente no hay curva de aprendizaje.

Para la **lectura del modelo** se utiliza, simplemente, código Java acompañado por la librería EMF. La generación de código lee completamente el modelo, y genera una **estructura intermedia** en memoria. Los elementos de dicha estructura intermedia son enviados a las plantillas, que generan el texto. Esta decisión ***se fundamenta de la siguiente manera***: un generador tipo "template" requiere una correspondencia uno a uno entre el artefacto generado y la plantilla. El parámetro de entrada para la plantilla es un único objeto. Se podría dar el caso de que un conjunto de artefactos se generaran a partir de un conjunto de elementos en el modelo del dominio, creando una relación de "muchos a muchos". El recorrido de los mismos elementos del dominio por cada artefacto a generar no es deseable, por razones de rendimiento, y, quizá más importante, por razones de claridad del código de la plantilla.

El texto generado no es escrito directamente a disco. El plugin calcula la ruta del archivo destino del texto generado, y verifica si existe. Si no existe, simplemente

crea el archivo, utilizando la clase "FileUtils". Sin embargo, existe la posibilidad de que exista, y que haya sido modificado por el usuario. Aquí entra en juego **JMerge**. JMerge es una utilidad de EMF (y sobre la cual la generación de código de EMF está basada) que nos **permite tomar dos clases java y mezclarlas**, en base a parámetros descriptos en un archivo de configuración ("merge.xml"). El archivo de configuración utilizado en este plugin está tomado del mismo EMF, por lo tanto las reglas de mezclado podrían simplificarse a esta explicación: para saber si el usuario modificó un método, se revisa el javadoc en busca del texto "@generated". Si se encuentra una línea que contenga ese comentario, se asume que el usuario no modificó el texto de ese método. Si, por el contrario, el comentario no existe o fue reemplazado por "@generated NOT", se sabe que el usuario modificó el código del método. JMerge se encarga de reescribir el código en métodos no modificados por el usuario, y deja intactos los métodos modificados.

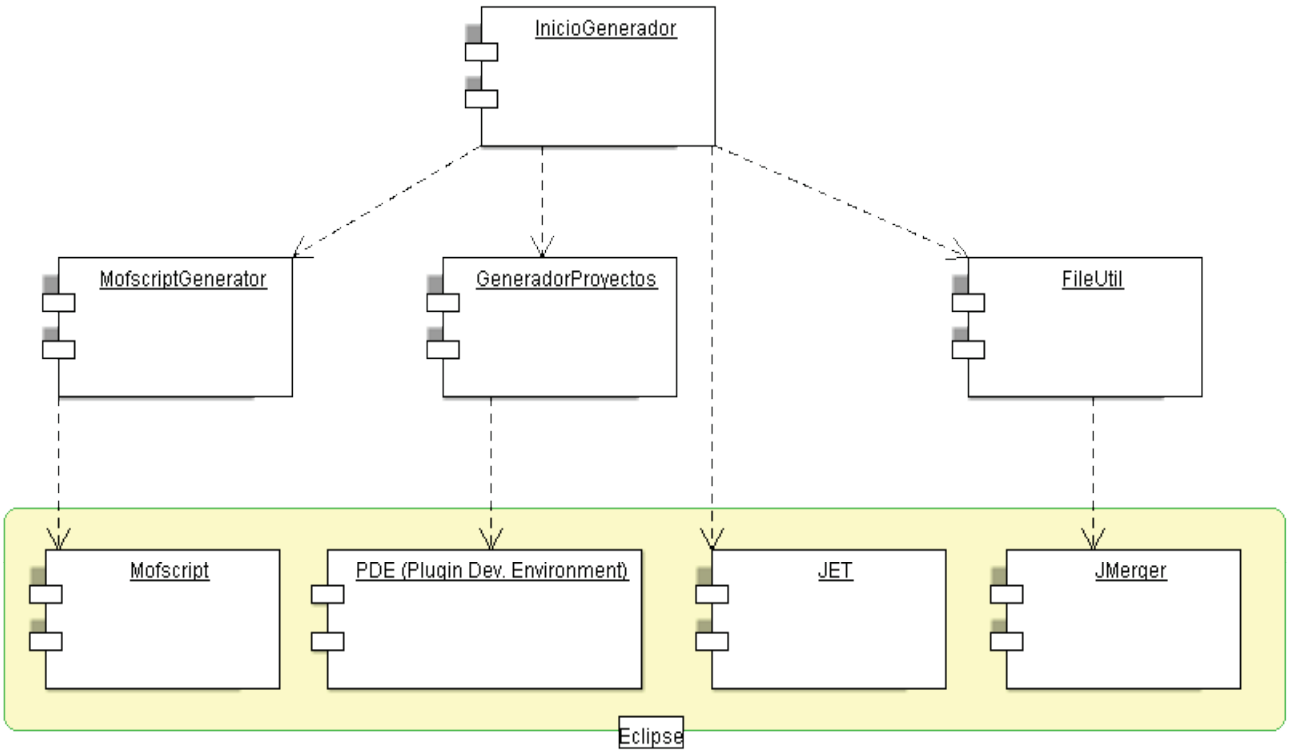


Ilustración 6-13. Estructura de los principales componentes de la generación

7.2 Estructura del Plugin

En esta sección se describirán los proyectos Java que componen el plugin, junto con sus componentes principales.

Proyectos:

- **GeneracionCodigo**: este proyecto contiene la mayor parte de la lógica de generación, y será examinado con mayor detenimiento en lo que resta de esta sección.

- **CommonLib**: Se utiliza para almacenar algunas librerías comunes a los otros proyectos. La extensión del trabajo no justifica usar maven u otro gestor de dependencias.
- **BotonGenerate**: contiene sólo la lógica para hacer funcionar el menú contextual que posibilita la generación a partir de un archivo. Luego delega en el proyecto "GeneracionCodigo". Aquí se utiliza el punto de extensión provisto por eclipse para realizar la tarea mencionada. Es el único de estos tres proyectos que aporta elementos a la interfaz gráfica, es decir, el menú contextual y los diálogos de avance e información.

Dentro del proyecto "GeneracionCodigo" se ven los siguientes elementos:

- **src**: contiene el código normal
- **srcjet**: contiene el código traducido automáticamente a partir de los templates
- **templates**: contiene las plantillas JET

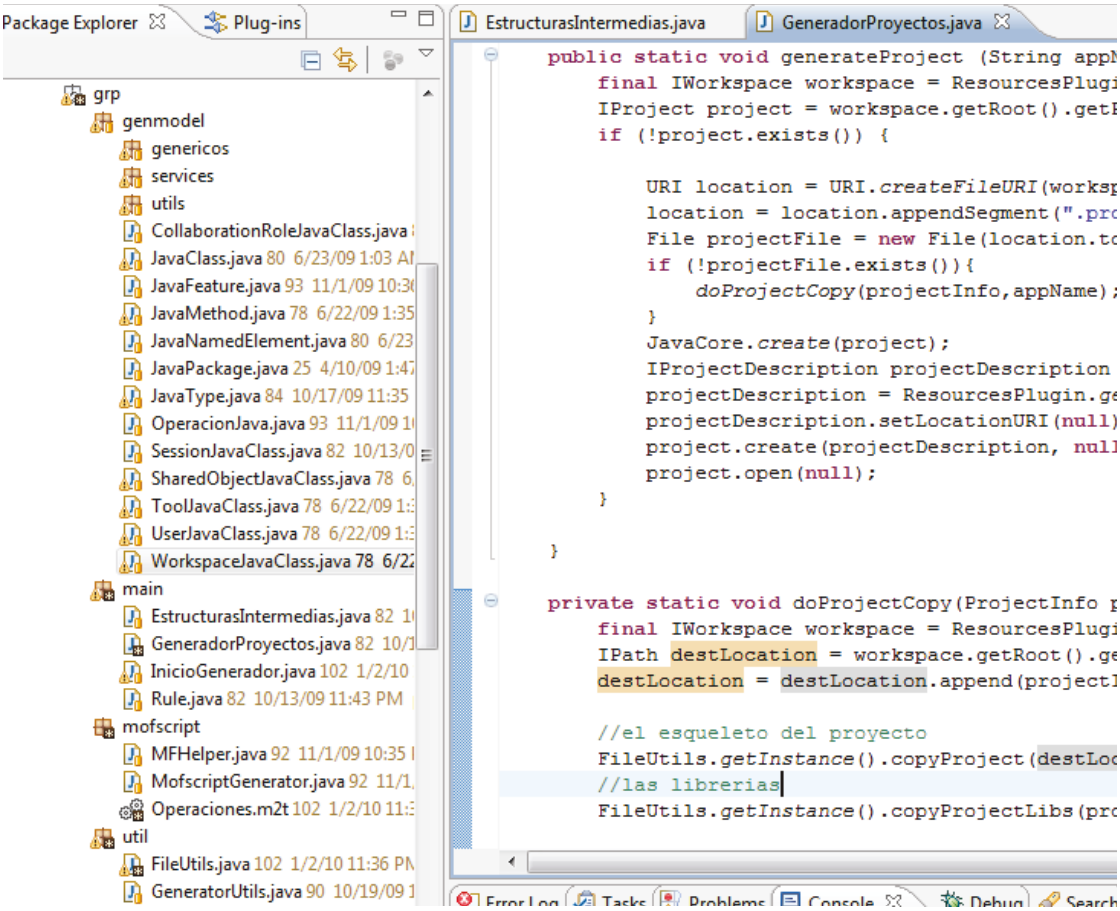


Ilustración 14: Codigo estandar del plugin

7.2.1 Principales clases del plug-in

En la figura se muestran algunas clases que están dentro de la carpeta "*src*". Aquí se repasan las características salientes de cada una de ellas.

GeneradorProyectos: Interactúa con PDE (especialmente el plugin de recursos) para generar los tres proyectos (SERVER,COMMON,CLIENT). Si ya existen no hace nada. Los crea en la raíz del espacio de trabajo, configura sus builders, sus carpetas de código, y luego los abre. En una sección posterior se repasan las nociones básicas de PDE, las cuales este componente se encarga de abstraer al resto del plugin.

EstructurasIntermedias: Encargada de construir las estructuras intermedias a partir del modelo. Su método principal es "`generarEstructurasIntermedias`", `que recibe la raíz del modelo EMF`.

InicioGenerador: Clase principal de la generación. El método principal, "generar", recibe un archivo (IFile). Se encarga de leer el modelo desde el archivo utilizando EMF y pedirle a "EstructurasIntermedias" que genere los objetos que luego le proveerá a las plantillas JET.

MofscriptGenerator: encargado de la generación de operaciones. Se detalla su funcionamiento en otra sección.

genmodel.JavaClass, genmodel.*: Son las clases intermedias que reciben las plantillas JET. Se ocupan de obtener los datos que las plantillas requieren, si es necesario recorriendo el modelo original. Existen clases diferentes para los diferentes artefactos a generar. Por ejemplo: roles, sesiones, herramientas, sesiones, la configuración spring (XML) de los servicios asociados a las sesiones, etc..

FileUtils: Utilidades para la generación de los archivos correspondientes, por ejemplo, a las clases. Se encarga del "merge" entre el texto que existe en disco y el que se le pide que escriba en disco. Es quien resuelve, según el tipo de artefacto a generar, en qué proyecto debe estar.

7.2.2 Plantillas

En la carpeta "templates" se encuentran las plantillas JET. Por cada tipo de artefacto a generar existe una plantilla, por ejemplo:

- ApplicationStart.javajet
- CollaborationRoleEntityGenerator.javajet
- SessionEntityGenerator.javajet
- SharedObjectEntityGenerator.javajet
- ToolEntityGenerator.javajet
- UserEntityGenerator.javajet
- WorkspaceEntityGenerator.javajet

Las subcarpetas son las detalladas a continuación y dividen los tipos de artefactos a generar:

- **client**
- **factory**
- **genericos**: algunas plantillas que no dependen del tipo de artefacto a generar.

- **gpfw**: es el código del framework. Recuerde que DSL recomienda que el código generado utilice un framework de dominio para quitarle complejidad al generador. Los detalles del framework están descriptos en un capítulo posterior.
- **partes**: plantillas reutilizables por otras plantillas, que producen elementos comunes, por ejemplo "*StandardFeaturesGenerator.javajet*"
- **projects**: contiene los recursos que se necesitan copiar en los proyectos, y que no necesitan ser modificados.
- **services**: contiene las plantillas necesarias para generar clases, interfaces y configuración spring (xml) para exponer la interfaz remota del servicio.

7.3 Herramientas Usadas

7.3.1 MOFSCRIPT (Generación de las operaciones)

Las operaciones realizadas sobre una sesión se representan en CSSL a través de la metaclase "*Operation*". Las operaciones incluidas dentro de una sesión tienen una generación especial, dado que, en lugar de generarse simplemente un método, se genera una clase por cada operación.

