FACULTAD DE INFORMATICA
UNLP

TESINA DE LICENCIATURA

TITULO: Impacto de reglas de refactorización en diagramas UML con restricciones OCL

AUTORES: Santiago Scolari

DIRECTOR: Dra. Claudia Pons

CODIRECTOR:

CARRERA: Licenciatura en Informática

Resumen

En los últimos años, el desarrollo de software ha ido generando una serie de inconvenientes, debido mayormente a los procesos de desarrollo utilizados.

El desarrollo dirigido por modelos (MDA) es una técnica de desarrollo de software que se centra en la creación de modelos para reducir la complejidad con la que tienen que tienen que lidiar los desarrolladores de software. Se basa en la creación y la evolución de modelos.

Los modelos en MDA son representados mediante UML y se van transformando a medida que avanza el desarrollo mediante sucesivos refinamiento. En muchos casos, el lenguaje UML no es suficiente para lograr una especificación precisa y no ambigua del problema que se está representando. En estos casos, el lenguaje de restricciones OCL se utiliza para complementar los diagramas UML.

En esta tesis se investigan las reglas de refactorización para diagramas UML, y se extienden para que tengan en cuenta el código OCL asociado. También se implementan las mismas en una herramienta CASE de soporte de MDA.

Líneas de Investigación

- Ingeniería de Software
- MDA
- Transformación de modelos
- Herramientas de soporte de MDA

Conclusiones

MDA está cobrando mucha fuerza en los últimos años. Mediante el uso de modelos y refactorizaciones de los mismos, pretende combatir muchos de los problemas actuales del desarrollo de software.
Sin embargo, para que la industria adopte este proceso de desarrollo es necesario contar con herramientas adecuadas. ePlatero es una de estas herramientas, pero, no proveía soporte para el refinamiento de diagramas UML con código OCL asociado.
Con el nuevo catálogo de reglas de refactorización y la extensión de ePlatero para el soporte de estas reglas, se avanzó un paso más hacia la adopción de este proceso de desarrollo por parte de la industria del software.

Trabajos Realizados

En el presente trabajo se investigaron las reglas de refactorización para diagramas UML. En particular, se analizaron estas reglas en diagramas con código OCL asociado.
Luego se creó un catálogo de reglas de refactorización, las cuales aplican a diagramas UML enriquecidos con restricciones OCL.
Estas nuevas reglas se implementaron en ePlatero.

Trabajos Futuros

Como trabajo futuro se propone seguir extendiendo el catálogo de reglas de refactorización. En esta tesis solamente se definió un conjunto de reglas, para la comprensión del tema de estudio y para demostrar la factibilidad de este tipo de transformaciones.
En una extensión de este trabajo podría ampliarse el catálogo, para abarcar mayor cantidad de opciones de refactorización.

Fecha de la presentación:

Índice

Capítulo 1: Introducción

1.1. Introducción

En los últimos años, el desarrollo de software ha ido generando una serie de inconvenientes, debido mayormente a los procesos de desarrollo utilizados. Los desarrolladores de software pasan la mayor parte del tiempo modificando y manteniendo productos existentes. Los sistemas junto con su diseño están en constante evolución. Manejar esta complejidad es una tarea difícil.

El desarrollo dirigido por modelos (MDA) es una técnica de desarrollo de software que se centra en la creación de modelos para reducir la complejidad con la que tienen que tienen que lidiar los desarrolladores de software. Mediante la creación y la evolución de los mismos, se consigue reducir los problemas del desarrollo tradicional de software.

Los modelos en MDA son representados mediante UML y se van transformando a medida que avanza el desarrollo de un proyecto de software mediante sucesivos refinamiento [1]. Existen diversas herramientas CASE que dan soporte a la creación de modelos así como también a las refactorizaciones de los mismos.

En muchos casos, el lenguaje UML no es suficiente para lograr una especificación precisa y no ambigua del problema que se está representando. En estos casos, el lenguaje de restricciones OCL se utiliza para complementar los diagramas UML.

Existen reglas de refactorización para diagramas UML, pero ninguna tiene en cuenta las restricciones OCL asociadas. Las herramientas CASE actuales tienen el mismo problema.

1.2. Objetivos

Esta tesis tiene como objetivo principal hacer un aporte a la propuesta planteada por MDA.

Se estudiarán las transformaciones de diagramas UML, en particular las correspondientes a diagramas UML con restricciones OCL. Se analizará la manera de extender las reglas de refactorización al código OCL asociado.

Además de ello, un objetivo adicional es hacer un aporte a las herramientas CASE actuales que dan soporte a MDA. Se implementará en ePlatero la solución encontrada al problema de las restricciones OCL en las refactorizaciones de los diagramas UML.

Capítulo 2: Introducción a Model Driven Architecture

2.1. Introducción

La OMG (Object Management Group) [8] es un consorcio dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos. Es una organización sin fines de lucro que promueve el uso de tecnologías orientadas a objetos mediante guías y especificaciones para las mismas.

Model-Driven Architecture (MDA) [11] es una iniciativa de OMG. Es una implementación de MDD (Model Driven Development). MDD no define tecnologías, herramientas, procesos o secuencia de pasos a seguir. La implementación MDA se ocupad de ello mediante el uso de un lenguaje de modelado UML y meta-pasos a seguir en el desarrollo de sistemas. MDA se basa en la construcción y transformación de modelos.

Los modelos en MDA van evolucionando mediante sucesivas transformaciones, cada una de las cuales da como resultado otro modelo con menor nivel de abstracción. Las transformaciones finalmente generan modelos con características de una tecnología particular, que puede transformarse directamente a código ejecutable.

2.2. Generalidades

El objetivo de MDA es reducir el nivel de abstracción con el que los desarrolladores de software tienen que trabajar. Esto se realiza promoviendo el uso de modelos para los componentes que se están desarrollando. Los modelos son representaciones de fenómenos de interés, y son más fáciles de modificar, actualizar y manipular que los fenómenos a los cuales representan. Los modelos son expresados usando un lenguaje de modelado apropiado: *UML*.

Un concepto clave dentro de MDA es el de *refinamiento* de modelos. Sin embargo, los lenguajes involucrados en MDA son basados en UML, y no poseen el nivel de formalidad requerido en este tipo de procesos.

A pesar de que el concepto de refinamiento es clave en MDA, el término está pobremente definido y abierto a malinterpretaciones. Esto se debe a la poca formalidad de los lenguajes utilizados, pero también se debe a la inmadurez de MDA hasta la fecha.

2.3. El proceso MDA

El uso de MDD tiene muchos beneficios. En particular, incrementa la portabilidad y productividad. MDA estandariza elementos de MDD, definiendo un lenguaje para especificar modelos, y una secuencia de pasos para generar modelos.

MDA brinda los beneficios de MDD, al mismo tiempo que mejora la interoperabilidad (diferentes proyectos pueden compartir modelos) y facilita la integración de datos y aplicaciones estandarizando el proceso de desarrollo sistemas.

Uno de los principales puntos de MDA, es el concepto de plataforma y el de independencia de plataformas. Una plataforma en MDA es:

"... un conjunto de subsistemas y tecnologías que proveen un conjunto coherente de funcionalidades…".

Independencia de plataforma significa que un sistema debe ser modelado de modo que el modelo sea independiente de cualquier plataforma tecnológica. Esta definición abarca plataformas como J2EE, .NET o CORBA, pero también pueden también ser incluidos otros lenguajes de programación.

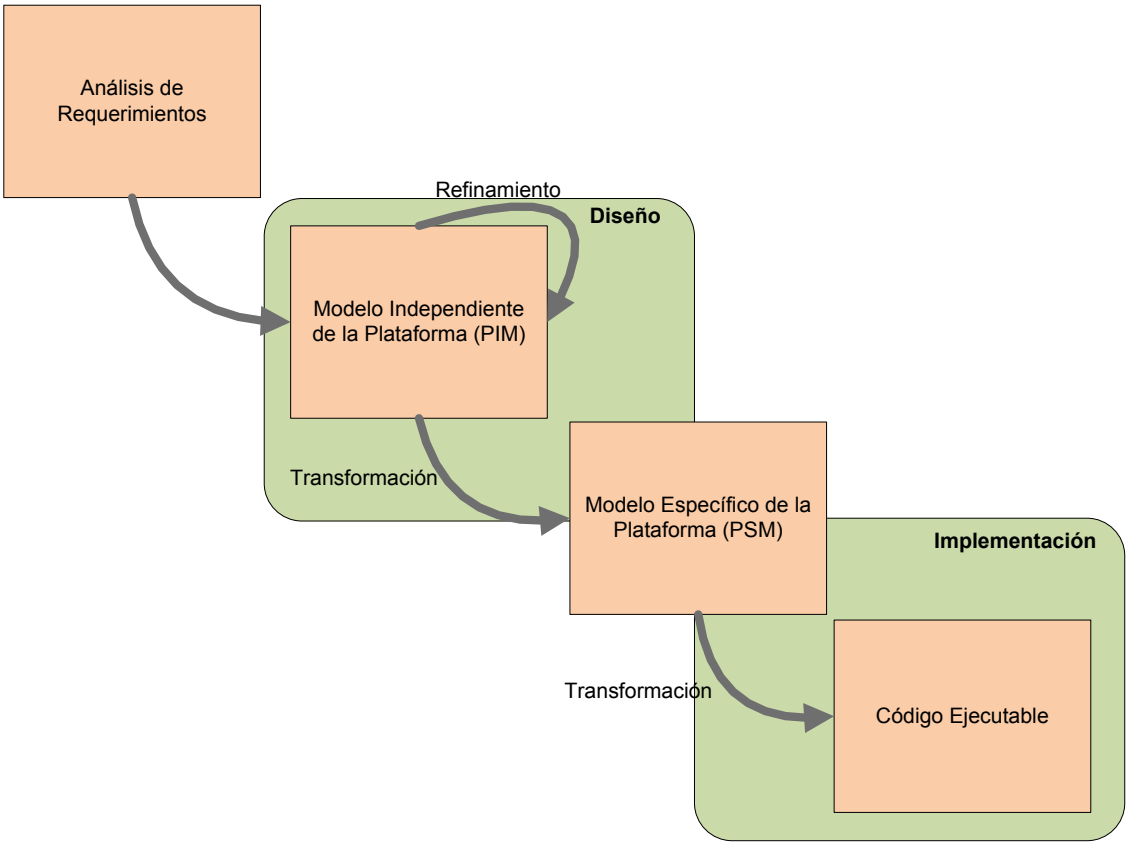
Los modelos abstractos son llamados modelos independientes de la plataforma (*PIMs*), en contraposición a los modelos que incluyen detalles de la plataforma, llamados modelos específicos de la plataforma (*PSMs*).

El proceso de MDA involucra sucesivos refinamientos los cuales generalmente son de dos tipos principales:

- **Transformaciones:** Son refinamientos que generan un nuevo tipo de modelo. Por ejemplo, transforman un PIM a un PSM o un PSM a código ejecutable. Cada transformación agrega detalles al modelo y también reducen el no-determinismo al tomarse decisiones de diseño. Por ejemplo, como representar datos, como implementar mensajes, etc.

- **Refinamientos:** Son transformaciones que preservan la semántica del modelo, y producen el mismo tipo de modelo. Por ejemplo, refinar un PIM en un nuevo PIM.

El proceso de MDA se muestra a continuación. En el grafico pueden verse los sucesivos refinamientos y transformaciones aplicados al modelo.



2.4. Modelos en MDA

A medida que se avanza en las transformaciones, los modelos se vuelven más concretos. Desde un modelo abstracto se llega a uno lo suficientemente concreto para ser compatible con una tecnología o plataforma en particular [13].

La situación inversa de llevar el código hacia un modelo concreto es lo que se conoce como ingeniería inversa. MDA promueve la separación entre las responsabilidades de requerimientos del negocio y las responsabilidades tecnológicas. La ventaja de esta “separación de responsabilidades” es que ambos aspectos pueden evolucionar individualmente sin generar dependencias entre sí.

Los diferentes tipos de modelos en MDA [12] son:

2.4.1. CIM (Computational-independent model)

El CIM se basa en los requerimientos y es el nivel más alto del modelo de negocios. Utiliza un lenguaje de modelado de procesos de negocio, y no UML. Describe el negocio y no un sistema de computación; se modela cada proceso de negocio y su interacción con personas y/o componente de hardware. Se basa en las interacciones entre procesos y responsabilidades de cada persona o componente.

Un objetivo fundamental del CIM es que pueda ser comprendido por cualquier persona que entienda el negocio y los procesos del mismo, ya que éste evita todo tipo de conocimiento especializado o de sistemas.

2.4.2. PIM (Platform-independent model)

El PIM (Modelo Independiente de la Plataforma) es un modelo que representa el proceso de negocio a ser implementado. Se utiliza UML para modelar el PIM. Este modelo representa los procesos y las estructuras del sistema, pero sin hacer ninguna referencia a la plataforma en la (o las) que será implementada la aplicación. Es el punto de entrada de todas las herramientas para MDA.

2.4.3. PSM (Platform-specific model)

El PSM (Modelo Específico de la Plataforma) es la proyección de los PIMs en una plataforma específica. Un PIM puede generar múltiples PSMs, cada uno para una tecnología distinta. Los PSMs son los que contienen los detalles específicos de los lenguajes de programación, las plataformas (CORBA, .Net, J2EE, ETC), de los sistemas operativos, etc.

2.4.4. Código Fuente

El código fuente está implementado en un lenguaje de programación de alto nivel, como por ejemplo Java, C#, C++, VB, JSP, etc. Idealmente, el código fuente debería derivarse del PSM y no debería requerir la intervención humana. La teoría de MDA dice que en un ambiente MDA maduro, no se debería pensar en el código fuente más que como simples archivos, o como un objeto intermedio para generar el ejecutable final.

Debido a que MDA no está todavía maduro, es prácticamente imposible llegar a no tener que tocar el código fuente. Los desarrolladores necesitan conocer la tecnología para complementar la generación de código, debuguear la aplicación y lidiar con muchos y variados errores. El despliegue de la aplicación si puede ser automatizado en su totalidad, con el avance actual de las tecnologías involucradas.

2.5. Ventajas de MDA

La ventaja principal de MDA está en la separación de responsabilidades. Por un lado se modelan los PIMs que representan los modelos del negocio, y por otro lado se modelan los PSMs con los detalles tecnológicos. Esto permite que ambos modelos evolucionen por separado.

Si se necesita modificar un aspecto técnico, basta con modificar el PSM sin que esto tenga impacto en los modelos de negocios. El modelado de la solución debe ser dirigido por el negocio. Un cambio en el negocio producirá un cambio en el código, pero no lo inverso. Los cambios en el código no deberían impactar en el negocio.

MDA también permite lidiar con la complejidad de los sistemas, modelando cada componente por separado y permitiendo su análisis y mejora. Además mejora la calidad de representaciones del negocio y procesos, mediante la utilización de modelos y la separación de responsabilidades.

2.6. Desarrollo tradicional Vs. Desarrollo con MDA

2.6.1. Desarrollo tradicional

Los procesos tradicionales de desarrollo de software incluyen las siguientes fases:

a) Especificación de requerimientos

b) Análisis detallado

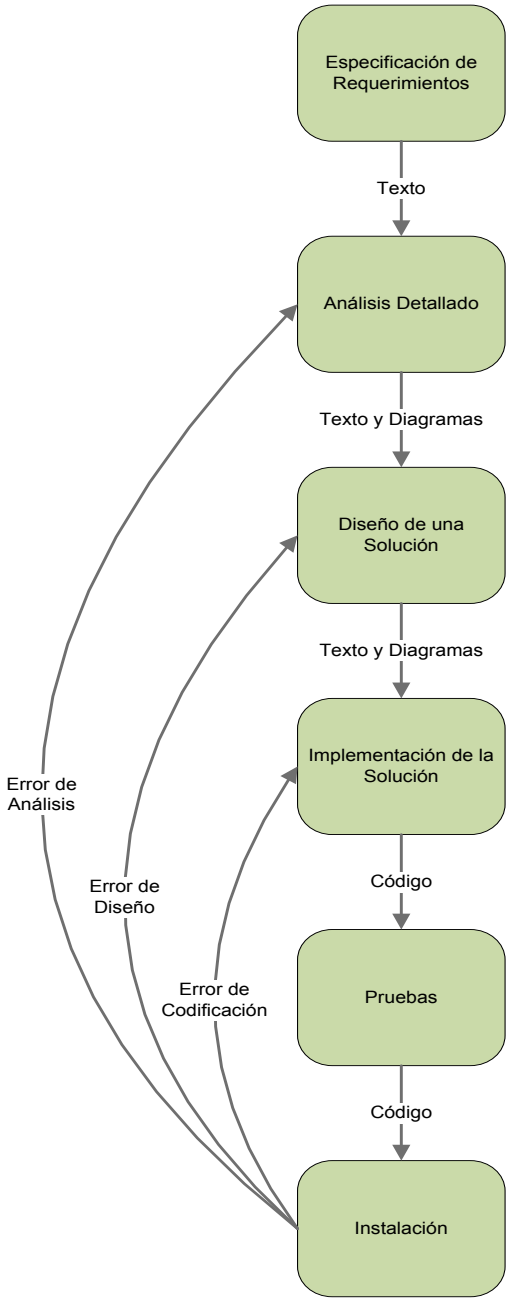
c) Diseño de una solución

d) Codificación de la solución

e) Pruebas

f) Instalación

El siguiente gráfico muestra el proceso:



En los últimos años se avanzó mucho en el desarrollo de software, lo que permitió construir sistemas más grandes y complejos. Sin embargo el enfoque tradicional del desarrollo de software presenta varios problemas:

- El problema de la Productividad. Durante el proceso tradicional se producen gran cantidad de documentos y diagramas, los cuales especifican requerimientos, diseño de componentes, etc. Gran parte de estos documentos pierden valor durante la fase de codificación, y finalmente se llega a un punto en el que pierden relación con el sistema. Cuando el sistema cambia a lo largo

del tiempo el problema es aún mayor: se hace imposible realizar los cambios en todas las fases (requerimientos, análisis, diseño, etc.), y las modificaciones se terminan realizando directamente en el código.

Para sistemas complejos, los diagramas y documentación de alto nivel son imprescindibles, pero está faltando un soporte para que los cambios en cualquiera de las fases se trasladen fácilmente al resto.

- El problema de la Portabilidad. La industria del software avanza muy rápido, y constantemente aparecen nuevas tecnologías y herramientas que brindan soluciones a problemas importantes. Es por esto que las empresas necesitan adaptarse a los avances tecnológicos, y muchas veces es necesario adaptar o migrar el software existente a nuevas tecnologías.

Esta migración es muy costosa, y demanda muchísimo esfuerzo.

- El problema de la Interoperabilidad. Los sistemas necesitan comunicarse con otros sistemas. Generalmente las soluciones de software se desarrollan sobre diferentes tecnologías respondiendo a necesidades del negocio, tecnológicas o políticas. La interoperabilidad entre sistemas debe lograrse de manera sencilla y uniforme, cosa que raramente ocurre.

- El problema del Mantenimiento y la Documentación. Documentar un sistema es una tarea costosa y que demanda mucho tiempo. Los desarrolladores de software no le dan a la documentación la importancia que realmente tiene, debido a que es mucho más usada en las etapas de mantenimiento y correcciones que durante el desarrollo en sí. Esto hace que no se le preste la atención necesaria a la documentación, y que la que se produce no sea de buena calidad.

Una solución es que la documentación se genere directamente del código fuente, asegurándose que esté siempre actualizada. Sin embargo, la documentación de alto nivel (diagramas y texto) debe ser mantenida a mano.

MDA brinda soluciones a estos problemas como se explica a continuación.

2.6.2. Desarrollo con MDA

En esta sección se explica cómo MDA resuelve los problemas recién explicados.

- Productividad. En MDA el desarrollo se basa en los PIM. Los PSMs son derivaciones semiautomáticas de estos. Es necesario definir las transformaciones exactas, lo cual es una tarea especializada y difícil. Pero una vez implementada una transformación, puede reutilizarse en muchos desarrollos. En la generación de código a partir de los PSMs ocurre lo mismo.

Este enfoque aísla los problemas específicos de las plataformas y ataca mejor las necesidades de los usuarios finales, ya que es posible agregar funcionalidades con menos esfuerzo. Gran parte del trabajo lo realizan las herramientas de transformación, y no los desarrolladores.

- Portabilidad. El problema de la portabilidad se resuelve en MDA al trabajar sobre los modelos PIM. Como son modelos independientes de la tecnología, las soluciones son completamente portables. Los detalles específicos de cada plataforma recaen sobre las transformaciones, en partículas sobre las de PIM a PSM.

- Interoperabilidad. De un mismo PIM pueden generarse múltiples PSMs, cada uno abarcando a una funcionalidad específica. En MDA, a las relaciones entre los diferentes PSM se las denomina "*puentes*".

Los PSMs no siempre pueden comunicarse entre sí, debido a que pueden pertenecer a distintas tecnologías. En MDA esto se soluciona generando tanto los PSMs como los puentes entre ellos. Estos puentes también son construidos por las herramientas de transformación.

- Mantenimiento y Documentación. En MDA, el PIM desempeña el papel de la documentación de alto nivel. La gran ventaja de MDA es que el PIM no se deja de utilizar tras la codificación. Los cambios a realizar en el sistema se aplican primero a los PIM, y desde allí se propagan a todos los niveles.

2.7. Modificaciones al proceso de desarrollo con MDA

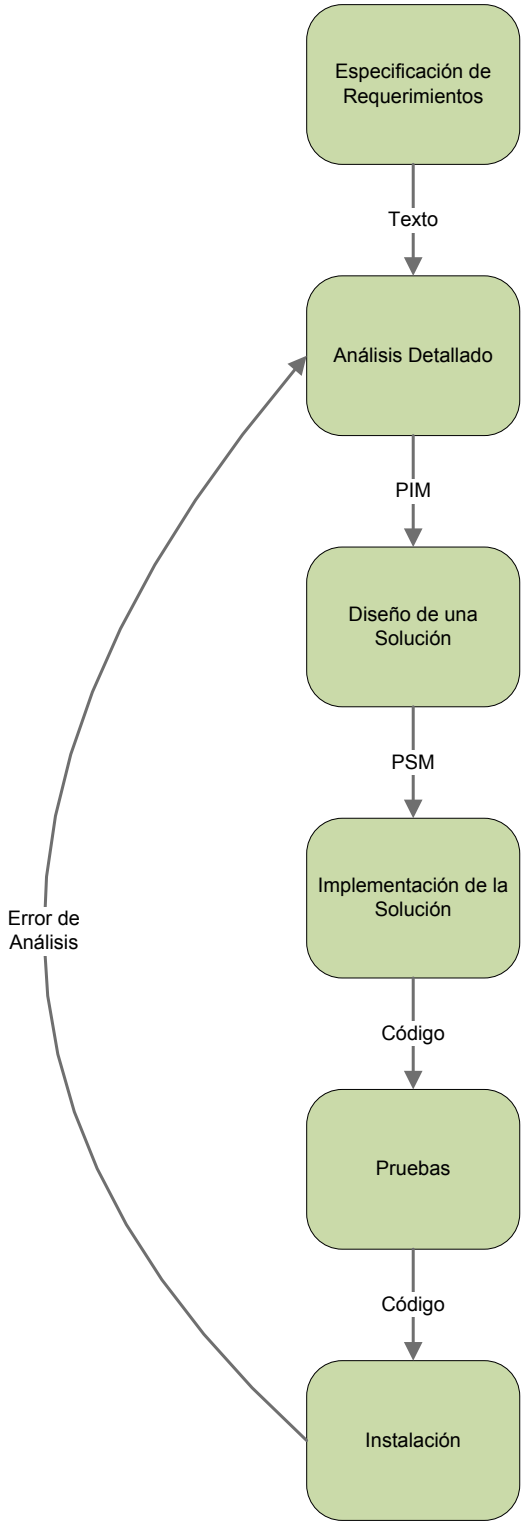
Las fases de desarrollo que abarca MDA son las de análisis, diseño y codificación, que se modifican de la siguiente manera respecto al desarrollo tradicional:

- Análisis: Los analistas del proyecto crean el PIM. En este modelo plasman las necesidades del cliente y las funcionalidades que el sistema debe proveer.

- Diseño: Un equipo especializado realiza la transformación del PIM a uno o más PSMs. Para este paso se requieren conocimientos específicos de distintas plataformas y arquitecturas, así como también de las herramientas y transformaciones disponibles. Con este conocimiento se logra una mejor elección de las plataformas y arquitecturas para cada caso en particular.

- Codificación: Esta fase se reduce a la generación del código fuente mediante herramientas especializadas. Los desarrolladores sólo deberán añadir las características que no pudieron ser reflejadas en los modelos, así como también realizar optimizaciones sobre el código generado.

En el siguiente gráfico se esquematizan los cambios que MDA introduce al proceso de desarrollo tradicional.



En el gráfico puede apreciarse como los errores o cambios detectados en la aplicación son corregidos en el PIM y desde allí se vuelven a ejecutar las transformaciones ya definidas para generar una nueva versión del software.

Capítulo 3: UML

3.1. Introducción UML

UML (Unified Modeling Language) [6] es un lenguaje de documentación y modelado para sistemas de software. Permite visualizar, especificar, construir y documentar software gráficamente.

Actualmente es un estándar de facto en la industria del software. Fue impulsado por Grady Booch, Ivar Jacobson y Jim Rumbaugh, quienes son muy reconocidos por sus métodos de modelado de sistemas orientados a objetos. Estos fueron contratados por Rational Software Co. para crear una notación unificada en la que basar sus herramientas CASE. También han participado varias empresas en el proceso de creación de UML, como por ejemplo Microsoft, Hewlett-Packard, Oracle e IBM, así como analistas y desarrolladores.

UML es muy aceptado debido al prestigio de sus creadores y a que conjuga las ventajas los métodos en los que se basa. Con UML finalizó la llamada “guerra de métodos”. Esta ocurrió en la década del 90, y en ella varios métodos de modelado competían por conquistar el mercado. Cuando alguno incorporaba una característica novedosa, los demás métodos se apresuraban a copiarla. Con UML se logró fusionar las notaciones y formar una herramienta compartida entre todos los desarrolladores de sistemas orientados a objetos.

3.2. ¿Qué es UML?

UML es la especificación de una notación orientada a objetos. Divide un sistema de software en un conjunto de diagramas, cada uno de los cuales modela un aspecto, componente o vista del sistema. La suma de los diagramas de un proyecto representa la arquitectura del mismo.

Con UML se puede especificar un sistema con diferentes niveles de abstracción. Logra simplificar la complejidad sin perder información. Esto permite que usuarios, líderes y desarrolladores puedan comunicarse mediante un lenguaje común.

UML apunta a convertirse en un lenguaje estándar para modelar todos los componentes del proceso de desarrollo de aplicaciones. Sin embargo, El estándar UML no define un proceso de desarrollo, sino que sólo se trata de una notación.

Distintos procesos de desarrollo utilizan UML. Esto permite que para diferentes dominios de trabajo se utilicen diferentes procesos, pero todos utilizando la misma notación.

En el caso concreto de esta tesis, las transformaciones son parte de MDA. Pero tanto el proceso de desarrollo como las transformaciones, están definidas sobre modelos de UML.

3.3. Modelos

El modelado es la construcción de un modelo partiendo de una especificación. Un modelo es una abstracción de algo. Se elabora para una mejor comprensión de ese "*algo*" antes de construirlo. Los modelos omiten los detalles que no son esenciales para comprender lo que están representando, y eso facilita la comprensión [15].

Un modelo UML está orientado a modelar objetos. Son abstracciones de un sistema o parte del mismo utilizadas durante el proceso de desarrollo de una pieza de software.

Además de la construcción de software, los modelos son usados en muchas actividades de la vida humana: los arquitectos utilizan planos para representar un edificio a construir, los músicos utilizan notas musicales como representación de la música, etc. Todos son modelos que abstraen una realidad compleja, para facilitar su comprensión por parte de terceros.

Como los modelos son representaciones que omiten detalles, permiten testear más fácilmente los sistemas que modelan y descubrir posibles errores.

Los modelos permiten una mejor comunicación con el cliente, por las siguientes razones:

- Permite mostrarle al cliente una aproximación de lo que será el producto final.
- Proporcionan una representación del problema que permite visualizar el resultado final.
- Reducen la complejidad del problema original en sub-problemas que pueden ser resueltos en diferentes etapas.

Un modelo es completo cuando captura todos los aspectos importantes de un problema y omite el resto. Los lenguajes de programación no son adecuados para realizar modelos completos, ya que necesitan de una especificación total, lo cual es muy difícil de conseguir especialmente en etapas tempranas del desarrollo. UML permite generar diagramas con diferentes niveles de abstracción, y son una muy buena herramienta para resaltar los aspectos importantes de la realidad que representan.

3.4. Estándares que conforman UML

- Infraestructura
- Superestructura
- OCL
- XMI / Intercambio de diagramas

- Infraestructura. La Infraestructura es un meta-modelo (un modelo de modelos), y se utiliza para modelar el resto de UML. En la Infraestructura de UML se definen los conceptos centrales y de más bajo nivel.

En general, los usuarios finales de UML no utilizan la infraestructura, pero es esta la que provee las bases sobre la cual se define la Superestructura. Esta última es la que utilizan los usuarios.

La Infraestructura brinda varios mecanismos de extensión, que hacen de UML un lenguaje configurable.

- Superestructura. La Superestructura de UML es la definición formal de los elementos de UML. Esta especificación contiene más de 640 páginas. Los desarrolladores de aplicaciones trabajan con la Superestructura para el modelado de sistemas.

- OCL. OCL son siglas, que en inglés que significan: Object Constraint Language (Lenguaje de Restricciones de Objetos). OCL define un lenguaje simple, que permite escribir restricciones y expresiones sobre elementos de un modelo.

En UML es utilizado para restringir los valores permitidos de los objetos de dominio, definir invariantes, pre-condiciones, post-condiciones y restricciones.

El OCL fue incorporado a UML en la versión 1.1, y originalmente fue especificado por IBM. Es un ejemplo de las muchas herramientas agregadas a UML.

- Especificación para el Intercambio de Diagramas. La especificación para el intercambio de diagramas permite compartir modelos UML entre distintas herramientas de modelado. En versiones previas de UML se utilizaba un Schema XML, pero este Schema no decía nada sobre como los modelos debían graficarse. Para solucionar esto, la nueva especificación fue desarrollada utilizando un nuevo Schema XML, y permite construir una representación SVG (Scalable Vector Graphics). Esta especificación es solamente utilizada por quienes desarrollan herramientas de modelado UML.

3.5. Diagramas

Los diagramas que se utilizan en el desarrollo de software están definidos en la Superestructura de UML. Se encuentran divididos en niveles:

- Básico (L1):
  - Diagramas de clases,
  - Diagramas de actividades,
  - Diagramas de Interacciones,
  - Diagramas de Casos de Uso
- Intermedio (L2):
  - Diagramas de estado,
  - Perfiles,
  - Diagramas de Componentes
  - Diagramas de despliegue.
- Completo (L3):
  - Acciones,
  - Características avanzadas
  - PowerTypes

Hay dos tipos básicos de diagramas, de estructura y de comportamiento.
Los diagramas de estructura representan los elementos que componen un sistema o una parte del mismo. Representan las relaciones estáticas entre los elementos. Algunos ejemplos son los diagramas de clases, de paquetes, de arquitectura, diagramas de objetos, etc.

Los diagramas de comportamiento representan el comportamiento de un sistema o de un proceso de negocios. Por ejemplo diagramas de actividades, casos de uso, maquinas de estados, tiempos, secuencias, repaso de interacciones y comunicaciones.

3.6. Diagramas de UML

Los diagramas más importantes de UML son los siguientes:

- Diagrama de Clases. Es el principal diagrama para el análisis y diseño. Es parte de la vista estática del sistema. En este diagrama se definen las características de las clases, interfaces, colaboraciones, relaciones de dependencia y generalización. Junto con las clases se definen atributos y operaciones.