La clase generada es subclase de *AbstractSesionOperation*. Veamos un ejemplo:

```
public class TrazarLnea extends AbstractSessionOperation implements IOperation {
    /**
    * @generated
    */
    public TrazarLnea(gen.model.Recta dibujo) {
       this.dibujo = dibujo;
     }

    /**
    * @generated
    */
    private gen.model.Recta dibujo;
    /** @generated */
    public gen.model.Recta getDibujo(){     return dibujo;}

    /**
    * @generated
    */
    public void setDibujo(gen.model.Recta myVar) {     this.dibujo = myVar;}

    /**
    * @generated
    */
    public boolean execute() throws GPFWException {
          //TODO: implementar
          return true;
      }
}
```

La generación de las operaciones se realiza por medio de esta transformación Mofscript:

```
/**
 * Transformation
 */
texttransformation OperacionTransformation (in cssl:"http://cm/1.0") {
  /**
   * Main (entry point)
   */
  cssl.Session::main () {
    self.features->forEach(f: cssl.Operation){
      f.writeFile(self);
    }
  }

//COMIENZO GENERACION OPERACION
  cssl.Operation::writeFile (s:cssl.Session){
    stdout.println("generando:" + normalizarNombreFeature(self.name))
    var nomClase: String  = normalizarNombreFeature(self.name).firstToUpper();
    file(nomClase + ".java");
'package gen.remoting.session.ops;
.....

public class ' + nomClase + ' extends extends AbstractSessionOperation implements IOperation {
    /** @generated */
    public ' + nomClase + '('
    var isFirst : Boolean  = true;
     self.parameters->forEach (p: cssl.Parameter){
          if (isFirst){
              isFirst = false
          }else{
            ' , '
          }
          p.writeParamString()
     }
     ') {
     '
          self.parameters->forEach (p: cssl.Parameter){
              var nomProp:String = normalizarNombreFeature(p.name).firstToLower();
          '
          this.' nomProp ' = '   nomProp ';'
          }
     '
     }


     '
    self.parameters->forEach (p: cssl.Parameter){
      '//inicioParametro
      ' p.writeFeatureString()
      '' //getFeatureString2(p)
      '//finParametro
      '
    }
    '
    /** @generated */
    public boolean execute() throws GPFWException {
        //TODO: implementar
```

```
        return true;
    }
}
    '
  }
//FIN GENERACION OPERACION

//REGLAS AUXILIARES:

  module::normalizarNombreFeature(st: String): String {
    var tmp:String = st.replace(".*->", "");
    tmp = tmp.replace("[^_0-9a-zA-Z]*", "");
    tmp = tmp.replace("^([0-9]*)", "");
    result = tmp;
    //result = java ("grp.util.StringUtils","normalizarNombreFeature" , st
    //,  "{thisPluginPath}/bin/;{pluginsEclipse}/org.eclipse.emf.ecore_2.4.1.v200808251517.jar"
    //)
  }

  cssl.Parameter::writeFeatureString () {
    var nFeat: String = normalizarNombreFeature(self.name).firstToLower();
    '/** @generated */
    private ' self.type.toFQJava()  ' '  nFeat ';
    /** @generated */
    public ' self.type.toFQJava() ' get' nFeat.firstToUpper() '(){ '
    '    return ' nFeat ';'
    '}

    /** @generated */
    public void set' nFeat.firstToUpper() '(' self.type.toFQJava() ' myVar) { '
    '    this.' nFeat ' = myVar;'
    '}'
  }

  cssl.Parameter::writeParamString () {
    var nFeat: String = normalizarNombreFeature(self.name).firstToLower();
    self.type.toFQJava()  ' '  nFeat "
  }


  cssl.CollaborativeElement::toFQJava ():String {
     result = 'gen.model.' + self.name
  }

  cssl.SharedObject::toFQJava ():String {
     result = 'gen.model.' + self.name
  }

  cssl.DataType::toFQJava ():String {
     result = self.name
  }
}
```

Mofscript permite expresar la transformación con mucha **claridad**. En el programa principal se usa uno de los iteradores del lenguaje, y por cada elemento (por cada operación dentro de la sesión) se llama a la regla "*writeFile*" del elemento "*cssl.Operation*".

El código escrito arriba funciona correctamente, pero para llegar a él tuve que superar algunos inconvenientes en el uso de esta herramienta. La versión que estoy utilizando es 1.3.5, sobre un Eclipse Ganymede (3.4.1). La fecha de las pruebas se encuentra entre 1/Oct/2009 y 1/Nov/2009.

El primer problema de Mofscript es su **escasa documentación**. La guía de usuario no se actualiza desde hace varios años. La versión 0.6 de dicho documento pertenece a la versión 1.1.11 y la versión actual de la herramienta es 1.3.5.

Esta declara que la característica llamada "traceability" (el soporte para regiones protegidas) no está funcionando. Eso no me impide usar la herramienta, porque puedo generar código y utilizar JMerge para implementar las regiones protegidas. Esto sólo es posible porque el código generado es Java.

Pero eso me obliga a ejecutar la transformación dirigiendo los resultados a una carpeta temporal, tomar los resultados y mezclarlos con el código generado anteriormente.

Mofscript no es la única herramienta utilizada para la generación, también estoy usando EMF para leer el modelo y JET para generar algunos artefactos.

Y existen métodos útiles para el manejo de identificadores que ya están escritos en Java, por lo tanto quise llamarlos desde Mofscript. La documentación declara que eso es posible.

La sintaxis es: java (nombreClase, nombreMétodo, parametros, classpath)
Por ejemplo:

```
module::normalizarNombreFeature(st: String): String {
  result = java ("grp.util.StringUtils","normalizarNombreFeature" , st , null
  // "{thisPluginPath}/bin/;{pluginsEclipse}/org.eclipse.emf.ecore_2.4.1.v200808251517.jar"
  )
}
```

En el ejemplo, se intenta llamar al método "normalizarNombreFeature". Pero el resultado es este mensaje:

Java Class Not Found: grp.util.StringUtils

Al hacer el mismo llamado reemplazando el *"null"* en el classpath por *"{thisPluginPath}/bin/; {pluginsEclipse}/org.eclipse.emf.ecore_2.4.1.v200808251517.jar"* funciona. Pero no hay forma de deducir ese classpath desde Mofscript.

La transformación es ejecutada desde dentro del plugin, y por lo tanto debería tener acceso a las clases del mismo. Por un momento pensé en revisar el código para ver que podía estar causando esto. El código en cuestión es:

FuncionCallEvaluator.java:

```
.....
if (classPath != null) {
    String [] paths = classPath.split(";");
    URL[] urls = new URL[paths.length];
    for (int i = 0; i < paths.length; i++) {
        File pfile = new File (paths[i]);
        try {
            java.net.URI uri = pfile.toURI();
            urls[i] = uri.toURL();
        } catch (MalformedURLException urlEx) {
            _env.notifyMessage("Malformed URL: " + urlEx);
        }
    }
    loader = new URLClassLoader (urls, ClassLoader.getSystemClassLoader());
} else {
   loader = ClassLoader.getSystemClassLoader();
}
// String clPath = System.getProperties().getProperty("java.class.path");
//clPath.toString();
```

Las clases son cargadas usando *"loader.loadClass(className);"*.

Aparentemente el classLoader obtenido con *"ClassLoader.getSystemClassLoader()"* no es el classLoader que conoce al plugin, sino conocería la clase "*grp.util.StringUtils*".

El código abierto tiene esta virtud. Aún sin documentación apropiada podemos suponer la causa de un error. Pero esto deja de ser útil si lo único que provee el autor como código fuente es el repositorio subversion, y no utiliza mecanismos de etiquetado para identificar las versiones. Así, si no se puede compilar la revisión actual (HEAD), tampoco se puede tomar el código que dio lugar a una versión en particular (en este caso 1.3.5).

7.3.2 JMerge

JMerge es una herramienta de código abierto contenida dentro de EMF ("Eclipse Modelling Framework"). JMerge permite modificar el código generado sin que esos cambios sean eliminados en la **regeneración**. Esta herramienta funciona únicamente con código Java.

JMerge puede ser utilizado desde dentro de un plugin. Aqui se adjunta código de ejemplo:

```
 // ... JMerger merger = getJMerger();
// especificar origen
merger.setSourceCompilationUnit(merger.createCompilationUnitForContents(generated));
// especificar destino
merger.setTargetCompilationUnit( merger.createCompilationUnitForInputStream( new
FileInputStream(target.getLocation().toFile())));

// mezclar origen y destino
merger.merge();
// extraer contenidos mezclados
InputStream mergedContents = new
ByteArrayInputStream(merger.getTargetCompilationUnit().getContents().getBytes());
// sobreescribir el destino con el contenido mezclado
```

```
target.setContents(mergedContents, true, false, monitor);
// ...
```

El código utilizado puede encontrarse en la clase **FileUtil**. Es invocado cada vez que se pide escribir una clase Java. El objeto "merger" es configurado a través de un archivo "merge.xml". Este especifica de qué manera se sobreescribirán o no métodos en la clase destino. La configuración usada en el plugin fue tomada de la generación de EMF, por lo tanto el mecanismo de merge es el mismo que usa EMF. Se puede explicar con un ejemplo:

```
/**
 * comentario del método
 * @generated
*/
public void testSimpleGetName() {
 // debido a el tag @generated,
 // cualquier código en este método será sobreescrito
}
 /**
 * test case for getName
 */
public void testSimpleSetName() {
 // el código en este método no será sobreescrito, porque se ha
 // eliminado el tag @generated.
 // La eliminación, o el reemplazo por "@generated NOT", evita la sobreescritura
}
```

7.3.3 PDE de Eclipse

El **ambiente de desarrollo de plugins** (PDE) provee un conjunto muy rico de herramientas para crear, desarrollar, probar, depurar e instalar plugins de Eclipse, fragmentos, características, sitios de actualización y productos RCP.

La plataforma Eclipse está estructurada como un **runtime central** y un conjunto de características adicionales que son instaladas como **plug-ins**. Los Plugins aportan nueva funcionalidad a la plataforma contribuyendo en ciertos "puntos de extensión" predefinidos.

La interfaz gráfica del workbench es contribuida por un plugin. Cuando se inicia el workbench, no se inicia un único programa Java. Se activa el runtime de la plataforma, que dinámicamente puede **descubrir los plugins registrados** e iniciarlos por demanda.

Cuando se desea proveer código que extiende la plataforma, se definen extensiones de sistema en el nuevo plugin. La plataforma contiene un conjunto predefinido de **puntos de extensión** (lugares donde uno se puede "enganchar" en la plataforma y aportar comportamiento). Desde el punto de vista de la plataforma, el nuevo plugin no es distinto a los plugins fundamentales como el manejo de recursos o el workbench.

Por lo tanto, para transformar cierto código en un plugin se debe:

- Decidir como va a ser integrado con la plataforma
- Identificar los puntos de extensión a los que se debe contribuir para integrar el plugin

- Implementar esas extensiones de acuerdo a la documentación de dichos puntos de extensión.
- Proveer un archivo de manifiesto ("manifest.mf") que describe el empaquetado y los prerequisitos del plugin. También se debe proveer un manifiesto de plugin ("plugin.xml") que describe las extensiones que se están definiendo.

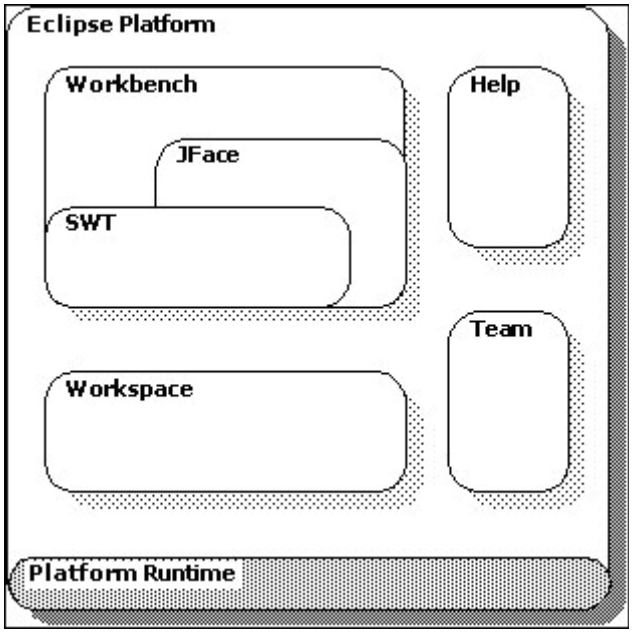


Ilustración 6-15. Arquitectura de la plataforma Eclipse (Fuente: ayuda de Eclipse)

El runtime de la plataforma está implementado usando el modelo de servicios de **OSGi**. Aún cuando los detalles de implementación pueden no ser importantes para muchos desarrolladores, los que estén familiarizados con OSGi reconocerán que un plugin de Eclipse es, en efecto, un "Bundle" OSGi.

El runtime define los plugins (org.eclipse.osgi y org.eclipse.core.runtime) en los cuales todos los demas plugins se apoyan. Es responsable de definir la estructura de los plugins y los detalles de implementación (bundles y classloaders) detrás de ellos. También es responsable de encontrar y ejecutar la aplicación principal y mantener un registro de plugins, sus extensiones y puntos de extensión.

En el caso del **plugin definido en este trabajo**, se utiliza bastante el plugin de recursos. La clase "GeneradorProyectos" lo utiliza para la creación de proyectos, y la copia de archivos. Pueden existir tres tipos de recursos: proyectos, carpetas y archivos. Eclipse define un workspace (espacio de trabajo), que contiene proyectos, que a su vez contiene carpetas y archivos.

7.4 Secuencia de pasos para la generación

Para esta tarea se uso EMF y JET. En un primer paso se realiza la lectura del modelo. A partir de los elementos del modelo, se genera un estructura intermedia en memoria. En un segundo paso, se genera código a partir de las estructuras intermedias.

Leer el modelo

El primer paso, la lectura del modelo y generación de estructuras intermedias, se realiza en la clase *"EstructurasIntermedias.java"*. Pero antes, se debe leer el objeto "cm.Model" desde el archivo. Este código EMF hace ese trabajo:

```
ResourceSet resourceSet = new ResourceSetImpl();
resourceSet.getPackageRegistry().put(CmPackage.eNS_URI, CmPackage.eINSTANCE);
resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put("cm",   new
XMIResourceFactoryImpl());

Resource resource = resourceSet.getResource(fileURI, true);
Model modelo = (Model) resource.getContents().get(0);
```

Generar objetos intermedios

El método "generarEstructurasIntermedias" (*"EstructurasIntermedias.java"*) es el encargado de leer el modelo y generar una colección de objetos intermedios. Se trata de recorrer todos los elementos del modelo, y por cada uno de ellos se realizará una operación. Sólo por la influencia de Mofscript las operaciones son llamadas Reglas. Dependiendo del tipo del elemento que se está recorriendo, se elegirá la regla a ejecutar.

Se realizan dos pasadas sobre el modelo. En la primera, se procesan las entidades. En la segunda, las asociaciones. Esto es para evitar que, al leer una asociación, esta trabaje sobre entidades que aún no fueron leídas. Es por esta razón que existen dos colecciones de reglas: "mapFirstPass" y "mapSecondPass". En la primera pasada, por cada elemento del modelo, se busca dentro de las reglas de *"mapFirstPass"* en busca de reglas compatibles con el elemento procesado. Si la regla es aplicable al elemento, se ejecuta.

El lector atento habrá notado que esa estructura intermedia se parece a un modelo intermedio. Y recordará que MDA propone llegar a la generación de código mediante el refinamiento de modelos, realizando sucesivas transformaciones de modelo a modelo. Pero a pesar de esta similitud, esto es algo diferente. Esta estructura intermedia es simplemente una traducción y no pide recibir nuevos parámetros específicos de plataforma, como sucede en MDA. Por esa razón **no** se realiza una transformación de modelo a modelo convencional, que requeriría definir un nuevo metamodelo en MOF y se escribir una transformación en un lenguaje de transformaciones M2M.

Usar las plantillas

Una vez creados estos objetos, se procede a la generación. Se comienza con las entidades (Sesiones, espacios de trabajo, objetos compartidos, herramientas, usuarios, roles de colaboración). Luego se generan los servicios por cada sesión. Terminado esto se continúa con otras clases laterales y la copia de las clases del framework, que no varían según el modelo.