- Diagrama de Componentes. Modela la vista estática de un sistema. Representa la organización y las dependencias entre los componentes de un sistema. Suele realizarse por partes, construyendo diferentes diagramas que representen partes del sistema.

- Diagrama de Estructura de Composición. Este diagrama representa la estructura interna un componente, un caso de uso o de una clase, incluyendo los puntos de interacción con otras partes del sistema.

- Diagrama de Despliegue Físico. Un diagrama de despliegue físico muestra de qué manera se desplegará el sistema. Incluye los nodos, hardware, software y el middleware que los conecta.

- Diagrama de Objetos. En este diagrama se representan las instancias de las clases modelados en el diagrama de clases. Muestra un momento concreto de la ejecución del sistema con los objetos y sus relaciones.

- Diagrama de Paquetes. Muestra como se divide un sistema grande en unidades más pequeñas.

- Diagrama de Actividades. El diagrama muestra la secuencia de actividades realizadas.

- Diagrama de Comunicaciones (anteriormente Diagrama de Colaboraciones). Muestra como los objetos colaboran entre sí. Modela relaciones entre los objetos y el orden en que se envían los mensajes.

- Diagrama de Revisión de la Interacción. Similar al diagrama de actividades. Muestra el flujo de control dentro de un sistema.

- Diagrama de Secuencias. Este diagrama es un modelo dinámico del sistema. Representa los llamados a métodos entre clases, para la realización de una funcionalidad en particular.

- Diagrama de Máquinas de Estado. Muestra los posibles estados de un objeto. Están orientados a representar objetos de una clase y los diferentes estados de sus instancias en el sistema.

- Diagrama de Tiempos. Este diagrama muestra los cambios en el estado de una instancia a lo largo del tiempo.

- Diagrama de Casos de Uso. Se emplean para modelar el comportamiento del sistema, o una parte del mismo. Define como debería ser el comportamiento del sistema y no como se implementa. Se utiliza también para que los analistas se comuniquen con los clientes o expertos de dominio sin llegar a niveles altos de complejidad.

Capítulo 4: OCL

4.1. Introducción

OCL (Object Constraint Language) [7] es un lenguaje formal para la expresión de restricciones. Dentro de los diagramas UML, se utiliza para especificar restricciones y expresiones. Tiene características de un lenguaje de expresión, de modelado y formal.

- Lenguaje de expresión. OCL es un lenguaje de expresión puro. Sus expresiones no tienen efectos colaterales, es decir, no permite cambiar el modelo. El estado del sistema no cambiará como consecuencia de una expresión OCL. Sin embargo, las expresiones OCL pueden utilizarse para especificar un cambio de estado, como es el caso de una post-condición.

- Lenguaje de modelado. OCL no es un lenguaje de programación, por lo que no permite escribir lógica de programa o flujos de control. Debido a que OCL es un lenguaje de modelado, permite modelar cosas que no sean ejecutables. Toda consideración de implementación está fuera de alcance y no puede expresarse en OCL. Cada expresión OCL es atómica, el estado de los objetos en el sistema no puede variar durante la evaluación.

- Lenguaje Formal. OCL es un lenguaje formal en el que todos los elementos tienen un significado definido formalmente. La especificación del OCL es parte del UML.

4.2. ¿Por qué OCL?

El un diagrama de clases UML, el modelo gráfico no es suficiente para lograr una especificación precisa y no ambigua. Es necesario especificar características adicionales sobre los objetos del modelo.

Algunas veces estas características se escriben en lenguaje natural o se omiten. Esto produce ambigüedades. Los lenguajes formales están diseñados para escribir estas condiciones no ambiguas.

El problema de los lenguajes formales tradicionales, es que están orientados a personas con una fuerte formación matemática, y es complicado modelar sistemas con ellos.

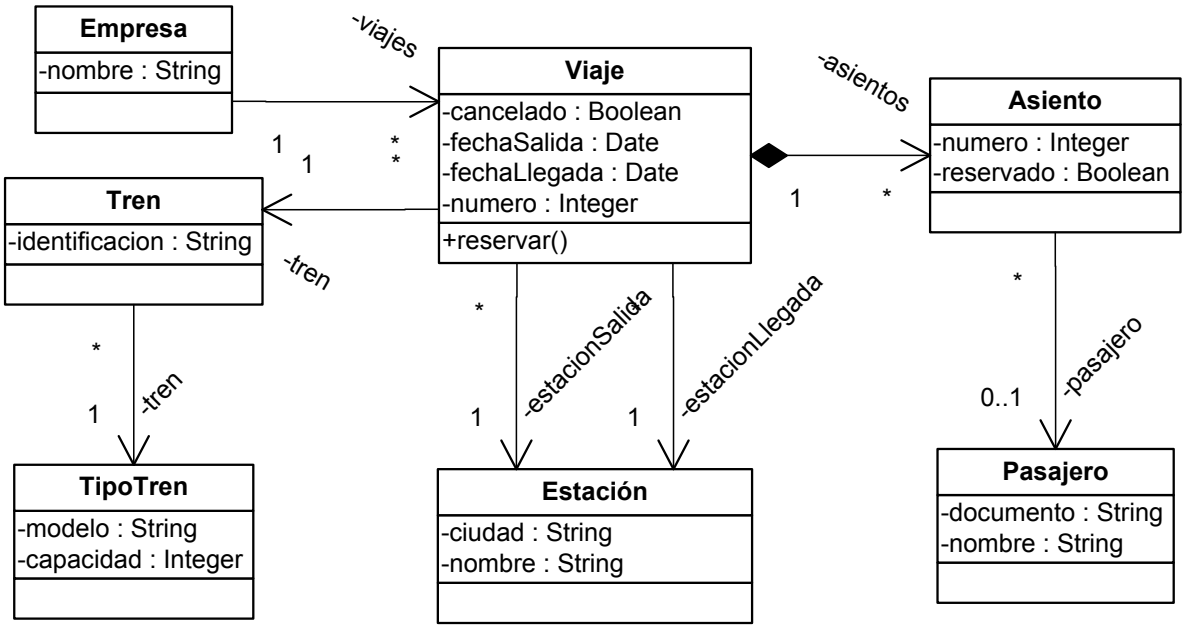
OCL viene para cubrir esta brecha. Es un lenguaje formal, que a la vez es fácil de leer y escribir [5].

4.3. Elementos de OCL

Los elementos principales de OCL [16] son:

- Restricciones
  - Invariantes
  - Definiciones
  - Precondiciones
  - Postcondiciones
- Expresiones de valor inicial
- Expresiones de valor derivado
- Expresiones de consulta

Para explicar cada uno de estos elementos se utiliza el siguiente modelo:



4.3.1. Restricciones

4.3.1.1. Invariantes

Un invariante es una restricción que se liga a una Clase o Interface.

Define una condición que debe ser válida en todas las instancias de la clase o interface a la cual aplica.

Se escriben con el estereotipo ***<<invariant>>***. Los invariantes se ligan a un solo tipo, y opcionalmente permiten definir una variable contextual.

Sintaxis:

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

Ejemplo:

El siguiente ejemplo sirve para expresar que la fecha de salida de un tren siempre debe ser menor a la fecha de llegada.

context *Viaje*
inv: *self.fechaSalida <= self.fechaLlegada*

Al comienzo de toda expresión OCL es necesario especificar el contexto al cual se restringe la expresión. Esto se especifica escribiendo *context* seguido de la definición del contexto, en este caso particular, la clase *Viaje*.

La palabra clave *self* se utiliza para referirse a una instancia del contexto. En este caso particular, *self* se refiere a una instancia de la clase *Viaje*.

Otra posibilidad es usar un alias para referirse a las instancias, de la siguiente manera:

context *v: Vuelo*
inv: *v.fechaSalida <= v.fechaLlegada*

La declaración previa indica que la variable ***v*** corresponde a una instancia de la clase *Vuelo*.

4.3.1.2. Definición

Una definición es una restricción que permite definir propiedades u operaciones y asociarlas a una clase o interface. El elemento definido puede utilizarse luego como parte de la clase o interface para la cual está definida. El propósito de esta restricción es definir expresiones OCL reusables.

Las definiciones se escriben con el estereotipo ***<<definition>>***.

Sintaxis:

> ***context*** *[VariableName:] TypeName*
> ***def:*** *[VariableName] | [OperationName(ParameterName1: Type, ...)]:ReturnType = < OclExpression >*

Ejemplo:

Se define una propiedad llamada capacidad, que retorna la capacidad de un viaje dependiendo del tipo de tren utilizado en el mismo.

> ***context*** *Viaje*
> ***def:*** *capacidad : Integer = self.tren.tipoTren.capacidad*

Luego de esta definición, la propiedad capacidad es conocida dentro del contexto Vuelo.

> ***context*** *Vuelo*
> ***inv:*** *self.asientos -> select (a | a.reservado) -> size() <= self.capacidad*

4.3.1.3. Precondición

Una precondición es una restricción asociada a una operación de una clase o interface. Establece una condición que debe ser verdadera antes de ejecutar la operación. En una expresión OCL de tipo *boolean*.

El estereotipo de esta restricción es ***<<precondition>>***.

Sintaxis:

> ***context*** *Typename :: operationName(parameter1 : Type1, ...): ReturnType*
> ***pre:*** *< OclExpression >*

Ejemplo:

Antes de la ejecución de la operación ***reservar(pasajero)*** correspondiente a la clase Viaje, hay que validar que el mismo no se encuentre cancelado y que tenga algún asiendo disponible.

> ***context*** *Viaje: reservar(pasajero: Pasajero) : Boolean*
> ***pre:*** *not self.cancelado and*
> *self.asientos -> select(a | a.reservado) -> size () < self.capacidad*

La propiedad capacidad es la que se definió en el ejemplo anterior para el contexto *Viaje*.

En la expresión se pueden utilizar conectores lógicos para componer condiciones.

4.3.1.4. Postcondición

Una postcondición es una restricción que aplica a operaciones de una clase o interface. Define que condiciones deben cumplirse luego de la ejecución de la operación. Consiste en una expresión OCL de tipo boolean.

En una expresión OCL de postcondición, los elementos pueden utilizar el postfijo ***@pre*** para referenciar el valor de un elemento antes de la ejecución de la operación. También puede utilizarse una variable especial llamada ***result*** que representa el valor de retorno de la operación. Los nombres de los parámetros también pueden ser utilizados en la expresión OCL.

Esta restricción usa el estereotipo ***<<postcondition>>***.

Sintaxis:

> ***context*** *Typename:: operationName(parameter1 : Type1, ...):ReturnType*
> ***post:*** *< OclExpression >*

Ejemplo:

Se define que luego de la ejecución de la operación ***reservar(pasajero)*** de la clase Viaje, el pasajero debe estar asignado al viaje correspondiente.

> ***context*** *Vuelo : reservar(pasajero: Pasajero) : Boolean*
> ***post:*** *result = self.asientos -> exists (a | a.reservado and a.pasajero = pasajero)*

4.3.2. Expresión de valor inicial

Una expresión de valor inicial está ligada a una propiedad. La expresión debe corresponder al mismo tipo definido para la propiedad. Define el valor que tomará inicialmente una propiedad.

Sintaxis:

context *Typename :: propertyName: Type*
init: *< OclExpression >*

Ejemplo:

La propiedad ***cancelado*** de la clase Vuelo tiene como valor inicial false.

context *Viaje :: cancelado : Boolean*
init: *false*

4.3.3. Expresión de valor derivado

Una expresión de valor derivado aplica a una propiedad. La expresión de valor derivado debe respetar el tipo de la propiedad sobre la cual está definida.

Al definir una propiedad con un valor derivado, se especifica el valor de la propiedad en base a otros valores de la clase.

Sintaxis:

> ***context*** *Typename :: propertyName: Type*
> ***derive:*** *< OclExpression >*

Ejemplo:

Para la clase Viaje se define la propiedad ***tipoTren***, que puede ser derivada de las relaciones de la misma clase ***Viaje***.

> ***context*** *Viaje :: tipoTren : TipoTren*
> ***derive:*** *self.tren.tipoTren*

4.3.4. Expresión de consulta

Esta expresión se liga a una operación de consulta de una clase o interface. Debe adaptarse al tipo de la operación. Como ocurre con las precondiciones y postcondiciones, los parámetros pueden ser utilizados en la expresión.

Sintaxis:

> **context** Typename :: operationName(parameter1: Type1, . . .) : ReturnType
> **body:** < OclExpression >

Ejemplo:

Se define la operación de consulta ***getCapacidad***, para la clase ***Tren***

> **context** Tren :: getCapacidad() : Integer
> **body:** self.tipoTren.capacidad

Dentro del contexto de una operación pueden utilizarse pre y post-condiciones.

> **context** Asiento :: getPasajeroQueReservo() : Pasajero
> **pre:** self.reservado
> **body:** self.pasajero

Capitulo 5: MDA con UML

5.1. OCL en MDA

MDA, UML y OCL son iniciativas de OMG. Todas las iniciativas de OMG tienen una fuerte relación entre sí. La esencia de MDA es que los modelos son los pilares del desarrollo de software. Es por ello que los modelos deben ser buenos, sólidos, consistentes y coherentes. La utilización de UML conjuntamente con OCL permite crear modelos con las características requeridas [2].

5.2. Niveles de Madurez del Modelado

Dentro del proceso de desarrollo MDA, la principal tarea a realizar es la construcción de modelos del sistema que se está construyendo. Es muy importante producir buenos modelos [19]. Para la construcción de los mismos se utiliza UML. Sin embargo, en diferentes proyectos se utilizan los lenguajes de modelado de distintas maneras.

Para discriminar la madurez de los modelos de un proyecto, Anneke Kleppe y Jos Warmer definieron los Niveles de Madurez de Modelado (MML: Modeling Maturity Levels) [14] que son los siguientes:

- MML 0: Sin especificación
- MML1: Textual
- MML2: Texto con Modelos
- MML3: Modelos con Texto
- MML4: Modelos Precisos
- MML5: Solo Modelos

MML 0: Sin especificación

Es el nivel más bajo de la clasificación. En este nivel la especificación del software no está escrita, solamente está presente en los conocimientos de los desarrolladores.

Este nivel generalmente es usado en desarrollos de software no-profesional, entre gente que recién comienza a programar. También suelen usarlo desarrolladores que realizan un proyecto para sí mismos, donde la especificación suele cambiar continuamente en base al desarrollo que la misma persona va realizando. Cuando el desarrollador tiene una idea, la incorpora al software.

El principal problema de este nivel es que sin especificación, el software se torna rápidamente inmantenible. Cuando se agrega un nuevo desarrollador al proyecto, necesita realizar ingeniería inversa para averiguar cómo funciona y qué es lo que hace el sistema. Incluso cuando los desarrolladores originales continúen trabajando en el proyecto, tampoco pueden recordar exactamente todos los detalles del mismo y ese conocimiento se pierde.

MML 1: Textual

En MML 1 la especificación del software es en texto en lenguaje natural, escrita en uno o varios documentos. Este es el primer nivel de desarrollo profesional del software.

En este nivel los clientes pueden leer la especificación y hacer comentarios de la misma. El problema es que el texto es ambiguo por naturaleza, y puede llevar a diferentes interpretaciones y confusiones entre las diferentes personas involucradas.

Otro problema de este nivel es la actualización de la documentación. En general, al comenzar la implementación la documentación deja de actualizarse. Luego de un tiempo, la documentación queda desactualizada y se torna obsoleta.

MML 2: Texto con Modelos

En MML 2, la especificación es escrita como en el nivel anterior, pero se completa con modelos que reflejan diversas partes del sistema. Los modelos generalmente son diagramas UML de alto nivel, que representan las principales partes, componentes u objetos del sistema.

En este nivel la especificación es más sencilla de entender que en MML 1. Sin embargo, la misma sigue siendo textual y los diagramas no son muy precisos, motivo por los cuales los problemas de la ambigüedad y de la actualización siguen estando presentes.

Nivel 3: Modelos con Texto

El nivel MML 3 posee la especificación del software definida en uno o más modelos. Además de estos modelos, se utiliza lenguaje natural para explicar detalles como el contexto del proyecto, motivación de los modelos, etc. Sin embargo, los modelos son la base de la especificación.

Este es el primer nivel en el que los modelos son una buena representación de software a construir. Los modelos no son solo ayudas para el lector de la documentación, sino que representan la estructura del proyecto y son una fuente de documentación.

Por otro lado, la transición de modelos a código sigue siendo manual. Esto lleva a que con el paso del tiempo el código evolucione, pero no así los diagramas. Luego de un tiempo, los diagramas dejan de representar el sistema construido. Cuando el cliente pide algún cambio o algún bug es corregido, los cambios son implementados en el código pero no se actualizan los diagramas.

El código es el producto, y no los diagramas. Mantener los modelos actualizados no suele considerarse importante y es visto como una actividad que consume mucho tiempo, por lo que con el paso del tiempo los diagramas se tornan obsoletos.

Nivel 4: Modelos Precisos

En este nivel de madurez la especificación del software se escribe en uno o varios modelos. El lenguaje natural se utiliza para complementarlos, explicar el contexto y la motivación de los modelos, pero estos tienen un rol similar a la documentación del código fuente. Si se usan varios modelos, la relación entre los mismos está claramente definida.

Los modelos son lo suficientemente precisos para tener una relación directa con el código fuente, lo que brinda la posibilidad de generar gran parte del código automáticamente.

Los cambios en el sistema se realizan directamente sobre los modelos, y a continuación el código es regenerado. De hecho, los modelos son considerados parte del código fuente. Es por esto que es fácil mantener el código y los modelos actualizados.

Este tipo de modelos son más difíciles de generar, pero el valor agregado también es muy grande. Incluso en el caso de que no se utilice generación automática de código, la velocidad de desarrollo es muy elevada debido a que está completamente especificado que es lo que hay que implementar y como.

Nivel 5: Solo Modelos

En MML5 los modelos son lo suficientemente precisos y detallados para permitir la generación completa de código. Los generadores de código para este nivel deben ser tan confiables como son los compiladores hoy en día, de manera tal que los desarrolladores no tengan necesidad de mirar el código generado. Igual que pasa hoy en día entre los compiladores y el código assembler.

Desafortunadamente, todavía no existe un lenguaje de modelado con el que se puedan construir modelos MML 5.

5.3. Construyendo Buenos Modelos

MDA requiere modelos en el nivel de madurez 4. Este es el primer nivel en el que los modelos son más que un papel. Los modelos son los suficientemente precisos para permitir la generación automática del código, lo cual está incluido dentro del proceso de MDA.

La mejor opción en cuanto a lenguaje de modelado para construir modelos MML 4 es UML combinado con OCL. Especificar modelos con esta combinación de lenguajes permite obtener modelos de alta calidad.

Para crear modelos que sean completos, son necesarios tanto los diagramas como las expresiones OCL:

- Sin las expresiones OCL, al modelo le faltaría especificación;
- Sin los diagramas UML, las expresiones OCL no tendrían a que referirse.

Solamente la combinación de diagramas y restricciones permite una especificación completa de los sistemas, y la generación de modelos con el nivel de detalle requerido en MDA.

Capítulo 6: Refinamientos

6.1. Refinamientos en MDA

La definición de MDA no es muy específica en el concepto de refinamiento. MDA habla de modelos: PIM, PSM y otros modelos. Los refinamientos son definidos informalmente como el proceso de transformar modelos dentro de MDA, por ejemplo, de PIM a PIM, de PIM a PSM o de PSM a código.

6.2. Tipos de transformaciones

En MDA existen cuatro tipos principales de transformaciones [4]:

- PIM a PIM
- PIM a PSM
- PSM a PIM
- PSM a PSM

PIM a PIM: Transformaciones que agregan o quitan información del modelo.

Casos típicos de transformaciones PIM a PIM son:

- Pasaje del análisis al diseño
- Transformación del diseño a otro diseño más detallado.
- Abstraer algún elemento del modelo, para reducir los detalles funcionales.

Estas transformaciones no son automáticas. Requieren de la intervención de un desarrollador para que realice el proceso.

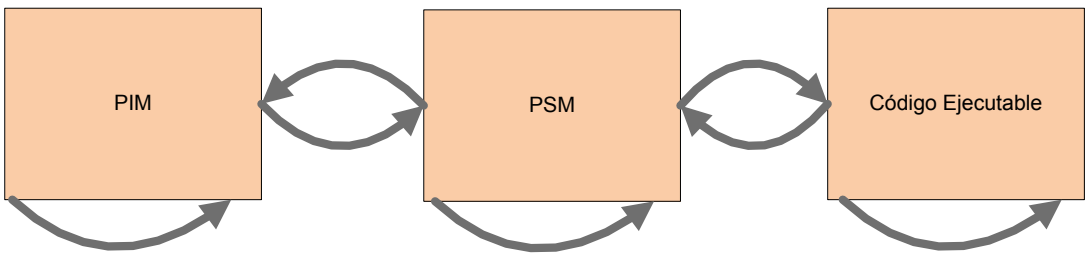
PIM a PSM: Estas transformaciones se realizan cuando los modelos PIM son lo suficientemente completos para ser implementados en una tecnología en particular.

Para realizar estas transformaciones se utilizan reglas de transformación, que pueden ser generalizadas y reutilizadas, lo que las hace altamente automatizables.

PSM a PIM: Son utilizadas sobre un PSM de una aplicación existente, para generar el PIM correspondiente. Con el estado actual de la tecnología estas transformaciones son muy difíciles de implementar.

PSM a PSM: Estas transformaciones se realizan durante las fases de desarrollo y optimización. Generalmente una sola transformación PIM a PSM no es suficiente para generar el código ejecutable óptimo, por lo que a veces es necesario transformar de PSM a PSM.

En la siguiente figura se muestran las transformaciones de modelos recién explicadas. La generación de código es considerada una transformación más.



Al especificar un sistema en MDA, se crean diferentes versiones de un mismo modelo con distintos niveles de abstracción. Dentro de un proceso de desarrollo iterativo incremental como MDA, es imprescindible asegurar que los modelos conserven consistencia mientras son refinados y cambiados. Parte del objetivo de esta tesis es contribuir en ese aspecto.

6.3. Causas de refactorización de modelos

Las causas más comunes de refactorización de modelos [3], son las siguientes:

- **Requerimientos:** Se refactoriza un PIM para adaptarlo a las necesidades del cliente. Partiendo de un PIM se genera uno nuevo con más semántica.
- **Patrones de diseño:** La utilización de patrones de diseño es una muy buena práctica. Los modelos suelen transformarse para incluir alguno de ellos.
- **Decisiones técnicas:** Decisiones tomadas por un desarrollador para cumplir con requerimientos no funcionales.
- **Requerimientos de calidad:** Necesidad de cumplir con estándares de calidad puede originar la refactorización del modelo para cambiar alguna característica.

6.4. Reglas de Refactorización

Esta tesis se centra en el caso de las transformaciones PIM a PIM. Este tipo de transformaciones se realizan desde UML a UML, es decir, se hacen refinamientos sobre un modelo UML.

Las técnicas de refactorización actuales se concentran en los diagramas, pero no tienen en cuenta las restricciones OCL asociadas. Estas pueden quedar sintácticamente incorrectas al refactorizar el diagrama de clases asociado.

El proceso de refactorización de un diagrama puede ser visto como una serie de *pasos atómicos* aplicados a partes específicas del diagrama. Bajo esta visión, es posible crear un conjunto de reglas que sirvan para formalizar los cambios que van ocurriendo en las transformaciones de diagramas.

Existen muchos catálogos de reglas de refactorización para diferentes lenguajes de programación. La refactorización sobre elementos más abstractos como los modelos UML, es una rama de investigación reciente.

Actualmente en la refactorización de diagramas UML se tienen en cuenta los cambios que ocurren a nivel de clases, atributos y métodos. Existen numerosos estudios al respecto y herramientas que facilitan los cambios y mantienen la consistencia entre diagramas. Sin embargo, ninguna técnica aplica sobre diagramas UML enriquecidos con restricciones OCL. Al aplicar las técnicas existentes sobre estos diagramas, se produce el problema que se explica en el siguiente capítulo.

Capítulo 7: El problema del código OCL en refactorizaciones

7.1. Problema del código OCL en refactorizaciones

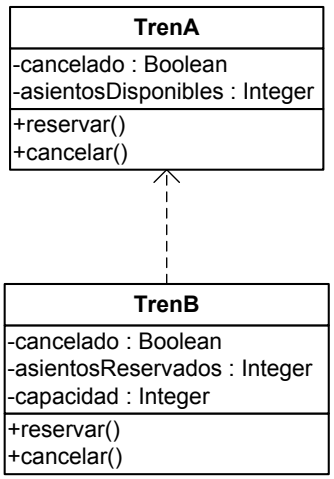
Como se explico en los capítulos anteriores, la mejor opción para la creación de los modelos en MDA es la utilización de diagramas UML complementados con restricciones OCL, para ampliar su expresividad.

Las herramientas existentes se ocupan de las refactorizaciones de diagramas UML, pero no tienen en cuenta las restricciones OCL asociadas a los mismos. Al cambiar los diagramas UML, el código OCL queda desactualizado. Es necesario que el código OCL evolucione conjuntamente con el UML para garantizar la coherencia de los diagramas.

A continuación se explica en detalle el problema que surge si solamente se actualizan los diagramas UML.

7.2. Ejemplo:

A continuación se presenta un ejemplo de una refactorización de un diagrama UML sin restricciones OCL:



El diagrama representa una clase llamada ***TrenA***. Esta posee dos propiedades, ***cancelado*** y ***asientosDisponibles***.

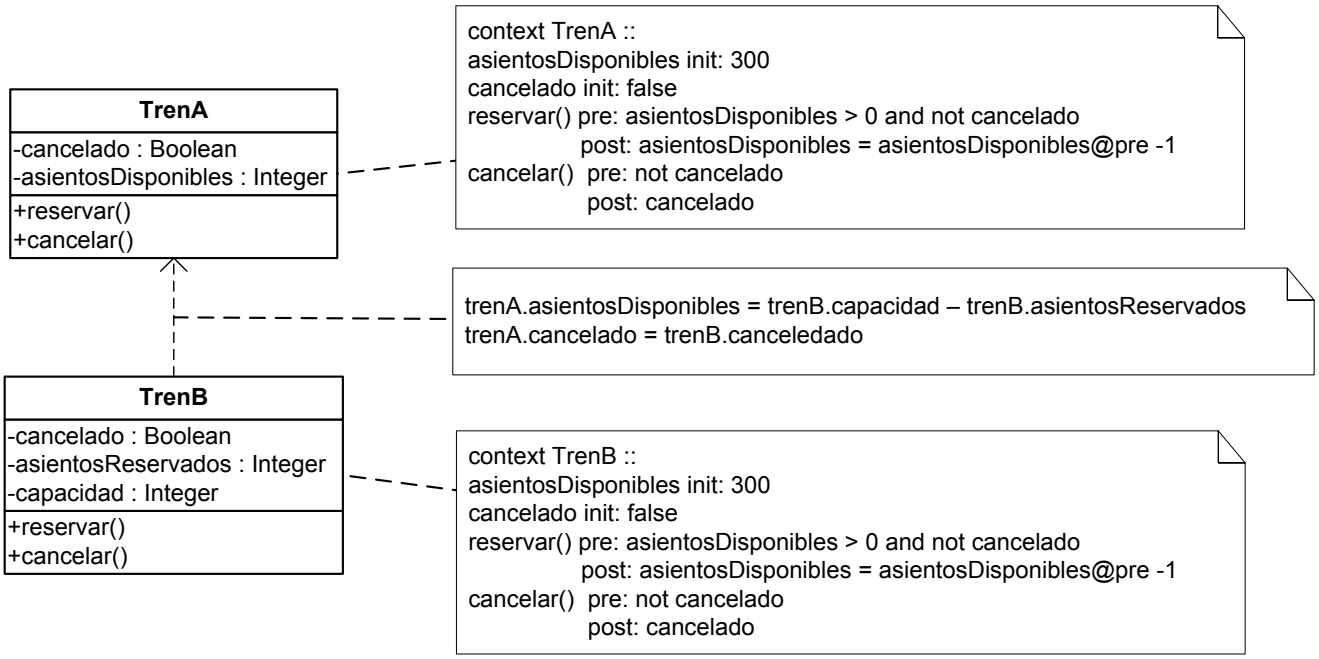
También está representada la clase ***TrenB***, que es un refinamiento de ***TrenA***. Esto se puede apreciar por la relación con el estereotipo ***<<refine>>***. El diagrama no especifica nada sobre cómo se relacionan las dos versiones de la clase.

Mirando el diagrama, puede verse que con el refinamiento la variable ***asientosDisponibles*** de la clase ***TrenA*** se transforma en dos variables de la clase ***FlightC***: ***asientosReservados*** y ***capacidad***.

Analizando más en detalle los nombres de las variables, podría inferirse que para obtener la cantidad de asientos libres en la clase ***TrenB*** hay que calcular ***capacidad – asientosReservados***. Pero esto es solamente una suposición. De haber elegido nombres distintos para las variables, este conocimiento se habría perdido por la ambigüedad propia del diagrama.

En este sencillo ejemplo puede verse como aparecen problemas en la refactorización de diagramas. En aplicaciones reales pueden surgir problemas mucho más importantes.

Para solucionar estas ambigüedades a la hora de construir diagramas y refactorizarlos, se utilizan las expresiones OCL. Continuando con el ejemplo anterior, al agregarle restricciones OCL al diagrama se obtiene la siguiente versión mejorada del mismo:



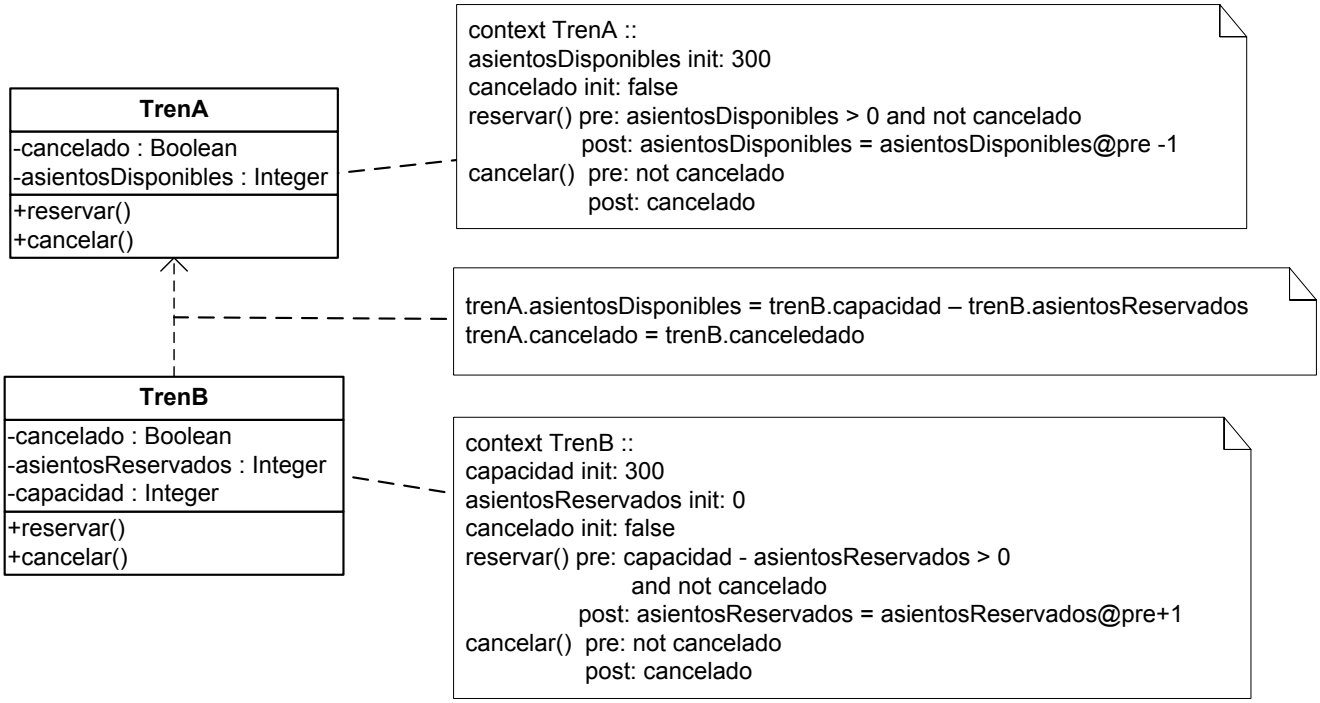
En este segundo ejemplo aparece mucha más información:

- En la clase ***TrenA,*** se especifican mediante restricciones OCL valores iniciales de las variables y las pre y post condiciones de las operaciones.
- El refinamiento aparece acompañado de información de cómo se relacionan los elementos de las clase ***TrenA*** y ***TrenB***.