Ya dijimos que el código JET tiene dos etapas. La plantilla escrita debe ser traducida a un programa Java. Luego el plugin utiliza la clase generada a partir de la plantilla, y pasándole por parámetro un objeto obtiene el texto que se escribirá. El modo de determinar, por cada clase objeto de las estructuras intermedias, qué plantilla se debe utilizar, es muy simple. Todas las subclases de JavaClass implementan el método "generateFile", que devuelve un String con el texto de la clase generada. Este método es quien se encarga, por ejemplo, de usar una plantilla distinta para "SharedObject" y "Session".
Por ejemplo, en *CollaborationRoleJavaClass*:

```
public String generateFile() {
        CollaborationRoleTemplate temp = new CollaborationRoleTemplate();
        return temp.generate(this);
    }
```


Illustration 1: Clases intermedias en la generación

Ilustración 16: Diagrama de secuencia. Generación de código.

7.4.1 Ejemplo para un artefacto

Para la generación de texto con JET se deben seguir los siguientes pasos:

1. Preparar el proyecto para trabajar con JET. Existe un asistente para convertir el proyecto a un proyecto JET. Se especifica la carpeta de plantillas.

2. Crear el archivo de la plantilla. Por ejemplo, "CollaborationRoleEntityGenerator.javajet" contiene este texto:

```
<%@ jet package="grp.jet" class="CollaborationRoleTemplate"
   imports="grp.genmodel.*
         grp.util.*
         " %>
<%
CollaborationRoleJavaClass clazz = (CollaborationRoleJavaClass) argument;
%>
<%=new PrefijoClaseTemplate().generate(clazz)%>
import gpfw.model.ICollaborationRole;
import gen.model.factory.*;

public class <%= clazz.getName() %>
   extends <%= clazz.getSuperClassFQN() %>  implements java.io.Serializable,ICollaborationRole {

   /**
   * @generated
   */
   public <%= clazz.getName() %>() {
   }

   <%=    new StandardFeaturesTemplate().generate(clazz)    %>

   /**
   * @generated
   */
   public Integer getId() {
      return CollaborationRoleFactory.<%= StringUtils.constNotation (clazz.getName()) %>;
   }

   /**
   * @generated
   */
   public String getNombre() {
      return "<%=StringUtils.escapeStringLiteral(clazz.getDescripcion())%>";
   }
}
```

De manera similar a JSP, el comienzo "**<%@**" indica el comienzo de una directiva. Las secuencias que comienzan con "**<%=**" encierran una expresión Java, cuyo resultado será añadido al texto de salida. Las sentencias Java encerradas entre "**<%**" y "**%>**" son ejecutadas, pero su resultado no es añadido al texto de salida.

La directiva "jet" define el nombre de la clase Java que se creará al guardar la plantilla. En este caso es "CollaborationRoleTemplate". Esta es la traducción de la plantilla, llamada *clase de implementación de la plantilla*. Define el método "generate", que recibe un objeto y retorna un texto. Ese método es el que se llama

para generar el texto de la plantilla. En el ejemplo se muestra como la plantilla llama a otra plantilla, con la expresión

```
"<%=    new StandardFeaturesTemplate().generate(clazz)    %>"
```

Esta segunda plantilla nos evita repetir el código que genera las propiedades de la clase por cada plantilla que escribamos.

- Por último, se llama al método *"generate"* de la traducción, y se escriben los resultados a la clase destino. En el caso de estudio existe el método *"FileUtil.generarFile",* que escribe los contenidos generados a disco, previo paso por JMerge.

El código generado puede verse así:

```
package gen.model;

import gpfw.model.ICollaborationRole;
import gen.model.factory.*;

public class UsuarioComunChat
    extends Object  implements java.io.Serializable,ICollaborationRole {

  /**
  * @generated
  */
  public UsuarioComunChat() {
  }


  /**
  * @generated
  */
  public Integer getId() {
    return CollaborationRoleFactory.USUARIO_COMUN_CHAT;
  }

  /**
  * @generated
  */
  public String getNombre() {
    return "UsuarioComunChat";
  }
}
```

7.5 Resumen

A lo largo de este capítulo se estudió la implementación del plugin de eclipse implementado en este trabajo.

Se comenzó con una visión general de las tecnologías utilizadas, haciendo una breve reseña de cada una y de cómo fueron utilizadas y con qué objetivo. Se

describió como se aportó la funcionalidad a la interfaz de Eclipse, como fue leido y procesado el modelo, como se realizó la generación y como se escribió lo generado a disco, cuidando de no sobreescribir el código modificado por el usuario.

También se cubrieron cuestiones formales de la implementación, que documentan el código y en algunos casos permiten entender algunas decisiones de diseño. Se describió la estructura del plugin y los proyectos que lo componen. Las principales clases y sus funciones, como también las principales plantillas, fueron explicadas de manera breve.

Luego se describió la secuencia de pasos que realiza el plugin para generar código. La lectura usando EMF, el código que genera una estructura intermedia en memoria, el uso de las plantillas JET para generar el texto, la utilización de Jmerge para evitar sobreescribir cambios realizados a código generado anteriormente; todos esos temas son tratados, y en algunos casos se plantean alternativas y se justifica la decisión tomada. Por ejemplo, se considera el uso de transformaciones modelo a modelo y luego se lo descarta.

Finalmente se ejemplificó la generación para dos tipos de artefactos: las operaciones, generadas mediante Mofscript, y los demás artefactos, utilizando JET. En cada caso de analizó la conveniencia de usar esas herramientas, encontrando pros y contras. Especialmente para Mofscript, donde se encuentran algunos problemas en su utilización.

En fin, el capítulo está destinado a que se comprendan las tareas de investigación y codificación que fueron necesarias en el transcurso del trabajo.

8 MODELADO DE SISTEMAS COLABORATIVOS

8.1Introduccion

La construcción de sistemas colaborativos es una tarea compleja dado que involucra varios actores intensamente interactivos dispersos en diferentes ubicaciones. La adopción de un proceso que guíe la construcción de dichos sistemas es crucial porque abarca prácticas repetitivas y técnicas que organizan el desarrollo de software y favorecen la calidad de software.

En el campo de la ingeniería de software, el MDD ("Desarrollo Dirigido por Modelos") emergió como un cambio de paradigma desde el desarrollo centrado en el código hacia el desarrollo basado en modelos. Este enfoque promueve la sistematización de la construcción de artefactos de software.

Para aplicar MDD al desarrollo necesitamos crear modelos abstractos. Podríamos hacerlo en UML, pero existe un salto muy grande entre la semántica de UML y la del dominio de los sistemas colaborativos. Esto justifica la creación del lenguaje específico de dominio para los sistemas colaborativos, llamado CSSL.

CSSL no es el primer DSL utilizable para el dominio. Existen:

1. Un lenguaje para modelar sistemas distribuidos y de hipermedia, [14]
2. UML-G: una notación para modelar información compartida en sistemas colaborativos. [15]
3. Un lenguaje para especificar sistemas distribuidos [16]
4. Una extensión a UML para aplicaciones interactivas. [17]

Ninguno abarca el dominio de manera exhaustiva. Además, ninguno está pensado para aplicar MDD para el desarrollo de un sistema groupware.

Los anteriormente mencionados proveen mecanismos para especificar algunos elementos existentes en sistemas colaborativos, pero la mayoría puede usarse en otro tipo de sistemas. Es decir, no están enfocados hacia los sistemas colaborativos. La excepción es UML-G, y es por eso que haré una breve reseña de [15].

8.2 UML-G

Antes de describir CSSL voy a describir brevemente UML-G, ya que ambos comparten algunos objetivos.

UML-G es una extensión a UML. La extensión a UML se hizo a través de un "UML Profile" (perfil de UML).

Intenta identificar las necesidades específicas del modelado de datos compartidos. Luego de identificar las necesidades las satisface usando las herramientas provistas por el mecanismo de extensión de UML (Estereotipos, etiquetas, restricciones..).

- Se necesita marcar los objetos como compartidos. Estereotipo <<shared>>.
- Se necesita marcar el objeto compartido como persistente, o persistente sólo por un período de tiempo. Etiquetas: "persistence", "time-persistence".
- A veces un objeto debe notificar sus cambios a los usuarios del sistema. Etiqueta: "observable"
- A veces es necesario tener un control de acceso sobre un objeto compartido. Etiqueta: "access-controlable".
- Los objetos compartidos pueden ser bloqueables. Etiqueta: "lockable".
- Se puede especificar la ubicación de los objetos compartidos (distribuidos, centralizados, ..). Etiqueta: "distribution={central, asymmetric, semi-replicated, replicated}
- Los actores de un sistema colaborativo necesitan ser modelados, porque también representan información que otros usuarios podrían necesitar. Estereotipos: <<SharedRole>>, <<sharedActor>>.
- Las actividades deben tener un elemento para modelarlas. Estereotipo: <<sharedActivity>>

Esto permite hacer cambios sobre la generación de código estándar de UML de algunas herramientas. Por ejemplo, se puede hacer que los <<sharedObjects>> sean subclases de una clase de un framework, o que contengan algunas propiedades según sus etiquetas.

El objetivo de este trabajo es, a través del lenguaje CSSL, mejorar la automatización de la generación de código, permitiendo generar artefactos más complejos basándose en modelos especificados en CSSL.

Por esa razón, la siguiente sección describe CSSL. Está basada en el trabajo *"A Domain Specific Language for the Development of Collaborative Systems"* [13], y se comienza a analizar las posibilidades de generación de código para algunos de sus elementos.

8.3 CSSL

CSSL ("Collaborative Software Systems Language") es un lenguaje bien definido utilizado para modelar sistemas colaborativos. Al ser un lenguaje gráfico, el mecanismo para la definición del lenguaje es el de metamodelado. El lenguaje utilizado para definir un lenguaje es llamado metalenguaje. Dentro de la especificación de la OMG, ese lenguaje (el modelo del metamodelo) es MOF ("Meta Object Facility"). La implementación de Eclipse de MOF es EMF. CSSL está definido utilizando EMF.

UML también está definido utilizando MOF. La última versión de CSSL utiliza algunas de las metaclases de UML y las especializa. Pero sólo un pequeño subconjunto, y sin utilizar UML2, que es la implementación más conocida de UML 2.0 usando EMF. El paper en el que se basa esta sección ([13]) está desactualizado en este aspecto.

Por esa razón, la generación de código de CSSL se puede realizar dentro de Eclipse sin necesidad de instalar el plugin de UML2. El código sólo necesita leer el

archivo con extensión cm en formato XMI generado por el editor gráfico, o modificado manualmente por alguien que sepa cómo hacerlo .

El lenguaje está organizado en tres paquetes:

- **Kernel**: contiene los conceptos y abstracciones base para los demás paquetes
- **Workspaces**: contiene los elementos para modelar las características estructurales de un sistema groupware.
- **Protocols**: contiene los elementos para describir el comportamiento del sistema colaborativo

8.3.1 Kernel.

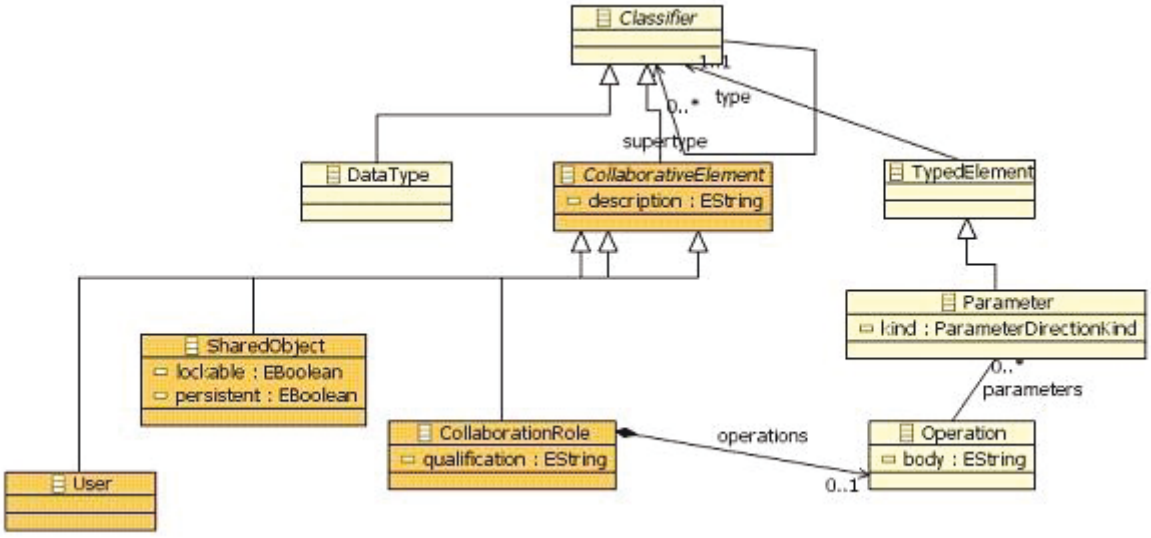


Ilustración 7-17 . Paquete Kernel

Vemos en el gráfico que se extiende Classifier con los siguientes elementos:

- *Collaboration Role:* rol que puede tener un usuario en el sistema. El modelo está obligado a describir sus responsabilidades y capacidades. Describir sus responsabilidades significa declarar un conjunto de operaciones.
- *User:* la clase que describe a los usuarios
- *Tool:* herramientas usadas para comunicarse con las sesiones colaborativas
- *SharedObject:* cualquier objeto accesible por varios usuarios en forma concurrente

8.3.1.1 Consideraciones de generación:

Desde la metaclase "Collaboration Role" se pueden crear en el modelo elementos tales como:

- Rol Moderador.
- Rol Participante Común

A partir del rol Moderador se creará una clase, llamada por ejemplo "RolModerador". La clase generada será un singleton. Es decir, no tiene sentido que existan dos instancias creadas a partir de la clase. Tendría sentido si una instancia de RolModerador referenciada desde un usuario tuviera datos distintos a los datos de otra instancia. Pero eso no sucede. Tal como se explicó en el capitulo de metamodelado, en los lenguajes específicos de dominio, el hecho de achicar el dominio y concentrarse en él nos permite agregar datos a la generación de código. Esto en una generación abstracta no lo podríamos asumir, ya que no sabríamos como se comporta la clase.

La metaclase *User*, por el contrario, es utilizada para definir múltiples instancias. Por ejemplo, en el modelo puede existir el elemento:

- UsuarioAplicacion, de tipo User

La generación puede crear una clase "UsuarioAplicacion". Conociendo el dominio podemos saber que debe tener:

1. *una lista de roles*
2. *un nombre de usuario y credenciales*
3. *puede tener un conjunto de objetos compartidos*. Estos podrían ser el perfil que el usuario hace visible a los demás e información de awareness (ej: estado de conexión).

Se creará, por cada usuario, un objeto contenedor, donde se agregarán las propiedades correspondientes a los objetos compartidos. Ante cambios en estos datos se puede desear informar a los demás actores de la colaboración.

Los objetos que se pondrán dentro de ese contenedor se deducirán por la asociación <*"shared"*> entre un usuario y un objeto compartido.

La generación de la metaclase Tool se explicará en el siguiente paquete de CSSL.

En cuanto a "SharedObject", se generará una clase de la misma manera que cualquier generador de código a partir de UML lo haría. La única diferencia es que se forzará que sea serializable.

8.3.2 Workspaces.

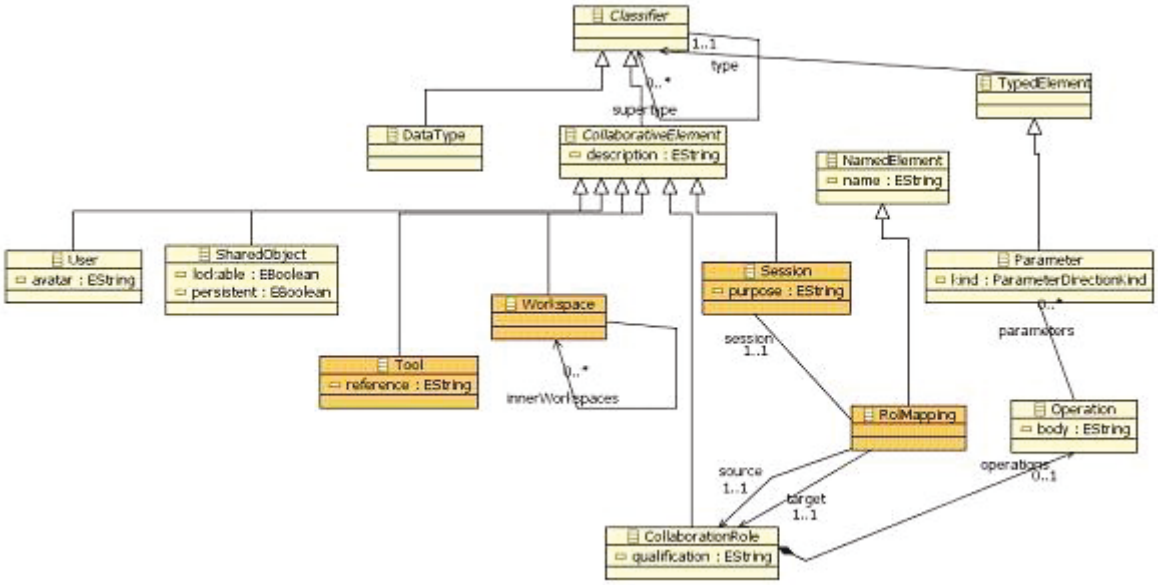


Ilustración 7-18 . Paquete Workspaces

Es usado para modelar el ambiente donde la colaboración se lleva a cabo.

Vemos que se agregan los siguientes elementos:

- *Session:* es un período de tiempo limitado en el cual se realiza un intercambio de información (o se modifica un objeto compartido) en busca de un objetivo común.
- *Workspace:* representa el lugar físico donde sucede la interacción, y donde están los objetos compartidos y herramientas. Las sesiones ocurren dentro de estos espacios de trabajo.

Con estos datos que nos provee el dominio, la generación de código puede darnos un valor agregado importante. Con sólo definir una sesión, tendremos creada la clase que la representa. Esa clase sabrá cual es el espacio de trabajo al cual pertenece.

Sabemos que la sesión tiene un conjunto de usuarios, y además sabemos que cada usuario interactúa con los demás con un rol determinado. Y ese rol puede cambiar según la sesión. Un mismo usuario puede tomar un rol de moderador en una discusión, y de participante en otra. Ese rol se suele determinar al comenzar la sesión, y claramente no se puede deducir conociendo los roles posibles de un usuario (sólo se puede restringir, es decir, podría haber usuarios que no pueden ser moderadores).

Además sabemos que las interacción en un sistema colaborativo se realiza a través de sesiones. También sabemos que los usuarios interactúan sobre la sesión a través de herramientas. Y CSSL nos permite especificar cuales son la herramientas permitidas para un tipo de sesión. Por ejemplo: la sesión de chat puede ser accedida a través de la herramienta "Ventana de Chat".

8.4 Modelo espacial

Este enfoque de modelado de los sistemas colaborativos nos permitirá pensar los estilos de colaboración y sus especificidades en términos de espacios. Es decir, traduciendo el conjunto de soluciones específicas a diseños particulares de espacios virtuales, sus vínculos mutuos, de la ubicación de los objetos, de la dinámica de la colaboración (protocolo de colaboración) y como van a interactuar los usuarios entre si y con los objetos, de la forma en que los usuarios ingresan, permanecen y salen del espacio.

En el mundo real la gente cambia continuamente entre diferentes estilos de colaboración en diferentes tiempos, diferentes lugares, formales e informales, en ambientes restrictivos o permisivos etc. En el caso de los sistemas groupware, muchas tecnologías fueron concebidas para mantener una situación colaborativa específica... En consecuencia, cuando una persona requiere cambiar de estilo de colaboración, tiene que cambiar de aplicación.

Trabajar con un modelo espacial, puede alimentar una amplia gama de estilos de colaboración diferentes en el mismo sistema colaborativo, por ejemplo podemos tener espacios en los que los usuarios pueden colaborar sincrónicamente y en otras pueden hacerlo en distinto tiempo. Usar una metáfora de ambiente, "room methaphor", ayuda también a mitigar o remover algunos gaps técnicos y ayuda al usuario a entender conceptualmente la aplicación. Asimismo facilita la transición (puede ser organizada con links entre los diferentes ambientes) entre los diferentes estilos de colaboración.

Particularmente, la metáfora de ambiente modela fácilmente qué es lo que la gente puede hacer en ese ambiente y es una forma natural de proveer oportunidades colaborativas. Los ambientes virtuales, análogamente a los ambientes físicos, pueden ser usados por los grupos dentro de la organización para realizar las actividades conjuntas.

8.5 Implementación de CSSL

8.5.1 Objetos Compartidos, Sesiones y Espacios

Uno de los requisitos esenciales para que una aplicación sea considerada colaborativa es que existan objetos compartidos. Un segundo requisito es que los usuarios que comparten un objeto estén ubicados en sitios diferentes.

Los asuntos a tratar son:

- **Donde ubicar** (lógicamente) los objetos compartidos.
- **Cómo acceder** y modificar los objetos compartidos

8.5.1.1 Ubicación lógica (Abstracción de programación)

Como se puede apreciar en el diagrama, tanto las sesiones como los espacios pueden contener objetos compartidos. Una sesión de chat pueden tener un log de chat, que sería simplemente una lista de mensajes. La lista de mensajes deja de tener utilidad cuando se termina la sesión. En otro caso el objeto compartido puede estar en el espacio de trabajo (por ejemplo un documento que se modifique por varios grupos de trabajo en sesiones de trabajo distintas).

Se puede trazar una analogía a la arquitectura de Servlets Java, donde los objetos correspondientes a la interacción entre un usuario y el servidor pueden estar en el pedido, la sesión, o la aplicación. Y en donde cada uno de ellos, respectivamente, es más amplio que el anterior. Y donde el más amplio (aplicación en un caso, Workspace en el otro) puede tener datos accedidos desde varias sesiones.

8.5.1.2 Ubicación física

El estado compartido debe ser mantenido de manera consistente, y para eso existen varias alternativas. En cuanto a la ubicación física de los objetos sabemos que puede ser:

- **Centralizada**, donde un servidor central contiene todos los objetos
- **Distribuida,** donde los objetos están replicados en todos los sitios

En un escenario distribuido, para hacer efectiva y consistente la distribución de los datos *sin un servidor que coordine las modificaciones*, se pueden utilizar algoritmos de Transformaciones Operacionales**[21] [22].** Básicamente se trata de encapsular las operaciones y enviarlas a cada uno de los otros sitios. La complejidad en un sistema distribuido radica en saber qué evento sucedió antes y cual después, y poder reordenar las operaciones para que el orden de ejecución sea el mismo en todos los nodos. Esto está explicado en el capítulo dedicado a los sistemas colaborativos, en la sección de control de concurrencia**.**

En un modelo de hub-and-spoke, donde un servidor recibe todas las operaciones y las distribuye, la idea de encapsular las operaciones utilizando el patrón Command ([29]) puede ser utilizada.

Es por eso que CSSL permite modelar operaciones (como las responsabilidades de un rol, por ejemplo). Y desde esas operaciones se modificarán los objetos compartidos.

8.5.1.3 Código generado para objetos Compartidos, Sesiones y Espacios

Esta primera implementación de la generación de código utiliza el modelo hub-and-spoke. Se encapsulan operaciones, que los clientes envían al servidor. Dichas operaciones modifican los objetos compartidos en el servidor y si no hubo ninguna excepción se hace un *broadcast* de la operación a todos los usuarios de la sesión.

Por cada sesión se genera un servicio. Por ejemplo, de un tipo de sesión "ChatSession" se generaría "ChatSessionService". El servicio es capaz de resolver al menos tres operaciones:

- *Creación (y registración) de la sesión*: simplemente ingresa la sesión a un registro, según su identificador único.
- *Bajar el estado actual de la sesión:* recibe el identificador de la sesión y trae el objeto hacia el cliente.
- *Ejecutar una operación sobre la sesión:* prepara, ejecuta y distribuye la operación

El proceso completo es el siguiente. El objeto operación es creado por código escrito manualmente. El usuario crea una operación y la envía a través de una herramienta. La herramienta conoce su sesión.

El Framework, automáticamente:

- Almacena el identificador de sesión y de herramienta en la operación. (Cliente)
- Envía la operación al servidor a través del *invoker* del servicio correspondiente al tipo de sesión. (Cliente)
- Recibe la operación. El servidor, ayudado con los identificadores recibidos y el registro propio, completa las propiedades SessionAbstract y ToolAbstract contenidas en el objeto operación. (Servidor)
- Llama a operacion.ejecutar()
- Si no hubo problemas (excepciones) en la operación anterior, hace un *broadcast* (difusión) de la operación a todos los usuarios de la sesión.
- Nuevamente en el cliente, se ejecuta la operación
- Se dispara un evento por la llegada de la operación. (Probablemente la ventana de chat esté registrada como un Observador de este evento, ver patrón Observer, [29]).

El código generado puede ser modificado. Existe la posibilidad, por ejemplo, de reducir el difusión de una operación a sólo un subgrupo de los usuarios.

8.6 Analizando la generación

Siguiendo las descripciones de características comunes a los DSL, y contrarias a ellos, del libro de Kelly y Tolvanen [3], encontramos algunas situaciones que no son comunes en el DSM.

- ***Generación de código parcial que necesita ser modificado***: CSSL necesita que se modifiquen regiones de código. El trabajo mencionado anteriormente no lo recomienda; sin embargo, el mismo libro propone tres maneras de especificar código relacionado con los modelos, en el ejemplo del capítulo 8 "Mobile Phone Applications Using a Python Framework".

    - **Código agregado a los elementos del modelo**: Es lo mejor cuando sólo unas líneas de código deben ser especificadas
    - **Código reusado desde una librería***:* En los elementos del modelo sólo se especifica el nombre de la función a llamar y sus parámetros.
    - **Regiones protegidas***:* son regiones del código generado marcadas para que, una vez modificadas por el programador, no sean sobreescritas por el generador.

CSSL usa regiones protegidas para permitir al programador modificar el código generado. Si bien está claro que lo mejor es la generación de absolutamente todo

el código, en este caso la distancia existente entre los conceptos del dominio representados y su implementación lo impiden.

- ***Usar "Round-tripping"***: está claro que el libro cuando habla de round-tripping se refiere a mantener la consistencia entre código y modelo en ambos sentidos. En el caso de CSSL, los cambios hechos en el código no se trasladan al modelo. Los cambios en el modelo se llevan al código, pero sin modificar las regiones protegidas.

8.6.1 Productividad y Calidad

Las principales razones para la adopción de DSM en las empresas son la ganancia en productividad y calidad.

El mayor nivel de abstracción ya ha causado una vez un gran aumento de la productividad en el desarrollo de software, cuando se pasó del lenguaje ensamblador a los lenguajes de tercera generación (3GL). 3GL aumentó la productividad de los desarrolladores en un 450%. En cambio Java sólo nos permite ser 20% más productivos que BASIC.

Existen reportes en los que enfoques específicos al dominio resultan entre 300 y 1000% más productivos que lenguajes de modelado de propósito general.

Se puede decir que CSSL no cumple con todos los requisitos para producir ese aumento en la productividad porque:

- No permite la generación total del código
- No puede ser usado por expertos en el dominio que no sean programadores

Sin embargo, provee algunas de las ventajas:

- En la fase de *mantenimiento*, la adaptación a cambios en la plataforma o ambiente que afecten sólo al código generado sólo requieren cambiar el generador de código, y al regenerar desde los modelos la aplicación quedará adaptada. Lo mismo sucede con los errores que pudiera tener el generador.
- *Rapidez del desarrollo:* al automatizar algunos aspectos de la aplicación, el desarrollador no tendrá que preocuparse por ellos. Sólo le quedarán al desarrollador los aspectos no automatizables del modelo.

En cuanto a la *calidad del software*, en CSSL se pueden incluir reglas de corrección del dominio. Por ejemplo, la existencia de al menos un tipo de sesión, al menos una herramienta con la que se pueda acceder, roles que puedan participar en ella, etc. Al automatizar la relación entre el dominio y la solución, reducimos el riesgo de implementar de manera errónea.

Además, en relación a la prueba del software, DSM se ocupa de los errores al principio, en la especificación y no en la codificación. Garantiza que la aplicación seguirá una arquitectura, pues está forzada por el lenguaje y el generador. Los errores más típicos de la implementación desaparecen, al menos en el código que CSSL puede generar automáticamente. Existe un gran reuso de código, porque el generador se apoya fuertemente el el framework subyacente. Además, el modelo es una documentación *actualizada* del sistema.

8.6.2 Arquitectura del DSM. División del trabajo de automatización

El trabajo de automatización está dividido entre el lenguaje, CSSL, que provee la abstracción, el generador de código, un framework que será descripto a continuación, y la plataforma destino.

La plataforma destino es Java 6 SE, utilizando Spring como medio de comunicación entre clientes y servidor.

El código generado corresponde al modelo hub-and-spoke, donde los clientes envían operaciones a un servidor central, que a su vez se ocupa de comunicar la operación a los clientes que corresponda. Dicho código se apoya en un framework que intenta ocultar los detalles de Spring, y que provee conceptos genéricos que serán especializados por el código generado. Además, facilita la comunicación cliente-servidor y la utilización de objetos compartidos.