El código OCL que originalmente pertenecía a la clase ***TrenA*** ha quedado desactualizado y carece de sentido para la nueva clase ***TrenB***. Si igualmente se intenta utilizar el mismo OCL sobre la clase refactorizada, ocurre que hace referencia a variables inexistentes y las pre y post condiciones no son correctas.

Esta es la aproximación que se utiliza en las herramientas que dan soporte a la refactorización de diagramas UML. Estas se ocupan de actualizar el modelo UML pero no actualizan las restricciones asociadas.

La versión correcta del diagrama es:



7.3. Solución para el problema

Como solución a este problema se propone implementar sobre una herramienta de transformación de modelos, transformaciones del código OCL asociado a los diagramas UML.

Para lograr este objetivo, se implementará una solución basada en un conjunto de reglas de refactorización. Existen muchos catálogos de reglas de refactorización para diagramas UML, pero ninguno de ellos presta atención al código OCL [18].

El nuevo catálogo de reglas representarán los cambios especificados en el mapeo de la relación de refinamiento. Es decir, a partir del mapeo del refinamiento, se instanciaran las reglas correspondientes. La aplicación de estas reglas sobre el código OCL de la clase original, dará como resultado el código OCL correcto para la clase refactorizada.

Al integrar este catalogo de reglas con una herramienta case de soporte de MDA, se avanzará un paso más hacia la automatización de la refactorización de modelos.

Capítulo 8: Catalogo de Reglas de Refactorización

8.1. Nuevo catalogo de reglas de refactorización

En este capítulo se describen las reglas de refactorización más importantes para las restricciones OCL asociadas a diagramas UML. De cada regla se analizará si modifica la correctitud sintáctica de las restricciones OCL y se describirán los cambios necesarios para las mismas en caso de ser necesario.

Cada regla de refactorización se define mediante una expresión dentro del mapeo del refinamiento del diagrama UML. En base a dicho mapeo se detecta cual regla que se está aplicando, y cuáles son los cambios a aplicar sobre el código OCL.

Las reglas a desarrollar son las siguientes:

- Renombrar Atributo
- Mover Atributo
- Refactorizar Atributos
- Transformar Atributo en Valor Calculado
- Renombrar Operación
- Mover Operación

8.1.1. Renombrar Atributo

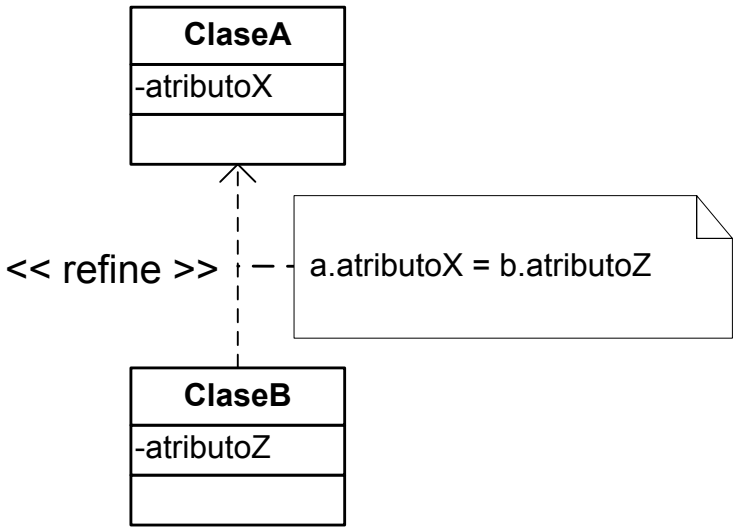
Descripción:

La refactorización cambia el nombre de un atributo. El atributo aparece en la clase refactorizada con un nombre distinto al nombre original y esta transición se describe en el mapeo.

Mapeo:

a.atributoX = b.atributoZ

Gráficamente:



Cambios OCL:

En las restricciones OCL de la nueva clase, cada ocurrencia del atributo ***atributoX***, se sustituye por ***atributoZ***.

8.1.2. Mover Atributo

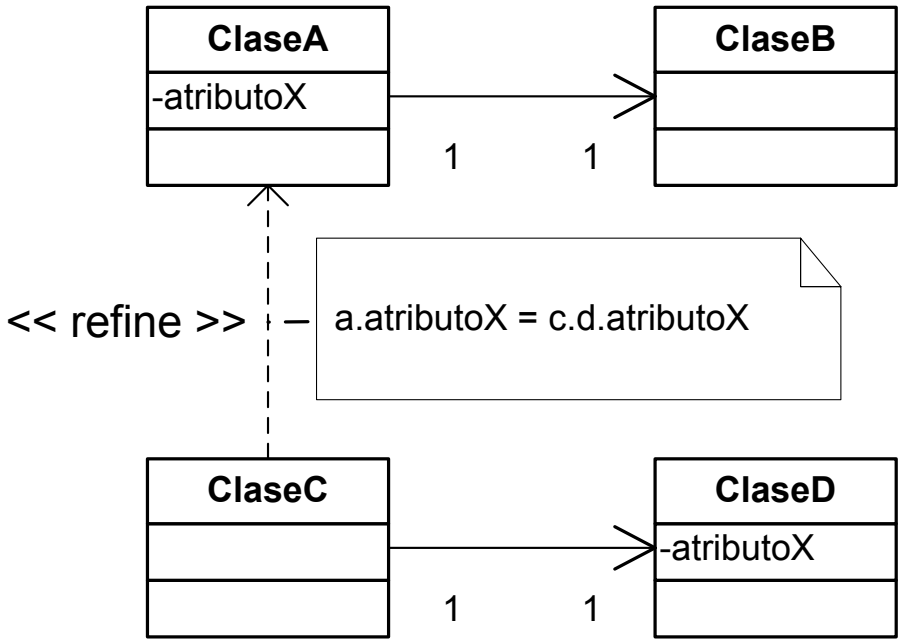
Descripción:

Al realizar la refactorización, un atributo de la clase origen para a ser parte de otra clase, la cual está unida a la nueva clase por una relación 1 a 1. La condición de que la clase debe estar unida por una relación 1 a 1 se da para garantizar que luego de la refactorización el acceso al atributo siga teniendo la misma semántica que antes de la misma.

Mapeo:

a.atributoX = b.c.atributoX

Gráficamente:



Cambios OCL:

Esta regla debe actualizar las restricciones OCL en todas las localizaciones donde se utiliza la variable movida. Los cambios de las expresiones OCL pueden ser vistos como una especie de “Navegación hacia adelante”.

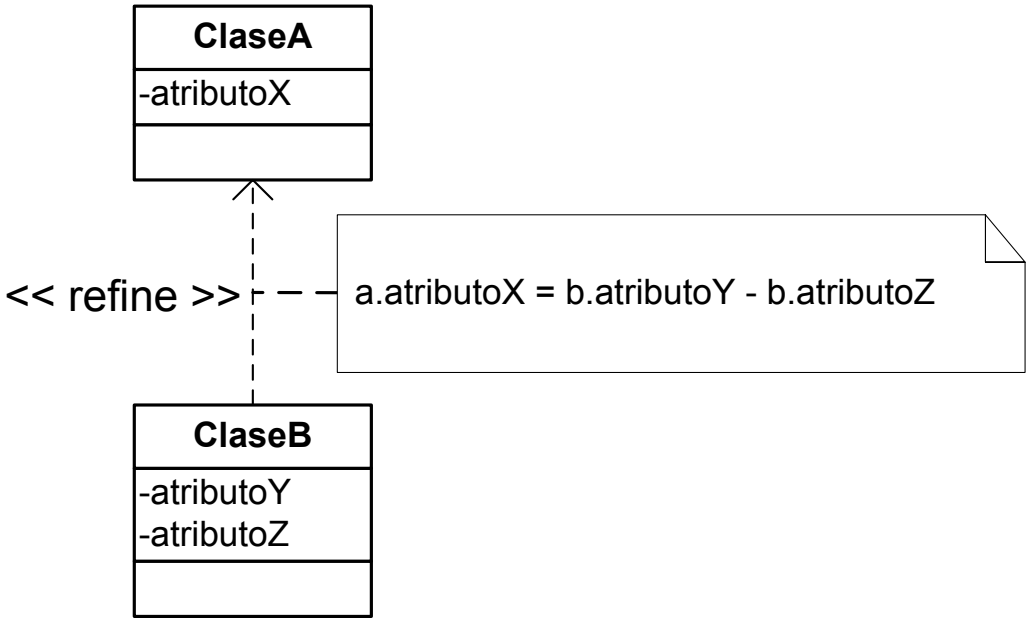
Las expresiones de la forma ***a.atributoX*** tienen que ser rescritas de la forma ***b.c.atributoX***

8.1.3. Refactorizar Atributos

Descripción:

Esta regla transforma un atributo en dos. La clase original tiene un atributo cuyo significado representa a dos de la nueva clase. Se mantiene una relación entre ambos atributos luego de la refactorización.

Gráficamente:



Mapeo:

a.atributoX = b.atributoY - b.atributoZ

Cuando la relación es una diferencia:

Cambios OCL:

Se crean los nuevos atributos ***atributoY*** y ***atributoZ***.

El atributo ***atributoX*** se elimina.

Si ***atributoX*** tenía definido un valor por defecto, este será despejado de forma tal de conocer los valores por defecto para los nuevos atributos.

En todos los lugares donde aparezca una referencia al ***atributoX***, esta será reemplazada por la relación entre los nuevos atributos.

Cuando la relación es una suma:

Mapeo:

a.atributoX = b.atributoY + b.atributoZ

Cambios OCL:

Se crean los nuevos atributos ***atributoY*** y ***atributoZ***.

El atributo ***atributoX*** se elimina.

Si ***atributoX*** tenía definido un valor por defecto, no se puede saber cómo deben ser las inicializaciones de los nuevos atributos. Es por ello que la elección de los valores iniciales se dejará abierta a la implementación de la herramienta.

En todos los lugares donde aparezca una referencia al ***atributoX***, esta será reemplazada por la nueva relación de atributos.

8.1.4. Transformar Atributo en Valor Calculado

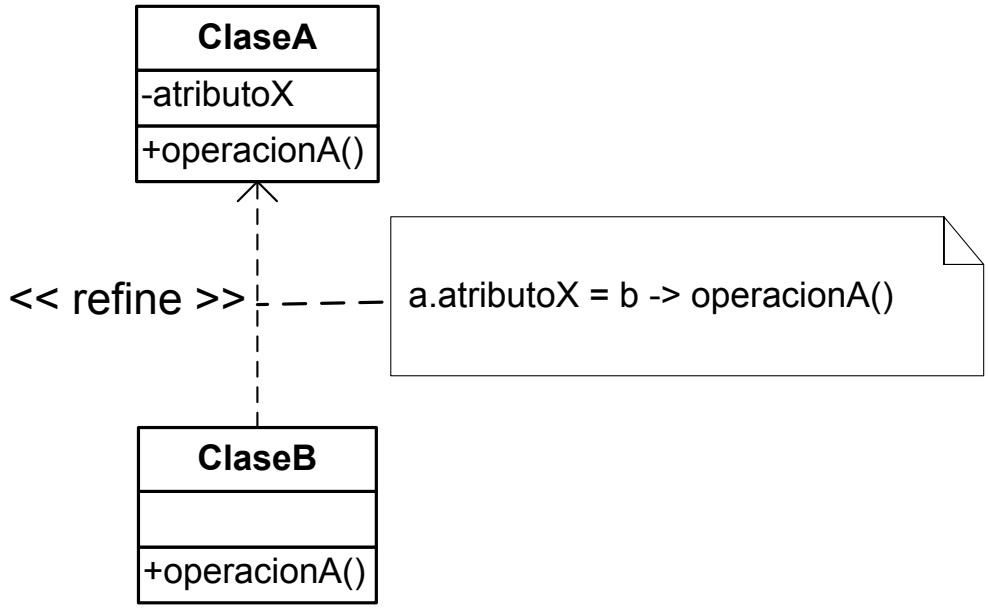
Descripción:

Con esta refactorización, un atributo se elimina y en su lugar se usa un valor calculado. En el cálculo del valor se utilizan atributos y operaciones de la nueva clase.

Mapeo:

a.atributoX = b -> operacionA()

Gráficamente:



Cambios OCL:

Esta regla debe actualizar las restricciones OCL que incluyan al atributo eliminado.

Si el atributo tenía restricciones de inicialización, estas son eliminadas del OCL de la refactorización.

En expresiones donde aparezca la variable, se debe reemplazar su uso por el cálculo especificado en el mapeo.

8.1.5. Renombrar Operación

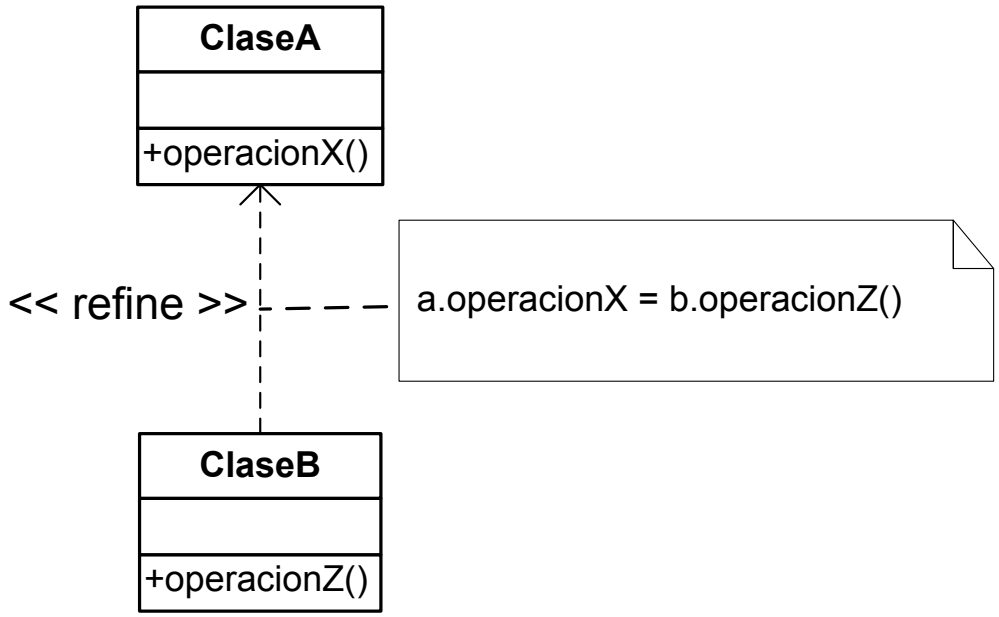
Descripción:

Con esta refactorización, una operación de la clase a refactorizar cambia su nombre al pasar a la clase refactorizada.

Mapeo:

a.operacionX() = b.operacionZ()

Gráficamente:



Cambios OCL:

Cada ocurrencia de la operación original, en el OCL refactorizado se sustituye por el nuevo nombre.

8.1.6. Mover Operación

Descripción:

Durante la refactorización, una operación es movida a una clase distinta de la propia clase que lo define.

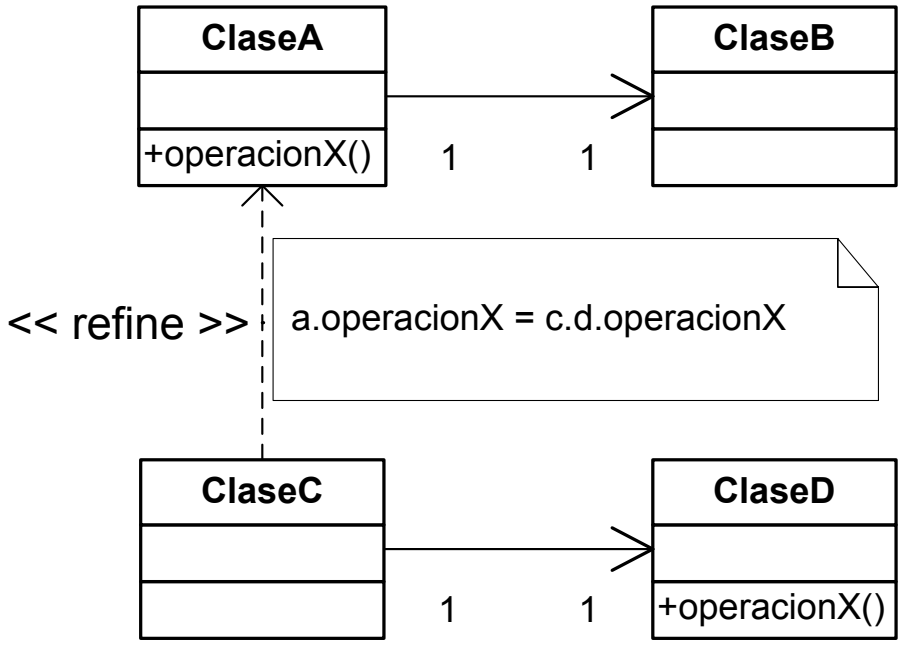
Se crea una nueva operación con el mismo comportamiento en la clase destino y se cambian todas las llamadas al método viejo por el nuevo.

Se puede cambiar el método viejo, convirtiéndolo en una delegación al nuevo método, o simplemente eliminarlo.

Mapeo:

a.operacionX = b.c.operacionX

Gráficamente:



Cambios OCL:

Los cambios en el OCL se realizan en tres pasos:

Cambiar contexto: Si la operación posee una restricción (ej: pre/post-condición), el contexto de esta restricción debe ser cambiado.

Por ejemplo desde el contexto ***a::operacionX()*** al contexto ***b::operacionX()***. En este paso, solamente se copia el contenido de la restricción, la adaptación del contenido se realiza en los próximos pasos.

Navegación hacia atrás: Después de cambiar el contexto, en la restricción de la operación movida se asume que ***self*** se refiere al tipo de la clase origen.

En el nuevo contexto, la referencia al ***self*** de la clase origen debe ser "simulada" mediante navegación desde la clase destino a la clase origen. Todas las ocurrencias de ***self.operacionX*** en la restricción movida deben ser rescritas como ***self.b.operacionX***. Esta navegación es posible por la multiplicidad 1 en el extremo de la relación correspondiente a la clase original.

Navegación hacia adelante: Las invocaciones a esta operación las tenemos que redirigir hacia la nueva operación. Esto significa sustituir todas las expresiones ***expression.operacion()*** por ***expression.b.operacion()***

Capítulo 9: ePlatero

9.1. Introducción a ePlatero

e-Platero es acrónimo de *"**E**clipse PLugin **A**iding **T**raceability in an **E**nvironment with **R**efinement-**O**rientation".* Es una herramienta CASE que soporta el diseño de software dirigido por modelos [10] [17].

Los principales objetivos de ePlatero son:

- Creación y edición de diagramas UML
- Edición y validación de restricciones OCL en el metamodelo
- Edición y validación de restricciones OCL sobre modelos
- Edición y validación de relaciones de refinamiento entre modelos
- Especificación, documentación y validación de relaciones de refinamiento entre modelos.

El objetivo agregado en la presente tesis es:

- Actualización automática de las restricciones OCL a partir de relaciones de refinamiento.

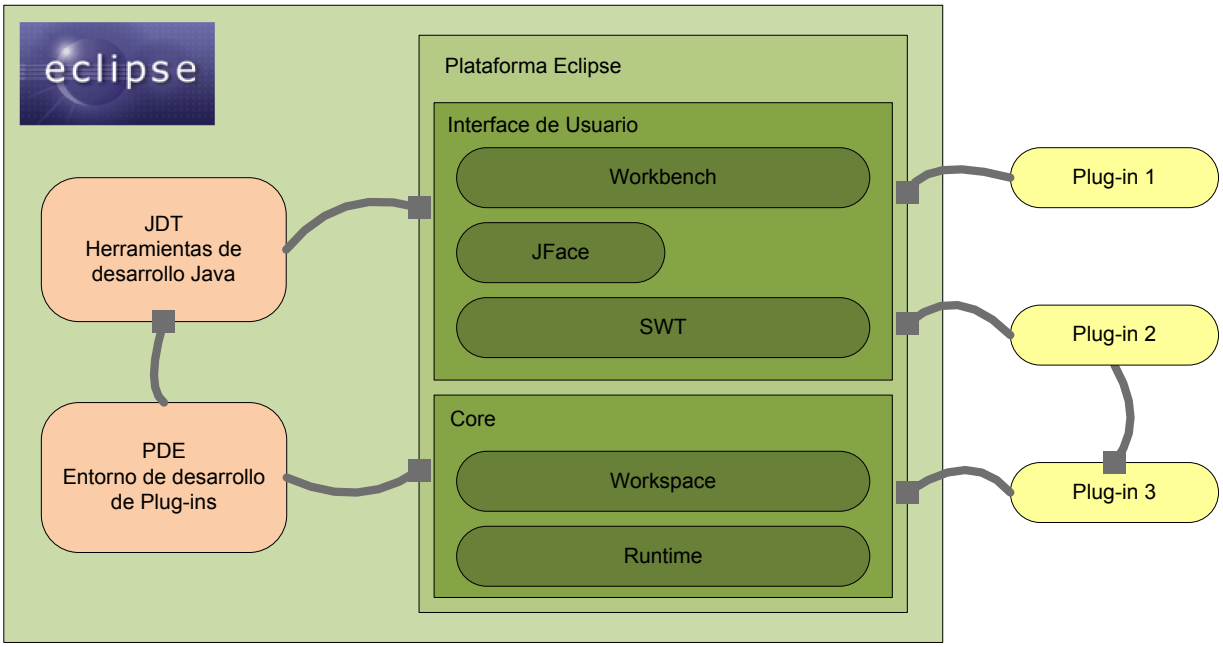
ePlatero está construido sobre la plataforma Eclipse [9], una plataforma de desarrollo integrada (IDE - Integrated Development Environment) que provee un conjunto básico de funcionalidades, las cuales sirven de soporte para la incorporación de otras herramientas. Estas herramientas son componentes desacoplados que reciben el nombre de plugins.

El mecanismo básico por el cual se extiende Eclipse es incorporando nuevos plugins que se integran con las funcionalidades de la plataforma e interactúan con otros plugins, muchos de los cuales vienen por defecto con las distribuciones de Eclipse.

La plataforma Eclipse fue diseñada para la construcción de IDEs. Sin embargo, sus funcionalidades básicas y los mecanismos de ampliación mediante plugins, la convierten en la plataforma indicada para el desarrollo de aplicaciones extensibles, con la posibilidad de integrar componentes de distintos fabricantes.

9.2. Arquitectura de Eclipse

La arquitectura básica de Eclipse se muestra en el siguiente gráfico.



La plataforma Eclipse posee un pequeño micro kernel. Exceptuando el kernel, todo el resto está escrito como un plugin. Cada plugin es cargado por el micro kernel.

Eclipse posee puntos de extensión específicos, los cuales pueden ser utilizados por los desarrolladores para extender la plataforma.

El núcleo de Eclipse es responsable de:

- Iniciar y correr la plataforma
- Declaración y manejo de plugins
- Manejo de recursos del sistema

La interface de Eclipse está compuesta por tres grandes grupos: workbench, JFace y SWT.

Desde la perspectiva del usuario que utiliza Eclipse, el workbench es la ventana principal de la aplicación. Está compuesta por vistas, barras de herramientas, menús y editores. El propósito del workbench es facilitar la integración de las herramientas. Estas herramientas se acoplan al workbench mediante los puntos de integración.

JFace es un pequeño conjunto de frameworks para el soporte en la creación de interfaces. Sirve para simplificar tareas comunes de desarrollo. Está diseñado para trabajar en conjunto con SWT.

SWT define los widgets que conforman la herramienta. Provee acceso de manera fácil y eficiente a las características del sistema operativo sobre el que corre Eclipse.

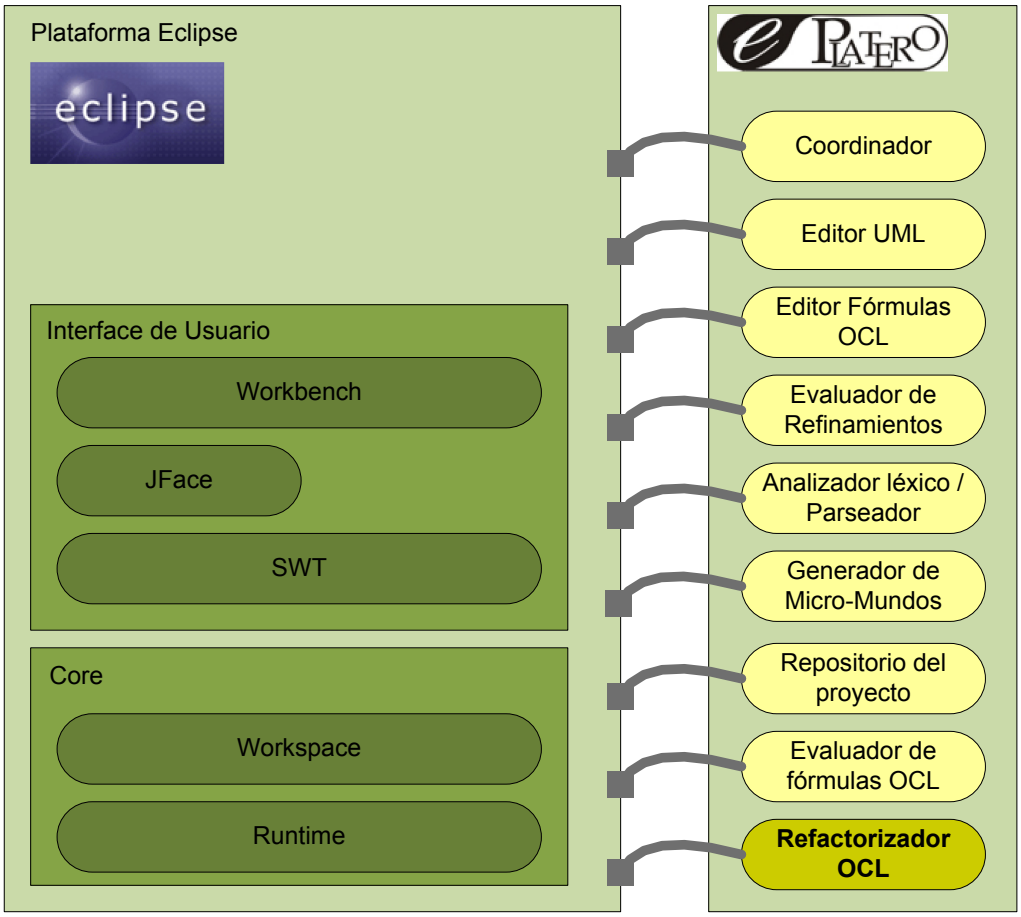
9.3. Arquitectura de ePlatero

ePlatero se adapta al estándar de Eclipse. Está formado por 8 plugins básicos, más el desarrollado en la presente tesis.

Los plugins que conforman ePlatero son:

- Analizador léxico / Parseador
- Repositorio del proyecto
- Coordinador
- Evaluador de fórmulas OCL
- Editor de fórmulas OCL
- Evaluador de refinamientos
- Generador de Micro-Mundos
- Editor UML
- **Refactorizador OCL**

En el siguiente diagrama se muestra gráficamente la estructura de componentes de la arquitectura de e-Platero:



Capítulo 10: Módulo de refactorizaciones OCL para ePlatero

10.1. Funcionamiento general del Plug-in

Para la implementación de la solución al problema planteado, se decidió incorporar un nuevo modulo (plugin) a la herramienta EPlatero. Este nuevo módulo es el encargado de refactorizar las restricciones OCL correspondientes a diagramas UML refactorizados.

El módulo es independiente de otros plugins o funcionalidades de la herramienta, pero necesita de los mismos para su funcionamiento.

Para poder aplicar las refactorizaciones de código OCL, es necesaria la creación de un ambiente de trabajo, utilizando las funcionalidades existentes de ePlatero.

El primer paso es la creación de un diagrama de clases UML. El mismo debe contener una refactorización de clases, especificada por una relación de refinamiento.

En el refinamiento se especificarán las reglas que posibilitan esa transformación de UML. Estas reglas serán utilizadas también para transformar el código OCL. El diagrama UML debe tener asociado un archivo de restricciones OCL, correspondientes al diagrama original.

Las restricciones aplican solamente al diagrama original, por lo cual, las diferencias de estructura o semántica introducidas en el refinamiento hacen que el archivo OCL no concuerde con la refactorización. El modulo está pensado para incorporarse al proceso de desarrollo en este punto.

Para realizar la refactorización del código OCL, se debe seleccionar del menú contextual del archivo OCL la opción ***"Refactor -> Apply rules"***. Con esto se accede a la funcionalidad del plugin.

El módulo dispone de una pantalla especial en la que se podrá seleccionar uno de los diagramas UML cargados en el workspace de EPlatero. Al seleccionar un diagrama, el plugin lo procesa y genera la representación del mismo en el metamodelo de UML.

Luego, a partir del archivo OCL elegido se instancian las restricciones del mismo en el metamodelo de OCL. Para esto se utiliza el OCLParser.

A continuación, se extraen del metamodelo UML los mapeos del refinamiento. Estos mapeos UML se analizan y transforman en un conjunto de reglas de refactorización para el código OCL.

Las reglas generadas se ejecutan sobre el metamodelo de OCL. Esto se realiza mediante un ***visitor*** que recorre la instanciación del metamodelo OCL. Para cada uno de los elementos del mismo se ejecutan las reglas mapeadas.

El orden de ejecución de las reglas se determina con un algoritmo encapsulado mediante el patrón ***strategy***.

Durante el procesamiento se los mapeos, se configura cada regla con información del elemento OCL al que aplica. Al ejecutarse cada regla sobre un elemento OCL, esta decide si debe o no ejecutarse sobre el mismo. Cada regla sólo actuara sobre una porción determinada del OCL.

Como resultado del procesamiento de las reglas, una nueva instanciación del metamodelo OCL es creada. Este nuevo conjunto de reglas aplican al diagrama UML refactorizado.

El último paso es realizar el proceso inverso al parseo inicial, es decir, se procesa el metamodelo OCL para generar el nuevo archivo de restricciones OCL asociado al diagrama.