Los lenguajes de modelado que sólo permiten capturar información de diseño parcial no son aconsejables. El ejemplo clásico es generar el código de una clase a partir de una clase en diagrama UML. Sólo es un cambio en la representación, no un cambio en el nivel de abstracción [3]. CSSL cambia el nivel de abstracción. Los artefactos generados son más que la información representada en el modelo, agregan un comportamiento característico a los elementos del dominio. Por ejemplo, para una sesión se genera un servicio que permite ejecutar operaciones sobre sus objetos compatidos. Y, gracias al conocimiento del dominio que posee el generador, se generan algunas caracteristicas estructurales de una sesión que no necesitan ser especificadas. Por ejemplo: no hace falta especificar que una sesión tiene una lista de usuarios.

CSSL usa regiones protegidas para permitir modificar el código. En ocasiones sucede que las partes escritas manualmente en regiones protegidas pasan a ser incorrectas por la modificación del modelo. Sin embargo, no debería ser habitual. Por ejemplo las operaciones, que son objetos que no pueden ser definidos completamente por el lenguaje por ser específicos a la aplicación, se basan en pocos elementos del framework y el código generado. Eso reduce las probabilidades de conflicto con un cambio en el modelo.

8.7 Implementación de CSSL. Descripción del código generado

Una sección anterior describe la arquitectura que siguen los sistemas generados por CSSL, y como funcionan.

Esta sección se propone describir los artefactos generados, detallando qué hace cada uno, pero sin describir la interacción entre ellos.
Como se acostumbra en DSM, CSSL apoya en su generación en un framework de dominio. Ver figura (tomada desde [3]).

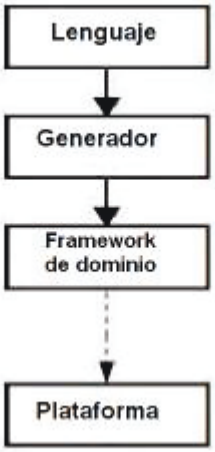


Ilustración 7-19 . Componentes de la generación

8.7.1.1 Framework

Se encarga de algunas tareas comunes al tipo de sistema:

- *Manejo de Usuarios del sistema*. Implementado en los paquetes "gpfw.remoting.user" y "gpfw.user".
- Clases que ayudan al código generado en sus funciones. Por ejemplo:
  - *Abstractas* desde donde extienden los Usuarios,Workspaces, Sesiones, Herramientas.Ubicadas en el paquete "gpfw.model".
  - *Soporte para operaciones sobre Sesiones*. Ubicado en el paquete "gpfw.remoting.session.ops".
  - *"gpfw.remoting.session.events.SessionEventManager"*. Se encarga de notificar a sus listeners las operaciones realizadas sobre una sesion.
- Utilidades para facilitar el *broadcast de operaciones*. Ubicadas en el paquete "gpfw.remoting.utils".
- Existen algunas *herramientas de GUI*, que en realidad se pueden considerar código de prueba, no hay necesidad de que esten. Paquete "gpfw.gui".

8.7.1.2 Código generado

Los siguientes artefactos son generados automáticamente:

- **Código de la clase por cada objeto graficado** (en el paquete "gen.model"). Por ejemplo: ChatSession (sesión), Message (objeto compartido), State (objeto compartido), ToolChatWindow (Herramienta).

Por cada sesión del modelo:

- **Operaciones genéricas** (Objetos operación, que extienden AbstractSessionOperation y contienen los ids de sesión y de Herramienta).

- **Herramientas**: una instancia de la clase herramienta pertenece a una unica sesión en un momento dado, y por lo tanto a partir de la herramienta se sabe a qué servicio de sesión llamar. La invocación de una operación se hace a través de la herramienta, y los identificadores de sesión y herramienta se llenan automáticamente al realizar la invocación.
- **Servicio para ejecutar operaciones**, dentro del paquete "gen.remoting.api".Las operaciones básicas son:
    - crear
    - ejecutar Operación
    - ingresar
- **Stubs de cliente (salida):** Stubs para ejecutar esas operaciones desde el cliente. Se generan con un nombre que cumple con el siguiente patrón: gen.client.remote.[nombreSesion]ImplOut
- **Stubs de cliente (entrada):** Stubs para recibir en el *cliente* esas operaciones desde la red. Se generan con un nombre que cumple con el siguiente patrón: "gen.client.remote.[nombreSesion]ImplIn". Este código realiza, por defecto, estas actividades:
    - Delega a un objeto ModelManager la ejecución (esto afectaría a los objetos del modelo)
    - Pide a SessionEventManager que levante un evento. (si el programador previamente había registrado ciertas ventanas para que reciban el evento de cada operación)
- **Stubs de servidor:** Stubs para recibir en el *servidor* esas operaciones desde la red. El código por defecto para esas operaciones en el servidor está en "gen.server.[NombreSesión]ServiceImpl", y realiza las siguientes acciones:
    - Ubica los OBJETOS Sesion y Herramienta segun los Ids que recibe desde la operación
    - Los inyecta en el objeto Operación
    - Detecta la ip origen del pedido y lo guarda en una variable ubicada en el almacenamiento local al hilo ("Thread Local Storage", abreviado TLS).
    - Delega la ejecución a ModelManager, una clase pensada para que sea sobreescrita por el programador, pero que por defecto solo llama a Operacion.execute()
    - Hace un broadcast de la operacion a TODOS los otros usuarios de la sesión. El programador puede modificar ese código para limitar la información a los que la necesitan.

8.8 Resumen

El tema principal de este capítulo fue el modelado de sistemas colaborativos. Se repasaron algunos lenguajes que tienen por objetivo realizar dicha tarea, UML-G y CSSL. Se explicaron los principales componentes de CSSL, sus paquetes “Kernel” y “Workspaces”.

Luego de este primer enfoque teórico, este capítulo prefigura el contenido de su sucesor, ya que comienza a analizar, todavía de forma vaga y general, la generación de código de CSSL. Comienza a ver los problemas y las características de la generación, y a contraponer las alternativas disponibles en la literatura del asunto con las decisiones tomadas en cada caso. Por ejemplo, se habla del aumento de productividad mediante la generación de código y se analiza si esto es real en el caso de CSSL. Se repasa la división de la automatización de la generación, y se describe como se estructura el framework de dominio en el caso

particular de este DSL. Se repasan las maneras de mezclar código generado y escrito a mano, y se explica cuál es la opción que se eligió para este generador.

9 UTILIZACIÓN DEL GENERADOR

Para ilustrar las ideas anteriores se mostrará el generador en acción usando una aplicación de ejemplo. Se presentarán los pasos necesarios para generar el esqueleto de la aplicación.

La aplicación que nos proponemos construir es un chat bastante sencillo. Las operaciones que se pueden realizar en una sesión de chat son enviar un mensaje, comentar un mensaje (es decir, realizar una marca sobre un mensaje anterior). Además, el usuario puede cambiar de estado y ese cambio debe ser notificado a los demás usuarios conectados.

9.1Creación del modelo de ejemplo

En primer lugar se debe generar el modelo a través del editor gráfico. Para eso:

- Crearemos un proyecto eclipse de nombre “AplicacionEjemplo”
- Crearemos la carpeta “model”
- Desde el menú contextual (click derecho), New -> Example -> Cm Diagram ->
- Se elige el nombre del modelo, en este caso dejaremos el valor por defecto (default.cm_diagram)
- Observamos que crea tanto el archivo "default.cm_diagram" como “default.cm”. El primero contiene la información gráfica del modelo, el segundo tan sólo la información del modelo.

El modelo de ejemplo es el siguiente.

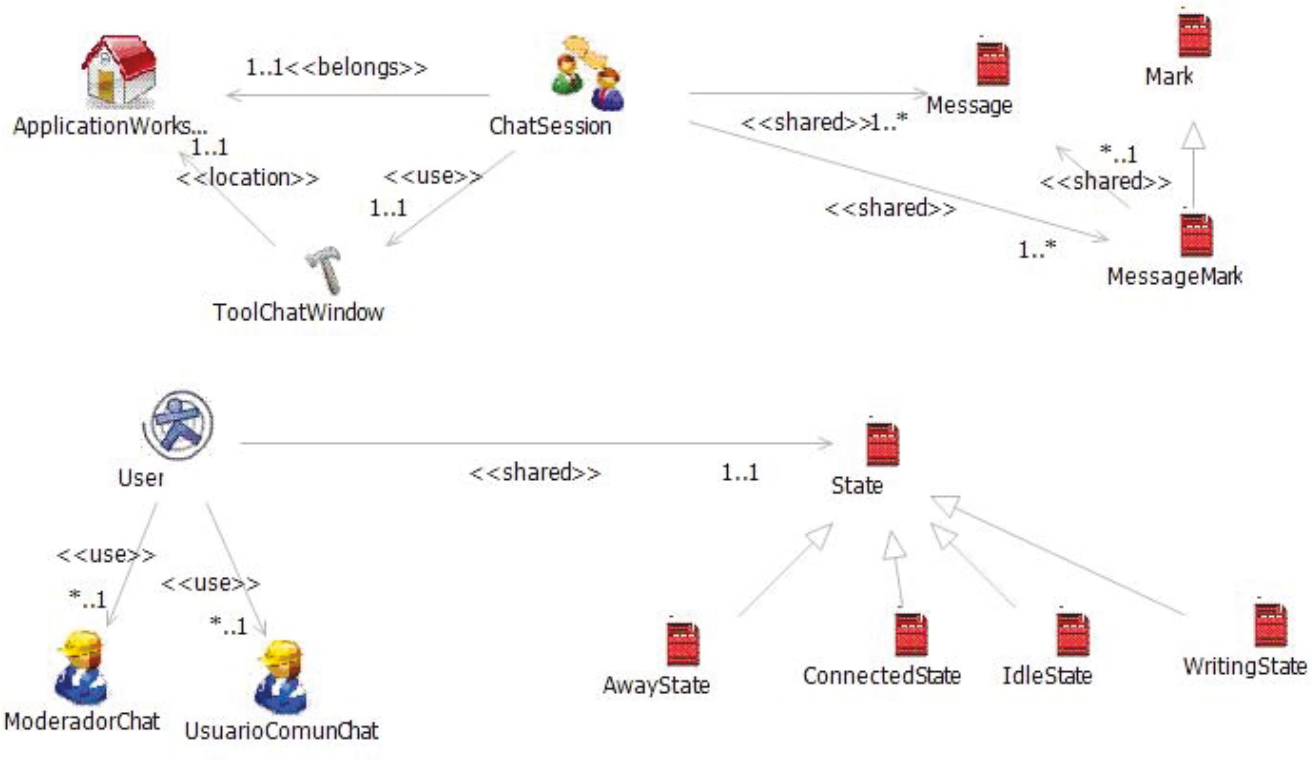


Ilustración 7-20 . Modelo CSSL de ejemplo

El diagrama, y por lo tanto la aplicación que se intenta modelar, está compuesta por los siguientes elementos:

- Un único workspace, llamado ApplicationWorkspace.
- Un único tipo de sesión, llamado ChatSession. La sesión obviamente, pertenece al único workspace.
- La herramienta ToolChatWindow es la utilizada para interactuar con la sesión. Esa herramienta está ubicada en el único workspace (ApplicationWorkspace).
- La sesión contiene los siguientes objetos compartidos:
  - Una lista de mensajes, (Message). Representa un mensaje en el chat.
  - Una lista de marcas (MessageMark). Las marcas son señaladores que comentan algo acerca de un mensaje.
  - El tipo que representa el usuario se llama "User", y puede tomar dos roles: "ModeradorChat" y "UsuarioComunChat". Además, el usuario conoce su estado (objeto "State", con sus subclases AwayState, ConnectedState, IdleState, WritingState), y este estado es distribuido a los demás usuarios. En otras palabras, es información de awareness.

9.2 Generación de los artefactos

El paso siguiente es hacer click derecho sobre "default.cm", elegir "CSSL -> Generar Código". Observamos que automáticamente se crean tres proyectos:

- **CSSLGenCommon:** contiene las clases usadas tanto por el servidor como por el cliente
    - *gen.model:* entidades
    - *gen.model.factory:* enumeradores e instancias únicas para algunas entidades
    - *gen.remoting.api:* interfaces de los servicios de sesión.
    - *gen.remoting.session:* utilidades para manejar las operaciones
    - *gpfw.*:* clases útiles del framework, no son generadas, no tienen relación con el modelo
    - *gpfw.model:* clases abstractas, desde donde extenderán las clases generadas
    - *gpfw.remoting.session.*:* útiles para el manejo de los servicios de sesión y sus operaciones
    - *gpfw.remoting.user:* útiles para el manejo de usuarios. Interfaz del servicio de usuarios.
- **CSSLGenClient:** contiene las clases usadas exclusivamente por el cliente
    - *gen.client.gui:* clases de ejemplo, donde se implementa una ventana y un observer de algunos eventos.
    - *gen.client.main:* inicio de la aplicación
    - *gen.client.remote:* implementación del servicio de la sesión en el cliente. Stubs para recibir y enviar operaciones.
- **CSSLGenServer:** contiene las clases usadas únicamente por el servidor
    - *gen.server:* implementación del servicio en el servidor, y broadcast de operaciones

9.2.1 Ejemplos de Artefactos generados

Comunes a cliente y Servidor

Ejemplo de entidad, la sesión "ChatSession", conteniendo los objetos compartidos "Message" y "Message Mark".

```
package gen.model;
import gpfw.model.ISharedObjects;
public class ChatSession extends gpfw.model.AbstractSession implements java.io.Serializable {
    public ChatSession() {
        setSharedObjects(new ChatSessionSharedObjects());
        setWorkspace(gen.model.ApplicationWorkspace.getInstance());
    }
    public static class ChatSessionSharedObjects extends ISharedObjects {
        public ChatSessionSharedObjects() {}
        /**
         * @generated
         */
        private java.util.List<gen.model.Message> messages = new java.util.ArrayList<gen.model.Message>();
        /**
         * @generated
         */
        public java.util.List<gen.model.Message> getMessages() {
```

```
                return this.messages;
        }
        /**
         * @generated
         */
        public void setMessages(java.util.List<gen.model.Message> arg) {
                this.messages = arg;
        }
        /**
         * @generated
         */
        private java.util.List<gen.model.MessageMark> messageMarks = new
        java.util.ArrayList<gen.model.MessageMark>();
        /**
         * @generated
         */
        public java.util.List<gen.model.MessageMark> getMessageMarks() {
                return this.messageMarks;
        }
        /**
         * @generated
         */
        public void setMessageMarks(java.util.List<gen.model.MessageMark> arg) {
                this.messageMarks = arg;
        }
    }
    /**
     * @generated
     */
    public int getClassId() {
        return 0;
    }
}
```

Cliente

La clase "ChatSessionImplOut", encargada de enviar el servidor los pedidos de ejecución de operaciones, creación de sesiones, unión a sesión, etc..

```
package gen.client.remote;
import gen.model.ChatSession;
import gen.remoting.api.ChatSessionService;
import gpfw.exception.GPFWException;
import gpfw.model.AbstractSession;
import gpfw.model.AbstractUser;
import gpfw.model.ICollaborationRole;
import gpfw.remoting.session.ops.IOperation;

public class ChatSessionImplOut implements gen.remoting.api.ChatSessionService {
        private static Logger logger = Logger.getLogger(ChatSessionImplOut.class);
        private ChatSessionService httpSessionInvokerProxy;
```

```
	/**
	 * @generated
	 */
	public ChatSession createNewSession(ChatSession session) {
		logger.debug("\ncreateNewSession (Salida del cliente al servidor - ChatSessionImplOut =>
				createNewSession)");
		return this.getHttpSessionInvokerProxy().createNewSession(session);
	}
	/**
	 * @generated
	 */
	public ChatSessionService getHttpSessionInvokerProxy() {
		return httpSessionInvokerProxy;
	}
	/**
	 * @generated
	 */
	public void setHttpSessionInvokerProxy(ChatSessionService httpSessionInvokerProxy) {
		this.httpSessionInvokerProxy = httpSessionInvokerProxy;
	}
	/**
	 * @generated
	 */
	public AbstractSession downloadState(String sessionId, AbstractUser user) {
		return this.getHttpSessionInvokerProxy().downloadState(sessionId, user);
	}
	/**
	 * @generated
	 */
	public void joinSession(String sessionId, AbstractUser user,ICollaborationRole rol) {
		this.getHttpSessionInvokerProxy().joinSession(sessionId, user, rol);
	}
	/**
	 * @generated
	 */
	public IOperation execute(IOperation operation) throws GPFWException {
		logger.debug("OUT. execute operation." + operation);
		return this.getHttpSessionInvokerProxy().execute(operation);
	}
}
```

Servidor

La clase "ChatSessionServiceImpl", implementación del servicio en el servidor, y broadcast de operaciones.

Se ven las operaciones:

- *createNewSession*: crea la sesión, la guarda en el registro de sesiones, y avisa a los participantes invitados.
- *joinSession*: permite a un usuario unirse a la sesión. Le avisa a los demás usuarios de este cambio en la sesión.
- *execute*: ejecuta una operación.
    - Obtiene la referencia a la sesión, según el sessionId
    - Delega a "ModelManager", quien a primero inyecta las variables "session" y "tool" y luego llama al método "execute" de la sesión.

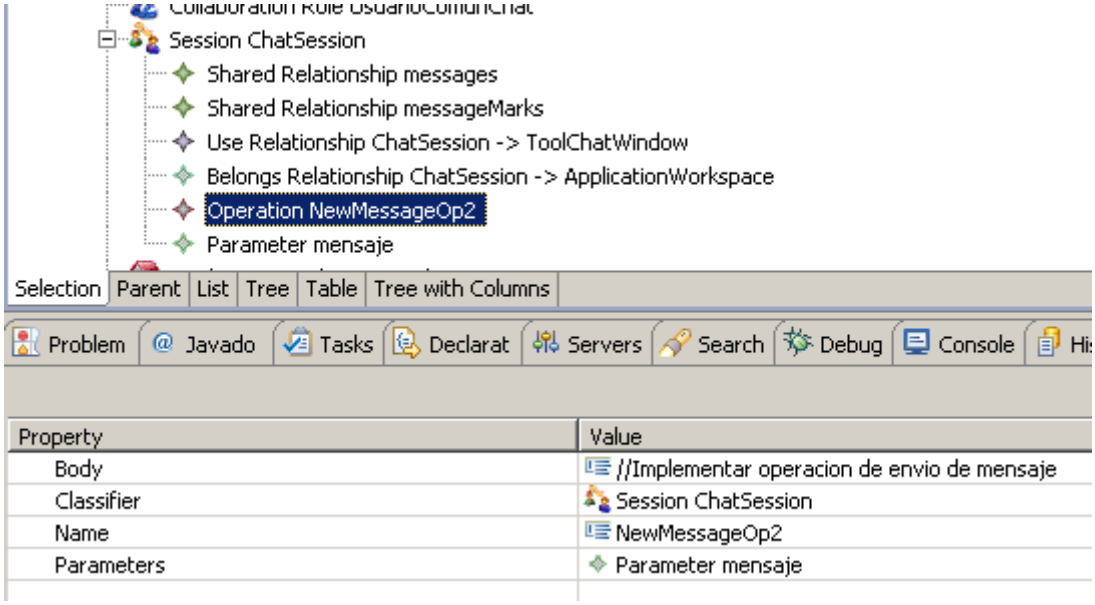
- Hace el broadcast de la operación a todos los miembros de la sesión. Si se desea reducir el grupo interesado en en esta modificación, por ejemplo por motivos de seguridad o performance, se puede modificar el método "broadcastExecute", y eliminar el "@generated" del javadoc.

```
package gen.server;
import gen.server.controller.ModelManager;
import gpfw.exception.GPFWException;
import gpfw.model.AbstractSession;
import gpfw.model.AbstractUser;
import gpfw.model.ICollaborationRole;
import gpfw.remoting.session.ops.AbstractSessionOperation;
import gpfw.remoting.session.ops.IOperation;
import gpfw.utils.SessionStorage;

public class ChatSessionServiceImpl implements gen.remoting.api.ChatSessionService {
	private static Logger logger = Logger.getLogger(ChatSessionServiceImpl.class);
	/**
	 * @generated
	 */
	public ChatSession createNewSession(ChatSession session) {
		String newId = SessionStorage.getInstance().generateSessionId();
		session.setSessionId(newId);
		SessionStorage.getInstance().getSesiones().add(session);
		broadcastCreateNewSession(session);
		return session;
	}
	/**
	 * @generated
	 */
	private void broadcastCreateNewSession(ChatSession session) {
		try {
			List<gen.remoting.api.ChatSessionService> lst = ChatSessionBroadcaster
					.getInstance().getInvokersAll(ChatSessionService.class);
			logger.info("broadcastCreateNewSession:" + lst);
			for (ChatSessionService srv : lst) {
				try {
					logger.debug("dbg" + srv);
					srv.createNewSession(session);
				} catch (RuntimeException e) {
					logger.error("broadcastCreateNewSession. error broadcasting",
e);
				}
			}
		} catch (Exception e) { logger.error("broadcastCreateNewSession", e);}
	}
	/**
	 * @generated
	 */
	private void broadcastJoinSession(AbstractSession session, AbstractUser user, ICollaborationRole rol) {
		try {
			List<gen.remoting.api.ChatSessionService> lst = ChatSessionBroadcaster
					.getInstance().getInvokersSession(session,
ChatSessionService.class);
			for (ChatSessionService srv : lst) {
```

```
                                        try {
                                                srv.joinSession(session.getSessionId(), user, rol);
                                        } catch (RuntimeException e) {
                                                logger.error("broadcastjoinSession. error broadcasting", e);
                                        }
                                }
                        } catch (Exception e) { logger.error("broadcastJoinSession. error broadcasting", e);}
        }
        /**
         * @generated
         */
        private void broadcastExecute(AbstractSession session, IOperation operation) {
                try {
                        List<gen.remoting.api.ChatSessionService> lst = ChatSessionBroadcaster
                                        .getInstance().getInvokersSession(session,ChatSessionService.cl
ass);
                        logger.debug("server. Comienzo Broadcast hacia: lst.size="+ lst.size());
                        logger.debug("server. la sesion es " + session.getSessionId() + ". Sus usuarios" +
session.getUsers());
                        for (gen.remoting.api.ChatSessionService srv : lst) {
                                try {
                                        logger.debug("server. broadcast.execute: " + srv);
                                        srv.execute(operation);
                                } catch (RuntimeException e) {
                                        logger.error("broadcastexecute. error broadcasting", e);
                                }
                        }
                        logger.debug("server. FIN Broadcast hacia: lst.size=" + lst.size());
                } catch (Exception e) { logger.error("Error broadcasting", e); }
        }
        /**
         * @generated
         */
        public AbstractSession downloadState(String sessionId, AbstractUser user) {
                // TODO: chequear permisos del usuario!
                return SessionStorage.getInstance().getSessionById(sessionId);
        }
        /**
         * @generated
         */
        public void joinSession(String sessionId, AbstractUser user, ICollaborationRole rol) {
                ChatSession session = (ChatSession) SessionStorage.getInstance()
                                .getSessionById(sessionId);
                ModelManager.getInstance().processJoinSession(session, user, rol);
                broadcastJoinSession(session, user, rol);
        }
        /**
         * @generated
         */
        public IOperation execute(IOperation operation) throws GPFWException {
                logger.debug("server.execute");
                ChatSession session = (ChatSession) SessionStorage.getInstance()
                                .getSessionById(((AbstractSessionOperation) operation).getSessionId());
                operation = ModelManager.getInstance().processOperation(operation);
                broadcastExecute(session, operation);
                return operation;
        }
}
```

Operaciones

A partir del siguiente segmento del modelo:



Se genera el esqueleto de la operación que envía el mensaje ("NewMessageOp2"). En la siguiente sección se verá como modificar la implementación (dentro del método "execute").

```
package gen.remoting.session.ops;

import gen.model.Message;
....
public class NewMessageOp2 extends AbstractSessionOperation implements IOperation {
	private static final long serialVersionUID = 1L;
	/**
	 * @generated
	 */
	public NewMessageOp2(gen.model.Message mensaje) {
		this.mensaje = mensaje;
	}
	// inicio Parametros
	/**
	 * @generated
	 */
	private gen.model.Message mensaje;
	/**
	 * @generated
	 * */
	public Message getMensaje() {
```

```
                return mensaje;
        }
        /**
         * @generated
         */
        public void setMensaje(gen.model.Message myVar) {
                this.mensaje = myVar;
        }
        // fin Parametros
        /**
         * @generated
         */
        public boolean execute() throws GPFWException {
                // TODO: implementar
                return true;
        }
}
```

9.3 Probando el código generado

Lo siguiente por hacer es comprobar que el código esté funcionando correctamente. Para eso, modificaremos la operación descripta en la sección anterior ("NewMessageOp2"):

```
package gen.remoting.session.ops;
..
import gen.model.ChatSession.ChatSessionSharedObjects;
..
public class NewMessageOp extends AbstractSessionOperation implements IOperation {
   private Message msg ;
   public NewMessageOp(Message message) {
      this.msg = message;
   }(…)
   /**
    * @generated NOT
   */
   public boolean execute() throws GPFWException {
      ((ChatSessionSharedObjects)getSession().getSharedObjects()).getMessages().add(getMsg());
      return false;
   }
}
```

Esta operación la enviaremos a través del servicio "ChatSessionService".

La operación simplemente toma la lista de mensajes de la sesión, y agrega el mensaje enviado. El método "getSession()" es una comodidad del framework, que nos permite encontrar la sesión entre los objetos de esa máquina. Nótese la eliminación o modificación de las anotaciones de generación, lo que impide que se sobreescriba en método execute con su contenido por defecto.

9.3.1 Código auxiliar para pruebas

Con el objetivo de probar el código generado, vamos a escribir cierto código que simule el comportamiento de dos usuarios. Uno de ellos creará una sesión, el otro esperará a que el primero lo haga y se unirá a ella. Se enviará un primer mensaje, y luego cada uno de ellos responderá al mensaje de su interlocutor.

Se lista, a modo de ejemplo, el código del segundo usuario:

```
package util.test;
…..
public class TestUtilUser2 implements SessionEventListener<ChatSession>{
    private static TestUtilUser2 instance = new TestUtilUser2();
    private ChatSession session;
    private ToolChatWindow tool;
    private ExecutorService threadPool;
   public static int counter = 0;
    private String comienzoMensajePropio = "Mensaje de Usuario 2. NUMERO:";
    private TestUtilUser2(){}
    private static final Logger logger = Logger.getLogger(TestUtilUser2.class);
    public static TestUtilUser2 getInstance(){
          return instance;
    }
    public boolean verSiHayOtroYUnirse (){
          ToolChatWindow ventanaChat = new ToolChatWindow();
          ChatSession sesion = null;
          ventanaChat.setSession(sesion);
          User user = (User) UserManager.getInstance().getUserLoggedIn();
          IWorkspace appSpace =
          this.getUserService().getWorkspaces(ApplicationWorkspace.getInstance().getId());
          ICollaborationRole rol =
          CollaborationRoleFactory.getById(CollaborationRoleFactory.USUARIO_COMUN_CHAT);
          if (appSpace.getSesiones().size() > 0){
                    AbstractSession sessPrx = appSpace.getSesiones().iterator().next();
                    AbstractSession nuevaSesion =
          getChatSessionService().downloadState(sessPrx.getSessionId(), user);
                    SessionStorage.getInstance().registrarSession(nuevaSesion);
                    getChatSessionService().joinSession(nuevaSesion.getSessionId(), user, rol);
                    session = (ChatSession) nuevaSesion;
                    SessionStorage.getInstance().registrarSession(nuevaSesion);
                    return true;
          }
          return false;
    }

    private ChatSessionService getChatSessionService() {
          return (ChatSessionService) SpringUtil.getApplicationContext().getBean("chatSessionImplOut");
    }

    public void loginUsuario2(){
          threadPool = Executors.newCachedThreadPool();
          SessionEventManager.getInstance().getListeners().add(this);
```

```
        State state = new ConnectedState();
        User user = new User ();
        user.setNick("SEGUNDO USUARIO");
        user.setState(state);
        try {
                Thread.sleep(2000);
                this.getUserService().login(user);
                UserManager.getInstance().setUserLoggedIn(user);
                boolean f = this.verSiHayOtroYUnirse();
                try{
                   logger.info("envio mensaje");
                   envioMensajeUsuario2();
                   logger.info("envio mensaje. fin");
                } catch (Throwable t){ logger.error("ERROR ENVIO", t); }
        } catch (UserServiceException e) {
                logger.info("el login no se pudo realizar:"+user);
                logger.debug(":", e);
        } catch (InterruptedException e) { logger.debug(":", e); }
    }

    public IUserService getUserService() {
        return (IUserService) SpringUtil.getApplicationContext().getBean("userOutImpl");
    }

    public void envioMensajeUsuario2(){
        this.tool = (ToolChatWindow) session.getToolByClassId(ToolFactory.TOOL_CHAT_WINDOW);
        AbstractUser user = UserManager.getInstance().getUserLoggedIn();
        Message msg = new Message();
        msg.setDate(new Date(System.currentTimeMillis()));
        synchronized (this){
                msg.setMessage(comienzoMensajePropio + counter++);
        }
        msg.setUser((User)user);
        NewMessageOp op = new NewMessageOp(msg);
        logger.info("envio mensaje de Usuario 2. execute");
        this.tool.execute(op);
        logger.info("envio mensaje de Usuario 2. execute ok");
    }

    public void onCreateNewSession(SessionEvent<ChatSession> e) {
        logger.debug("onCreateNewSession ignorado" + e.getSession().getSessionId());
    }

    public void onExecute(SessionEvent<ChatSession> e) throws GPFWException {
        AbstractSessionOperation opSession = ((AbstractSessionOperation)e.getOperation());
        if (!opSession.getSessionId().equals( this.session.getSessionId()))
                return;
        if (e.getOperation() instanceof NewMessageMarkOp){
                NewMessageMarkOp op = (NewMessageMarkOp) e.getOperation();
                logger.info("MOSTRAR en pantalla NewMessageMarkOp");
        }else if (e.getOperation() instanceof NewMessageOp) {
                NewMessageOp op = (NewMessageOp) e.getOperation();
                logger.info("MOSTRAR en pantalla NewMessageOp:" + op.getMsg().getMessage());
                //responder
                if (op.getMsg().getMessage().startsWith(comienzoMensajePropio )){
                        //responder
                        threadPool.execute(new Responder());
                }
        }
    }
```

```
    public static class Responder implements Runnable{
        public void run() {
           try {
                    Thread.sleep(5000);
           } catch (InterruptedException e1) { throw new RuntimeException(e1);  }
           TestUtilUser2.getInstance().envioMensajeUsuario2();
        }
    }
    public void onJoinSession(SessionEvent<ChatSession> e) { }
}
```

En "envioMensajeUsuario2" se muestra como se debe crear la operación, y el envío se realiza a través del método "execute" de la **herramienta**.
Desde el sistema del segundo usuario, para recibir las operaciones, lo primero que se debe ejecutar es:

```
SessionEventManager.getInstance().getListeners().add(this);
```

Esa es la registración de una clase como observer de las operaciones de sesión. El método "onExecute" atiende el evento de ejecución de una operación. En este caso, al recibir la operación se imprime en el log un mensaje, se esperan 5 segundos y se responde con otro mensaje.