10.2. Módulos de ePlatero

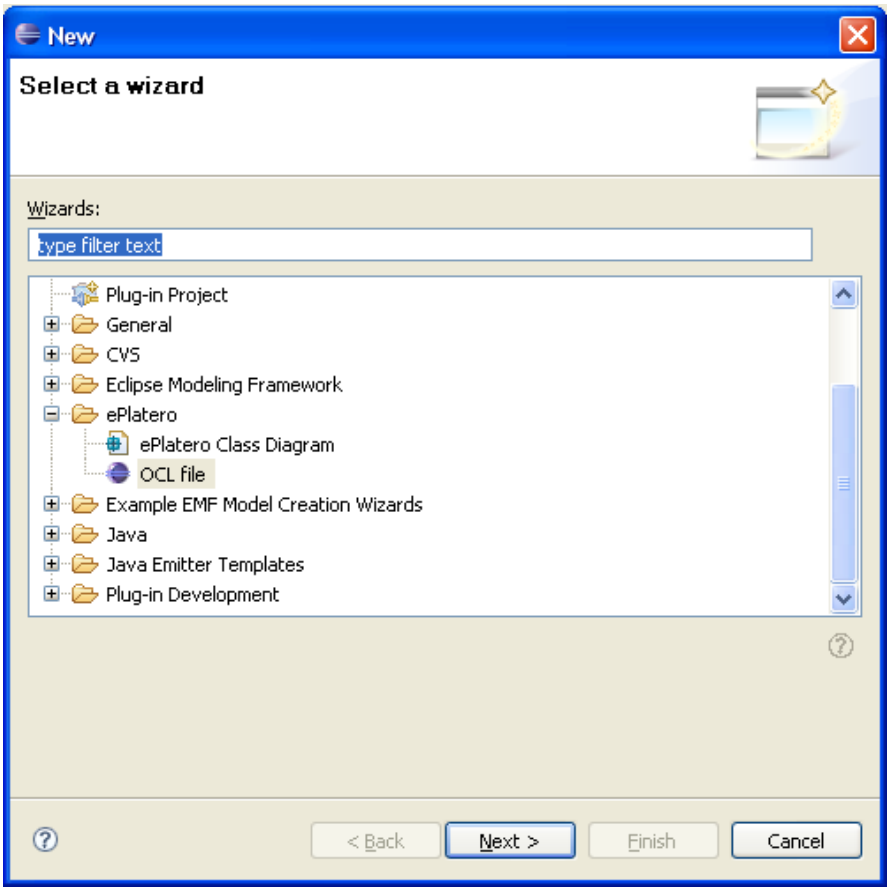
Para la implementación de la solución plateada, se utilizaron componentes de ePlatero. Se creó un nuevo plugin, que se agrego a los ya existentes. Luego se adaptaron las funcionalidades ya existentes para darle soporte a las nuevas funcionalidades.

A continuación se explica brevemente el funcionamiento de ePlatero y su integración con el nuevo plugin, para luego se explicar en detalle las extensiones planteadas en el presente trabajo.

10.3. Editor gráfico de modelos UML

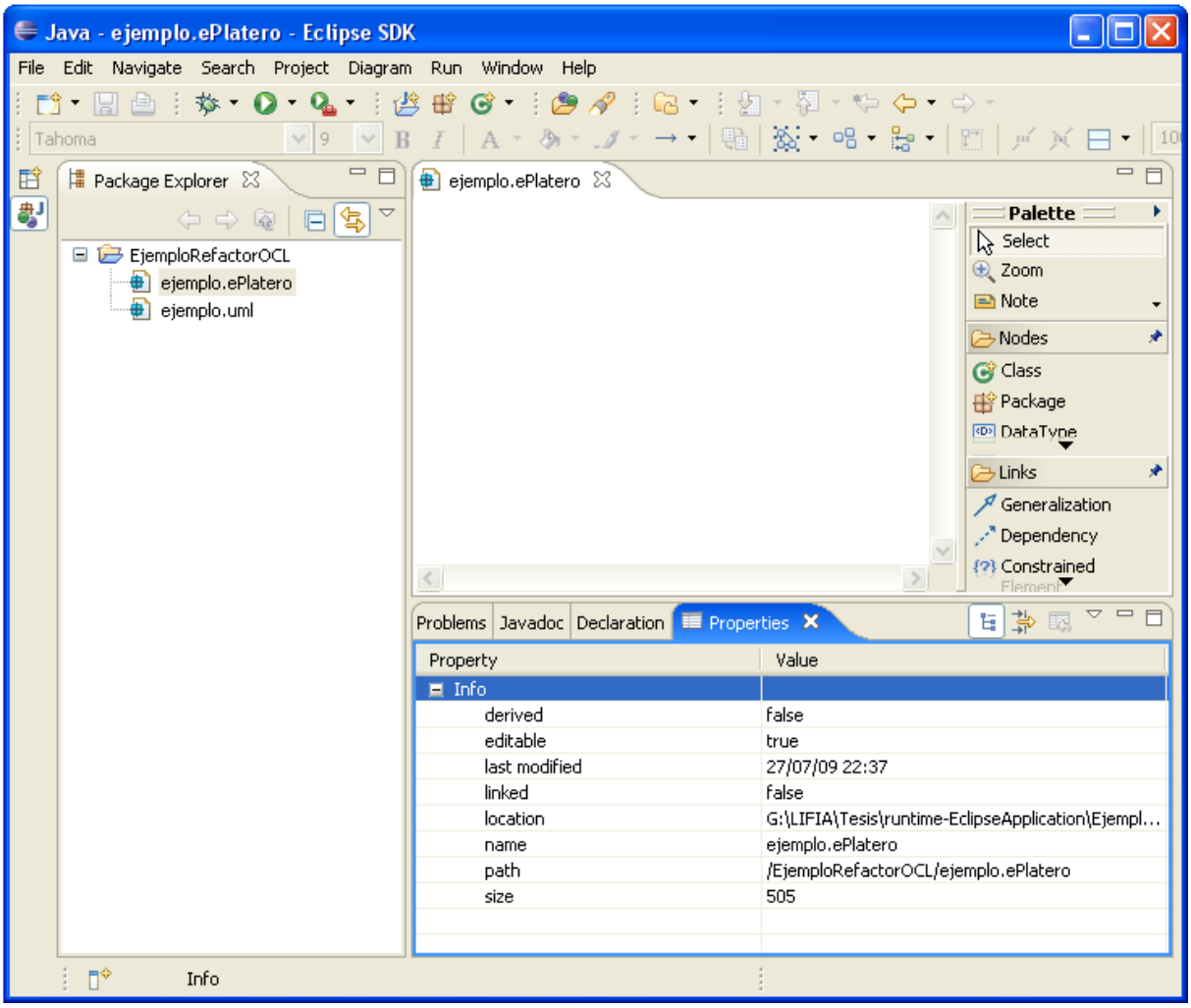
El editor gráfico de modelos UML está implementado sobre los plugins *GMF*, *EMF* y *UML2*. Un modelo ePlatero está dividido en dos partes, cada una de las cuales se guarda en un archivo separado. Por un lado se almacena un diagrama de clases UML, representado por las instancias en el metamodelo de UML2, y por otro lado se almacena un archivo con los elementos que conforman el diagrama. El primero tiene extensión .uml mientras que el segundo .eplatero.

A continuación se muestra la interface para la creación de un modelo ePlatero. Para acceder a la misma, desde un proyecto del workspace hay que ir a la opción *File>>New>> Other>>Model Wizard*. Luego hay que buscar la categoría ePlatero, bajo la cual se encuentran los wizards de la herramienta.



Las opciones relacionadas con esta tesis son: ***"ePlatero Class Diagram"***, que permite crear modelos **UML**, y **"OCL File"** que permite definir archivos OCL para especificar las restricciones sobre los modelos.

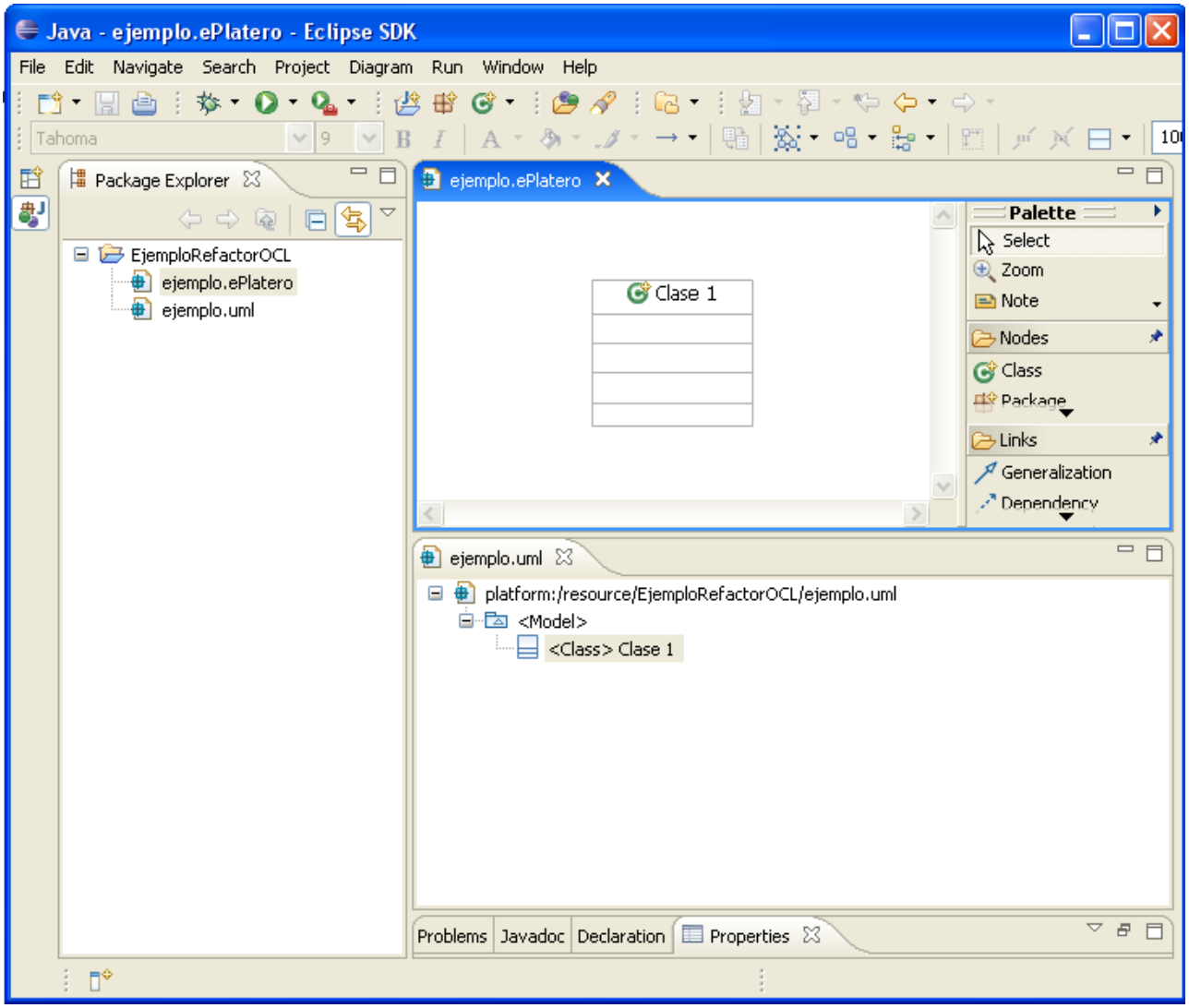
En la figura siguiente se muestra la interface del editor de diagramas de clases UML de ePlatero.



A la izquierda de la figura puede verse el **"Package Explorer"**. En él puede apreciarse un proyecto llamado *"EjemploRefactorOCL"*, y dentro de dicho proyecto, un modelo ePlatero llamado ejemplo.

Puede verse como se crearon los dos archivos mencionados anteriormente, el *.uml* y el *.eplatero*. Estos dos archivos se mantienen constantemente sincronizados. Los cambios efectuados en cualquiera de ellos, son trasladados al otro por la herramienta misma.

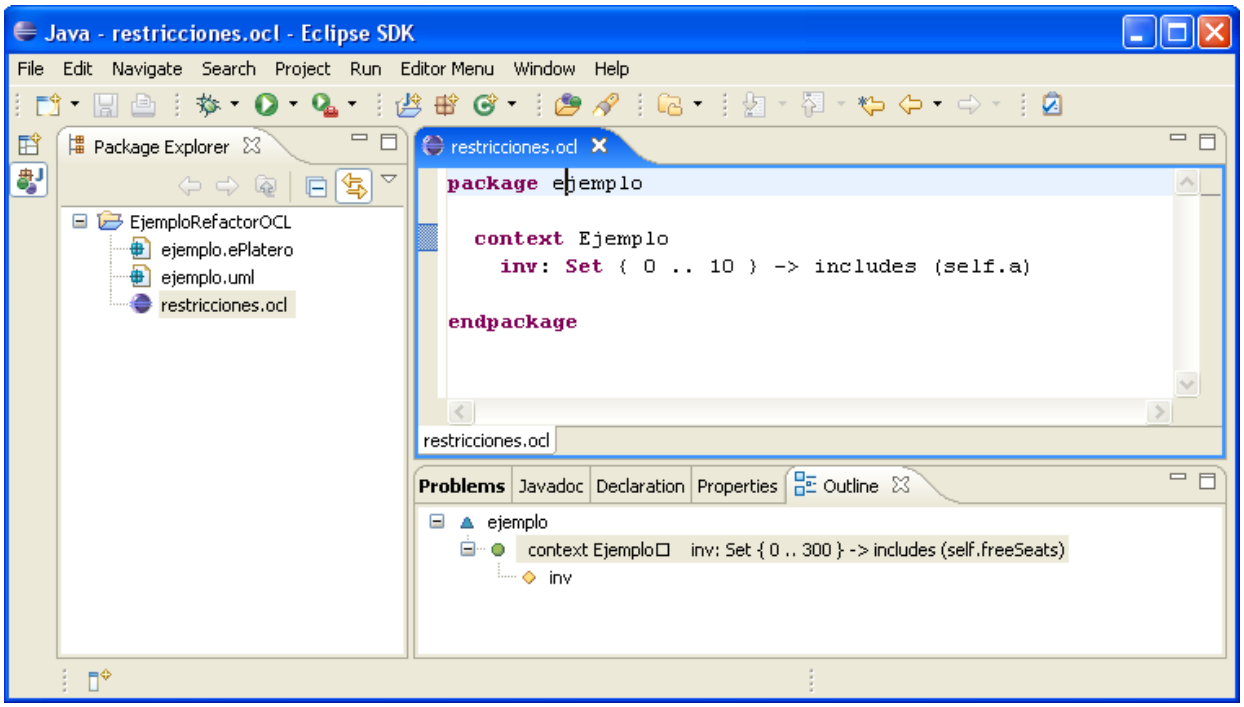
La siguiente figura muestra la vista gráfica del archivo .eplatero (a la derecha arriba) y el editor EMF que muestra el metamodelo UML del archivo *.uml* (a la derecha abajo).



10.4. Editor de restricciones OCL

El editor de fórmulas OCL es utilizado para definir restricciones OCL. El editor de ePlatero permite la edición de propiedades, sintaxis highlighting, content assitence, y corrección de errores. También dispone de una vista “outline” en la cual puede verse la estructura del archivo OCL.

A continuación se muestra el editor de formulas OCL:

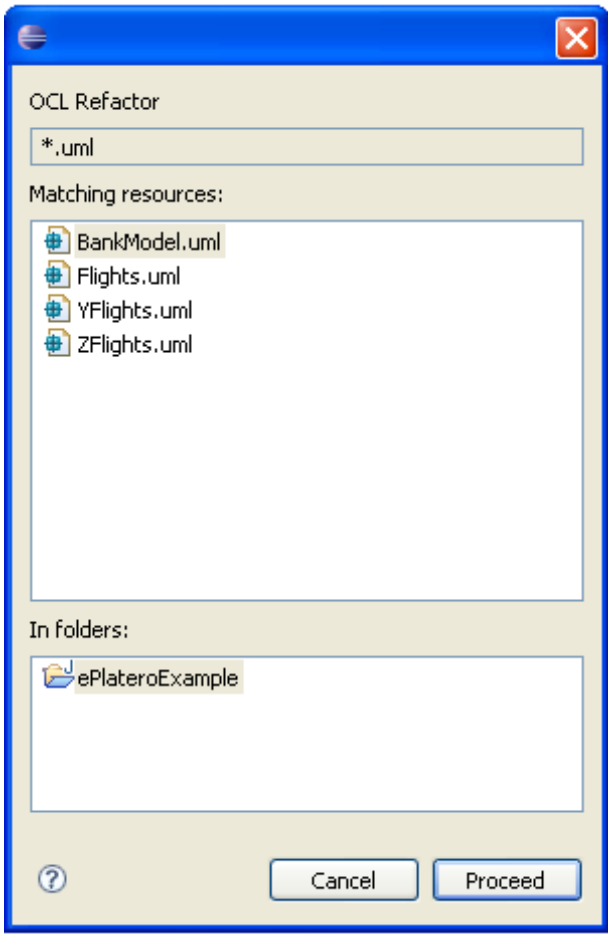


Puede verse como en la parte superior derecha de la imagen está el editor de restricciones OCL, con las características mencionadas. Por debajo, se encuentra la vista *Outline* en la cual se ve gráficamente la representación de las restricciones.