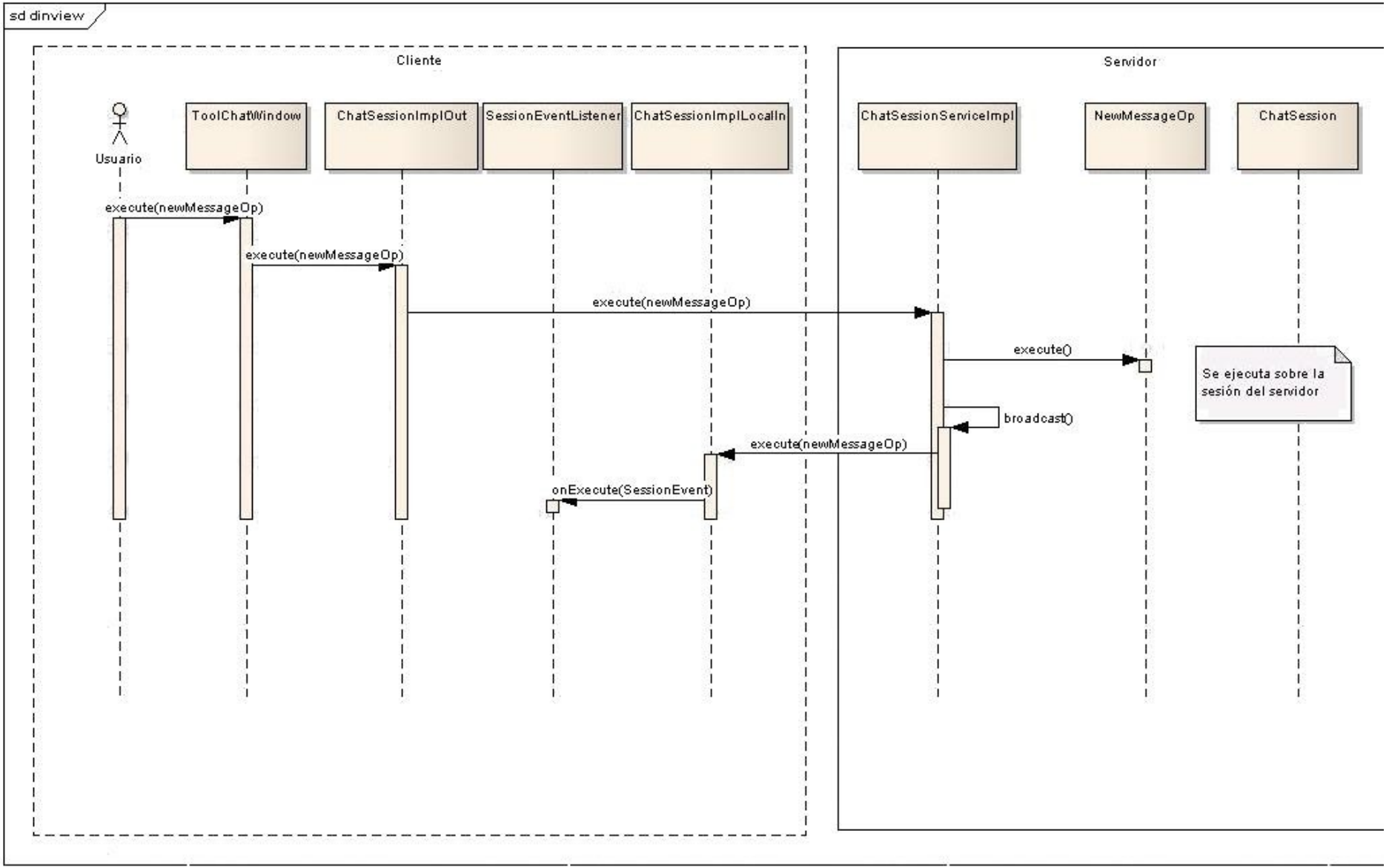
Comienzo de la aplicación: el inicio de la aplicación está en la clase ApplicationStart, método "run". Se debe reemplazar el código de esa método. En este ejemplo, con el siguiente código.

```
package gen.client.main;
import gpfw.utils.Configuracion;
import org.springframework.context.ApplicationContext;
import util.test.TestUtil;
import util.test.TestUtilUser2;

public class ApplicationStart implements Runnable, ApplicationContextAware {
        ApplicationContext context;
        public ApplicationStart() {
                Thread thread = new Thread(this);
                thread.start();
        }
        /**
         * @generated NOT
         */
        public void run() {
                if (Configuracion.getInstance().getClientPort() == 8080) {
                        TestUtil.getInstance().login();
                } else {
                        TestUtilUser2.getInstance().loginUsuario2();
                }
        }

        public void setApplicationContext(ApplicationContext arg0) throws BeansException {
```

```
                context = arg0;
        }
}
```

En negrita están resaltados los cambios hechos. El código que simula los el comportamiento de dos usuarios dentro del método "run". El tag de "@generated NOT" evita que se sobreescriba el método en una regeneración.

A continuación se grafica la interacción de los componentes:


sd dinview
Cliente
Servidor
Usuario
ToolChatWindow
ChatSessionImplOut
SessionEventListener
ChatSessionImplLocalIn
ChatSessionServiceImpl
NewMessageOp
ChatSession
execute(newMessageOp)
execute(newMessageOp)
execute(newMessageOp)
execute()
Se ejecuta sobre la sesión del servidor
broadcast()
execute(newMessageOp)
onExecute(SessionEvent)

9.3.2 La ejecución

Para ejecutar nuestro programa necesitaremos dos tomcats. Por comodidad conviene tener instalados los dos servidores en el mismo host. Uno de ellos escuchará en el puerto 8080 (valores por defecto), el segundo en el puerto 8081. Notar que el código de prueba (escrito a mano) utiliza este último hecho para identificar quien tomará el rol del primer usuario, y quien el del segundo.

Luego de la instalación, se necesita instalar la aplicación en ambos servidores. En "UtilesTest/build.xml" hay código escrito para realizar dicha tarea.

El tercer paso es levantar ambos servidores.

Por último, observaremos el log de ejecución de una de las aplicaciones web desplegadas en uno de los contenedores web.

```
DEBUG gpfw.remoting.user.UserServiceInImpl - [IN- login]
DEBUG gen.client.remote.ChatSessionImplOut - [
createNewSession (Salida del cliente al servidor - ChatSessionImplOut => createNewSession)]
DEBUG gen.client.remote.ChatSessionImplLocalIn - [IN - createNewSession]
DEBUG gen.client.remote.ChatSessionImplOut - [OUT. execute operation.NewMessageOp. TXT: "MENSAJE
USUARIO 1. NUMERO:0"]
DEBUG gen.client.remote.ChatSessionImplLocalIn - [client. execute]
INFO  util.test.TestUtil - [MOSTRAR en pantalla NewMessageOp MENSAJE USUARIO 1. NUMERO:0]
DEBUG gen.client.remote.ChatSessionImplOut - [OUT. execute operation.NewMessageOp. TXT: "MENSAJE
USUARIO 1. NUMERO:1"]
DEBUG gen.client.remote.ChatSessionImplLocalIn - [client. execute]
INFO  util.test.TestUtil - [MOSTRAR en pantalla NewMessageOp MENSAJE USUARIO 1. NUMERO:1]
DEBUG gpfw.remoting.user.UserServiceInImpl - [IN- login]
DEBUG gen.client.remote.ChatSessionImplOut - [OUT. execute operation.NewMessageOp. TXT: "MENSAJE
USUARIO 1. NUMERO:2"]
DEBUG gen.client.remote.ChatSessionImplLocalIn - [client. execute]
INFO  util.test.TestUtil - [MOSTRAR en pantalla NewMessageOp MENSAJE USUARIO 1. NUMERO:2]
DEBUG gen.client.remote.ChatSessionImplLocalIn - [client. execute]
INFO  util.test.TestUtil - [MOSTRAR en pantalla NewMessageOp Mensaje de Usuario 2. NUMERO:0]
DEBUG gen.client.remote.ChatSessionImplOut - [OUT. execute operation.NewMessageOp. TXT: "MENSAJE
USUARIO 1. NUMERO:3"]
DEBUG gen.client.remote.ChatSessionImplLocalIn - [client. execute]
INFO  util.test.TestUtil - [MOSTRAR en pantalla NewMessageOp MENSAJE USUARIO 1. NUMERO:3]
DEBUG gen.client.remote.ChatSessionImplLocalIn - [client. execute]
INFO  util.test.TestUtil - [MOSTRAR en pantalla NewMessageOp Mensaje de Usuario 2. NUMERO:1]
DEBUG gen.client.remote.ChatSessionImplOut - [OUT. execute operation.NewMessageOp. TXT: "MENSAJE
USUARIO 1. NUMERO:4"]
DEBUG gen.client.remote.ChatSessionImplLocalIn - [client. execute]
INFO  util.test.TestUtil - [MOSTRAR en pantalla NewMessageOp MENSAJE USUARIO 1. NUMERO:4]
DEBUG gen.client.remote.ChatSessionImplLocalIn - [client. execute]
INFO  util.test.TestUtil - [MOSTRAR en pantalla NewMessageOp Mensaje de Usuario 2. NUMERO:2]
DEBUG gen.client.remote.ChatSessionImplOut - [OUT. execute operation.NewMessageOp. TXT: "MENSAJE
USUARIO 1. NUMERO:5"]
DEBUG gen.client.remote.ChatSessionImplLocalIn - [client. execute]
INFO  util.test.TestUtil - [MOSTRAR en pantalla NewMessageOp MENSAJE USUARIO 1. NUMERO:5]
```

Se resalta el ingreso del primer usuario, el envío de un mensaje, luego el ingreso del segundo. Luego se alterna la recepción de mensajes del segundo con el envío de mensajes del primero, y su respectivo eco en la interfaz de entrada.

La lista de tres logs generados es la siguiente:

- `${tomcat1}/logs/snClient.debug`
- `${tomcat1}/logs/snServer.debug`
- `${tomcat2}/logs/snClient.debug`

Es decir, un log para cada uno de los dos clientes, y uno más para el servidor central.

10 CONCLUSIÓN

Este epílogo se divide en tres secciones. En la primera se realiza un repaso crítico de los contenidos teóricos expuestos. En la segunda, se discuten las ventajas, desventajas, y otras valoraciones del trabajo realizado, enfocándose especialmente en el software desarrollado. Finalmente se reflexiona sobre los posibles puntos de extensión de la tesina.

10.1 Repaso y conclusiones generales

La aplicación de modelos al desarrollo de software es una larga tradición, y se hizo aún más popular con el desarrollo de UML (Unified Modelling Language). Sin embargo, a menudo se trabaja con simple documentación, porque la relación entre el modelo y la implementación es simplemente una intención, no es formal. A este tipo de uso de modelos, en un proceso de desarrollo, lo llamamos "basado en modelos" ("model-based")[**12**].

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. Dentro del Desarrollo Dirigido por Modelos, existen varios enfoques. La OMG propone un grupo de estándares y tecnologías con su Arquitectura Dirigida por Modelos. Otro enfoque es el Modelado específico de Dominio.

Existen tres objetivos primordiales en el desarrollo de software (calidad, productividad, longevidad). Durante el desarrollo de la historia de la computación, pocas veces se logró un aumento significativo de estas tres variables al mismo tiempo. Y eso se logró de una única manera: **el aumento del nivel de abstracción**. La función de los modelos es, justamente, aumentar el nivel de abstracción de un sistema.

Los DSLs se caracterizan por aumentar el nivel de abstracción. Los usuarios deben especificar un modelo, y a partir de ese modelo se derivará el código del sistema. Esto es posible gracias a otra característica: concentrarse en un área de interés reducida.

Como su nombre lo indica, es específico a un dominio, no de propósito general. Cuanto más reducido es el centro de atención del lenguaje, más fácil resulta proveer soporte para la especificación de modelos y la automatización de la generación de código. Esto se debe a que el lenguaje y el generador de código conocen el dominio, y por lo tanto pueden completar la brecha semántica entre el modelo de entrada y el código de la aplicación.

Por otro lado, los sistemas colaborativos son sistemas basados en computadoras que ayudan a un grupo de personas comprometidas en una tarea u objetivo en común, y que proveen una interfaz a un ambiente compartido.

CSSL es un lenguaje específico al dominio de los sistemas colaborativos, y podría mejorar la productividad en el desarrollo inicial de estos sistemas. En el trabajo se mostró cómo aprovechar la información contenida en un modelo para generar algunos artefactos, que pueden solucionar algunos problemas en el desarrollo de este tipo de sistemas, ayudados por un framework apropiado.

Podría tomarse este trabajo como una primera aproximación a la aplicación de MDD al dominio de los sistemas colaborativos. Tal como se espera de cualquier generación de código a partir de un lenguaje específico de dominio, se estudio previamente el dominio para detectar características comunes a los sistemas de este tipo. Resulta útil conocer términos habituales en el desarrollo de groupware como "conciencia de grupo" (awareness), sesiones, objetos compartidos, etc.

Luego se estudiaron técnicas y herramientas para la generación de código. Para las características del metamodelo en cuestión resultó ser suficiente JET, una herramienta muy simple basada en plantillas. También se analizó Mofscript, que promete ser una herramienta excelente, con un lenguaje muy expresivo pensado para la conversión de un modelo MOF a texto. Pero la experiencia con Mofscript no fue del todo satisfactoria, por razones descriptas en el trabajo. En pocas palabras, hay características necesarias en el caso de estudio que aún no están documentadas (a tal punto que no puedo saber si están implementadas).

Por otro lado, se plantea como interrogante si es posible la generación total de código a partir del metamodelo actual del lenguaje CSSL. Es claro que no es posible, ya que generar un código final a partir de un modelo requiere que el generador de código aporte una cuota excesiva de información semántica a diagramas que no tienen demasiados elementos. Sin embargo, el lenguaje puede ser usado para generar código que permita iniciar el desarrollo de un sistema. Además, es posible mantener el modelo como documentación, y regenerar el código si se desea extender el modelo con nuevos elementos.

10.2 Ventajas, desventajas y otras apreciaciones

El trabajo tiene dos aspectos a valorar. Por un lado, la construcción de un software capaz de generar código automáticamente y por lo tanto **facilitar el desarrollo de aplicaciones colaborativas**. Por otro lado, el trabajo realiza una **exploración inicial del tema**, de la aplicación de las técnicas de MDD sobre los sistemas colaborativos.

La creciente importancia de los sistemas colaborativos no está en discusión, en un mundo donde la comunicación por medios digitales nos sorprende con nuevas aplicaciones que exploran nuevas maneras de comunicarse.

También parece prometedor el **futuro del desarrollo dirigido por modelos**. La historia del desarrollo de software muestra que el punto en común en los grandes avances en cuanto a productividad y calidad es el aumento del nivel de abstracción. Cuando se crearon los primeros compiladores el programador pudo abstraerse de los detalles del lenguaje ensamblador, para trabajar con otros conceptos más abstractos. Algo similar se propone MDD, y especialmente el modelado específico de dominio. En los últimos años se han visto aplicaciones en donde el programador trabaja con conceptos particulares a un dominio.

El plugin resultante de este trabajo hace que el programador **piense menos en términos de código y más en conceptos del dominio** de los sistemas colaborativos. Eso evita que el programador tenga que pensar en algunos detalles comunes al tipo de sistemas. La comunicación entre los usuarios, la coordinación y la coherencia del estado compartido, el broadcast de las operaciones a todos los participantes de una sesión, el mantenimiento de los usuarios y datos de una sesión; son las tareas que se evitan utilizando la generación automática de código.

Pero, en el caso de los sistemas colaborativos, y en particular de CSSL, no es posible hacer que la **generación de código sea completa**. La recomendación de la bibliografía es reducir el dominio. Definir el lenguaje de tal manera que permita la automatización de tareas en un dominio extremadamente reducido. Y CSSL es todavía **demasiado amplio** en su interpretación. Las operaciones de cada sistema en particular no pueden ser especificadas en un lenguaje nuevo, pues pueden variar muchísimo entre varios sistemas.

La generación completa de código es lo que permite **aumentos de productividad** del 400%. CSSL no podría alcanzar ese nivel de mejora en la productividad, porque como ya se explicó, no es posible la generación total. Pero sí nos permite considerar al **modelo como algo central en el desarrollo**. Por cada cambio en el sistema se puede modificar el modelo, y ese cambio no afectará los cambios realizados manualmente por el usuario, porque se utilizan regiones protegidas para que eso suceda. Regiones protegidas es el manejo más directo de este tipo de situaciones, sus alternativas (insertar en el modelo pedazos pequeños de código o vincular el modelo con clases escritas en una IDE) serían posibles sólo luego de una maduración del proyecto, maduración que incluye no sólo al generador, sino también al lenguaje.

Pero además de la productividad existen otras ventajas. Por ejemplo, una ventaja de la generación automática de código a partir de modelos es que permite **forzar la arquitectura del sistema**, porque algunas decisiones de arquitectura de software serán resueltas por el generador. Y el generador debe ser escrito por expertos en el dominio. Por lo tanto programadores con menos experiencia pueden aprovechar la experiencia de quienes escribieron el generador. En este sentido, CSSL evita que el usuario deba leer acerca del diseño de frameworks para sistemas colaborativos, aunque sigue siendo algo deseable.

También es necesario considerar el papel de los **frameworks de dominio**, y en particular el caso de CSSL. Se recomienda que el código generado por el generador utilice un framework de dominio. Esto permite que el generador se ocupe sólo de las características del sistema que varían según el modelo. Lo que es común a todos los sistemas generados debe estar en el framework.

En CSSL existía la posibilidad de utilizar alguno de los frameworks existentes en el dominio, y crear una delgada envoltura que modelaría los conceptos específicos que propone CSSL. Sin embargo, se decidió escribir desde cero el código de este framework. Esta decisión se debe a que los frameworks que se estudiaron no resultaron apropiados. Algunos no estaban escritos en Java, lo cual era un inconveniente (groupKit, COAST). Otros, son propietarios o no se puede tener acceso al código (DyCe, por ejemplo). Otros están enfocados en características que no son las necesarias por el trabajo. El framework de Google Wave parece apropiado, pero la liberación de su código fue posterior a la mayor parte de este trabajo. Un interesante trabajo a futuro podría ser reemplazar el framework actual, y pensar en que utilice Waves y Wavelets para comunicar las operaciones entre distintos clientes.

El **framework utilizado** existe solo para que exista código que represente los conceptos del dominio. Pero no tiene como objetivo ser un framework confiable, completo y robusto. Es probable que no maneje de la mejor manera desconexiones inesperadas de los usuarios, fallos en la red, fallos en servidores, etc..

Normalmente se dice que para escribir un framework se debe encontrar varias veces un determinado problema mientras se escribe código, y luego buscar la manera de evitarlo agrupando la funcionalidad común en clases, formando nuevos conceptos abstraidos en el framework. Una forma apropiada para el desarrollo de un DSL podría ser desarrollar primero el framework, comenzar con su utilización, refinar sus conceptos, hasta que se convierta en un framework completo, estable y robusto. Luego de haber identificado los conceptos, continuar con el lenguaje y el generador.

Un punto a destacar es el carácter de **exploración inicial** de este trabajo. Su objetivo no es sólo facilitar el desarrollo de aplicaciones colaborativas, sino que incluye un estudio completo del tema, que permitirá a quién lo continúe tomar alguna de las muchas extensiones posibles. Una lista de extensiones posibles es dada en la sección **"Trabajo Futuro"**.

10.3 Trabajo Futuro

Se plantean las siguientes líneas de trabajo:

1. **Framework y plataforma**: este trabajo se enfoca en la generación de código, pero descuida el framework y la plataforma necesarias para un sistema groupware. Se puede desarrollar un framework robusto que contenga los conceptos del dominio que describe CSSL, y que si es necesario, se apoye en algún framework existente.
2. En el trabajo se mostró cómo aprovechar la información contenida en un modelo para generar algunos artefactos. Sin embargo, el dominio es demasiado extenso y CSSL no puede cubrirlo completamente. Esto lleva a dos caminos posibles: reducir el dominio al que apunta la generación o enriquecer el lenguaje para cubrir más posibilidades. Sin embargo, no son posibilidades excluyentes, y se pueden llevar a cabo ambas tareas simultáneamente. Un ejemplo apresurado podría ser: aplicar los conceptos de CSSL al subdominio de los workflows, adaptarlos, usar uno de los muy buenos frameworks que existen en dicho subdominio. La reducción del dominio podría permitir la generación de la mayor parte del código de la aplicación.
3. En trabajo mucho más lejano podría ser incorporar y mezclar conceptos de otros dominios en modelos CSSL. El más claro es la interfaz gráfica, donde se podrían relacionar las herramientas modeladas con CSSL con el modelado de interfaces gráficas.

11 REFERENCIAS BIBLIOGRÁFICAS

[1] OMG, UML Infrastructure Specification, v2.1.2, Noviembre 2007. Disponible en http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF

[2] OMG, The Unified Modeling Language Superstructure version 2.1.2. Noviembre 2007. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF

[3] Kelly, Steven and Tolvanen,Juha-Pekka, Domain-Specific Modeling: Enabling Full Code Generation (2008) John Wiley & Sons, Inc

[4] K. Hussey, Getting Started with UML2, eclipse.org (August 4, 2005), http://www.eclipse.org/uml2

[5] J. Bruck and K. Hussey, 'Customizing UML', eclipse.org (June 19, 2008), http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html, accedido el 03/09/2009

[6] Fowler M., 'UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition', Addison Wesley (2004)

[7] Budinsky F., Steinberg, D., Ellersick R., Grose T., 'Eclipse Modeling Framework: A Developer`s Guide'. Addison Wesley (2003)

EMF. EMF Home Page. http://www.eclipse.org/modeling/emf/

[8] Powell A., 'Model with de Eclipse Modeling Framework', ibm.com (2004), 'http://www.ibm.com/developerworks/library/os-ecemf3/' accedido el 11/11/2009

[9] Perez G., Tesis de Licenciatura en Informática de la UNLP: "Un aporte gráfico a las herramientas para transformaciones entre modelos". Directora: Claudia Pons. (2008)

[10] Ruiz G., Andino L., Tesis de Licenciatura en Informática de la UNLP: "Análisis y uso de los frameworks de Eclipse para la definición de DSLs". Directores: L.M. Bibbó, C. Pons (2009)

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

[12] Thomas Stahl , Markus Voelter , Krzysztof Czarnecki, Model-Driven Software Development: Technology, Engineering, Management, John Wiley & Sons, 2006

[13] Bibbo, L. M.; García, Diego; Pons, C.; "A Domain Specific Language for the Development of Collaborative Systems," Chilean Computer Science Society, International Conference of the, pp. 3-12, 2008 International Conference of the Chilean Computer Science Society, 2008

[14] Mandel, L. And Koch, N. And Maier, C.: 'Extending UML to Model Hypermedia and Distributed Systems'. From: http://projekte.fast.de/Projekte/forsoft/intoohdm/

[15] Rubart, Jessica and Dawabi, Peter. Shared data modeling with UML-G. International Journal of Computer Applications in Technology, Volume 19, Nos. 3/4, 2004.

**[16] Dong, Ying and Li, Mingshu and Wang, Qing. A UML Extension of Distributed System. Proceedings of the Fist
IEEE International Conference on Machine Learning and Cybernetics, Beijing, 4-5 November 2002**

[17] Pinheiro da Silva, P. and Paton, N.W. UMLi: the unified modeling language for interactive applications. Proceedings of the UML International Conference 2000, LNCS, Vol. 1939, pp. 117-132. (2000)

[18] Schuckmann Christian, Kirchner Lutz, Schümmer Jan. Haake Jörg M (1996) Designing object-oriented synchronous groupware with COAST. Proceedings of the 1996 ACM conference on Computer supported cooperative work

[19] Miguel A. Martinez-Prieto, Pablo de la Fuente, and Carlos E. Cuesta. Agora: a layered architecture for cooperative work environments. In ENC '07: Proceedings of the Eighth Mexican International Conference on Current Trends in Computer Science, pages 73–80, Washington, DC, USA, 2007. IEEE Computer Society.

[20] D.A. Tietze, J. Rubart: 'DyCE- A Framework for Component-Based Groupware' (2001). http://www.go4teams.com

[21] C. A. Ellis , S. J. Gibbs, Concurrency control in groupware systems, Proceedings of the 1989 ACM SIGMOD international conference on Management of data, p.399-407, June 1989, Portland, Oregon, United States

[22] Leandro Quiroga and Alejandro Fernandez. A p2p groupware framework based on operational transformations. In ICDCSW '07: Proceedings of the 27th International Conference on Distributed Computing Systems Workshops, page 70, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Bibbo, L. M., Tesis de Magister en Ingeniería de Software de la UNLP: "Modelado de Sistemas Colaborativos". Directora: Claudia Pons. (2009)

[24] Lamport, Leslie, 'Time, clocks, and the ordering of events in a distributed system', Commun. ACM , vol. 21, no. 7, 558-565 (1978).

[25] M. Koch and T. Gross, "Computer-Supported Cooperative Work - Concepts and Trends," Proc. Conf. of the Association Information And Management (AIM), 2006.

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

[27] M. Kirsch-Pinheiro, Valdeni, and M. R. S. Borges. A framework for awareness support in groupware systems. In The 7th International Conference on Computer Supported Cooperative Work in Design, 2002., pages 13–18, 2002.

[28] Dias, M.S., and Borges, M.R.S, 1999, Development of groupware systems with the COPSE infraestructure, Proceedings of International Workshop on Groupware (IEEE Computer Society, Cancun), 278-285

[29] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design patterns: elements of reusable object-oriented software; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1995.

[30] Frankel, D: 'Model Driven Architecture, Applying MDA to Enterprise Computing'. (2003). Wiley.

[31] Kent Beck. Extreme Programming Explained: Embrace Change. Boston: Addison Wesley, 2000

[32] John Poole: 'Model-Driven Architecture: Vision, Standards And Emerging Technologies'. ECOOP 2001.

[33] Atul Prakash. Group editors. In Michel Beaudouin-Lafon, editor, Computer Supported Co-operative Work, volume 7 of Trends in Software, pages 103–133. John Wiley & Sons, 1999.

[34] Saul Greenberg and Mark Roseman. Groupware toolkits for synchronous work. In Michel Beaudouin-Lafon, editor, Computer Supported Cooperative Work, volume 7 of Trends in software, pages 135–168. John Wiley & Sons, 1999.

[35] MOF 2.0/XMI Mapping Specification, v2.1, Object Management Group, Framingham, Massachusetts, September 2005.

[36] http://www.waveprotocol.org/

OMG (Object Management Group) http://www.omg.org

The Eclipse Project. Home Page. Copyright IBM Corp, 2000-2005. http://www.Eclipse.org/.

MofScript. MOFScript Home page - www.eclipse.org/gmt/mofscript/.

Kleppe, Anneke G. and Warmer Jos, and Bast, Wim. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

Pinheiro & Lima & Borges. A Framework for Awarness Support in Groupware Systems