10.5. Refactorizador OCL

Al seleccionar un archivo de restricciones OCL en el “Package Explorer”, con el botón derecho y seleccionando la opción “Refactor -> Apply Rules” se accede al plugin de refactorizaciones de código OCL. Este posee una ventana de selección de los diagramas UML: una especialización del tipo ResourceListSelectionDialog, que es uno de los tantos diálogos que provee Eclipse. El mismo fue subclasificado y se le agrego el comportamiento para permitir realizar las selecciones necesarias en las refactorizaciones de OCL.

A continuación se muestra la interface de la ventana:



En la parte superior de la ventana se listan los diferentes diagramas UML del workspace actual. Los diagramas allí listados, corresponden a los que están definidos dentro del proyecto seleccionado en la parte inferior. El propósito de esta ventana es relacionar un archivo de restricciones .ocl a refactorizar con su modelo UML correspondiente.

El punto de acceso a la ventana recién presentada es el menú contextual de ePlatero. Para realizar la conexión, fue necesario configurar el archivo plugin.xml del coordinador, especificando que los archivos .ocl tengan disponible la opción de ejecutar el refactorizador de restricciones.

Se definió la clase RefactorSelectRulesAction para la coordinación del plugin. Esta clase es el punto de integración entre el EPlatero y el plugin de refactorizaciones OCL. Su tarea consiste en permitir y validar el ingreso de información por parte del usuario, lo cual se realiza con la ventana de selección recientemente explicada.

El siguiente paso es el de instanciar el metamodelo UML correspondiente al diagrama seleccionado. Por último se realiza el llamado al módulo de refactorizaciones, cuyo punto de entrada es el OCLRefactorFacade.

10.6. Facade OCLRefactor.

El punto de entrada a las funcionalidades de refactorización de código OCL es el facade OCLRefactorFacade, el cual implementa el patrón de diseño *facade*.

Se utilizó este patrón de diseño para proveer una interface simplificada de la funcionalidad. Con esto se logra una mejor comprensión y más fácil utilización de la funcionalidad implementada.

El OCLRefactorFacade también implementa otro patrón de diseño, el singleton. Esto se realizó para impedir la instanciación del mismo desde otras partes de ePlatero.

El método principal del OCLRefactorFacade es el “refactor”. Recibe como parámetro el archivo .ocl sobre el cual se realizará la refactorización. Una vez que comienza la ejecución, el facade es el encargado de la coordinación de las acciones a realizar por el plugin.

Lo primero que realiza es procesar el metamodelo UML instanciado en el paso anterior. Recupera del mismo el tipo abstracto, el concreto y el mapeo de la refactorización.

A partir del mapeo de la refactorización, se infieren las reglas a utilizar para transformar el código OCL.

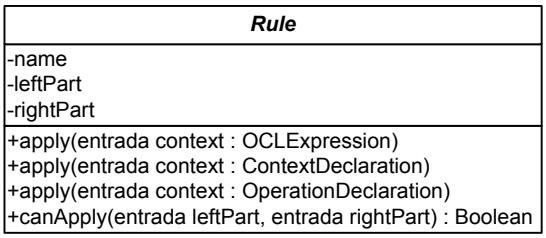
Finalmente se procesa el archivo OCL, del que se genera la instanciación del metamodelo.

10.7. Reglas de refactorización

Para realizar las transformaciones de las restricciones OCL se utiliza un conjunto de reglas de transformación. Las reglas inician su ciclo de vida como mapeos en el diagrama UML. En particular, forman el mapeo de la refactorización.

Al ejecutar el refactorizador de restricciones OCL son procesadas y transformadas en un conjunto de instancias de las clases que las representan para luego ejecutarse sobre una porción determinada del metamodelo de OCL.

Las reglas de refactorización implementadas corresponden a las reglas presentadas en el capítulo 8. Para agrupar las funcionalidades comunes a todas las reglas, se creó la interface llamada Rule. El diagrama se muestra a continuación:



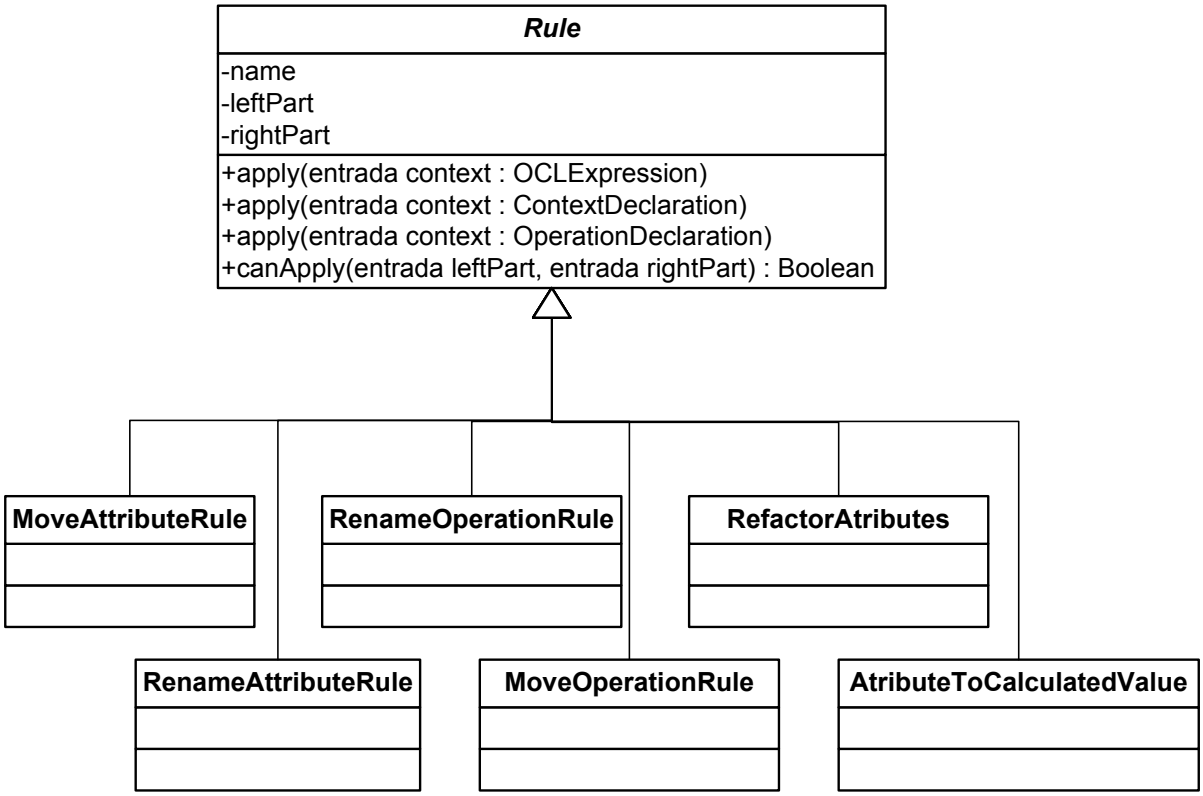
En el diagrama puede verse que los métodos a implementar por las reglas son canApply y apply.

El primero de los métodos se utiliza durante la fase de procesamiento del mapeo de refinamiento del diagrama UML. Por cada las regla mapeada en el diagrama UML, se le pregunta a la clase de su correspondiente representación en Java, si el mapeo se corresponde con su tipo. En caso de ser positiva la respuesta, se crea una nueva instancia de la regla con los datos contenidos en el mapeo.

AL finalizar este proceso, queda instanciado un conjunto de reglas a ejecutar sobre las restricciones OCL. Puede haber más de una regla de cada tipo, cada uno tendrá diferentes propiedades de acuerdo al mapeo.

Los métodos *apply* son tres, y cada uno aplica a diferentes elementos del metamodelo de OCL. Cada regla debe implementar todos o algunos de estos métodos con el código que implemente las transformaciones del metamodelo OCL.

Las clases que implementan cada una de las reglas del catalogo son las siguientes:



10.8. Ejecución de las reglas de transformaciones (visitor)

Cuando se procesa el archivo de restricciones OCL, se crea una representación en objetos del mismo a través de una instanciación del metamodelo OCL. Sobre esta instanciación se realiza el procesamiento de las reglas.

El encargado de recorrer el metamodelo es un objeto que implementa el patrón de diseño *visitor*. Su función es la de recorrer la estructura de objetos instanciada del metamodelo OCL y ejecutar las reglas explicadas en el punto anterior sobre cada uno de los componentes del mismo.

Como Java es un lenguaje que no dispone de multiple dispatch, fue necesario realizar una implementación que utilice doble dispatching para ejecutar cada una de las reglas sobre la instancia particular de la regla OCL.

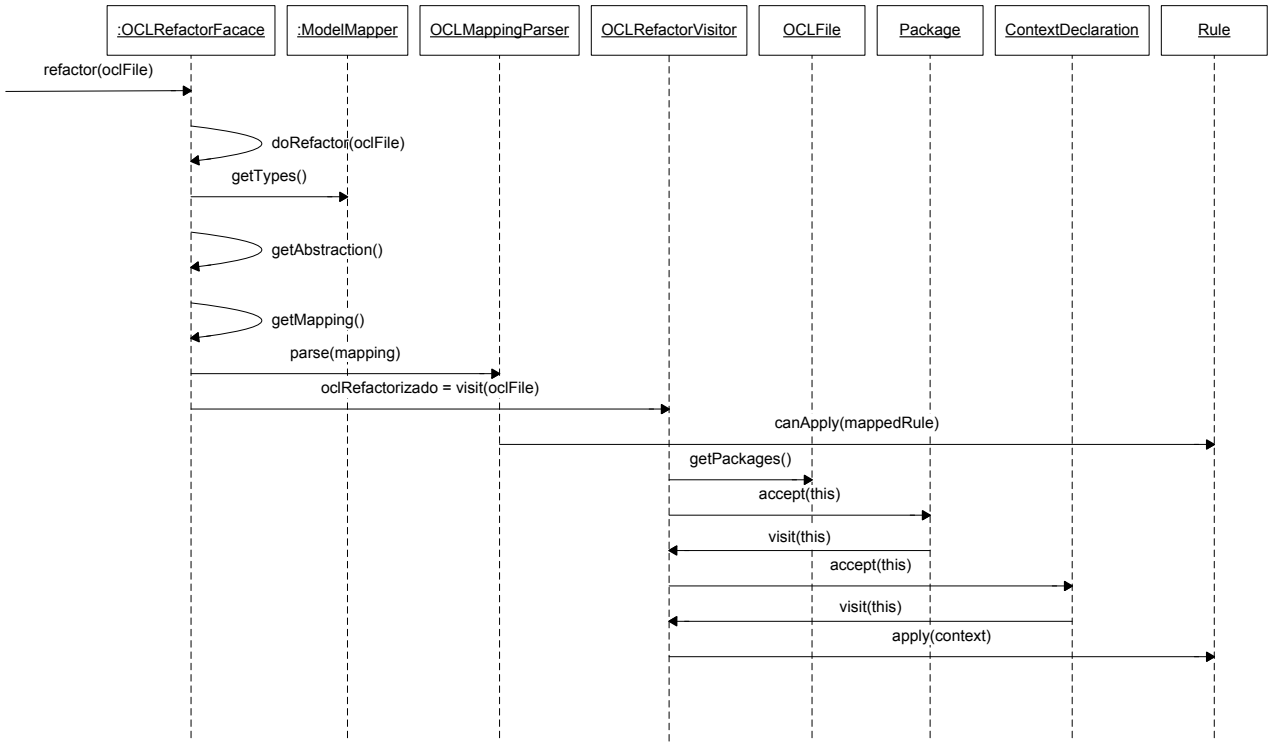
La ejecución de las reglas se realiza en un orden determinado. Este orden es especificado con otro patrón, en este caso el *strategy*. Este se encarga de proveer el orden con el que se van a ejecutar las reglas sobre cada elemento OCL particular.

La ejecución de las reglas trae como resultado la modificación de la instanciación del metamodelo OCL. Al finalizar la ejecución de las reglas, el grafo de objetos queda modificado. Las reglas OCL allí representadas son consistentes con respecto al diagrama UML refactorizado, tal como era el objetivo del plugin.

Finalmente, el metamodelo modificado se utiliza para realizar el proceso inverso. El mismo es recorrido y su información es utilizada para modificar el archivo de restricciones OCL original, con las restricciones consistentes con el diagrama.

10.9 Diagrama de secuencia

A continuación se muestra de forma simplificada, la secuencia de ejecución del plugin:

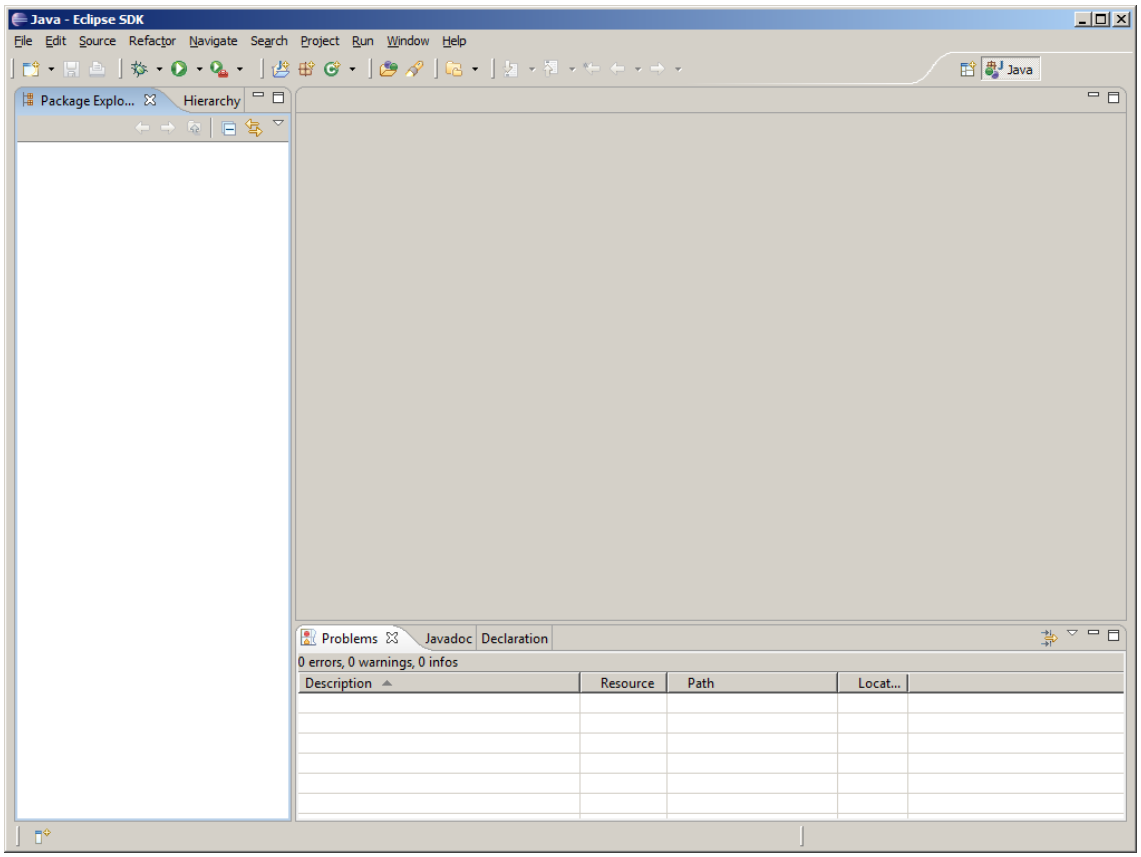


Capítulo 11: Ejemplos de uso del plugin.

11.1 Demostración de funcionamiento de ePlatero

En este capítulo se mostrará el funcionamiento general de la herramienta ePlatero. Se comenzará con las funcionalidades necesarias para la ejecución de los casos de ejemplo propios de esta tesis, y se finalizará con la demostración de la ejecución de diferentes ejemplos.

Al iniciar ePlatero por primera vez en un workspace vacio, el mismo luce como en la imagen siguiente



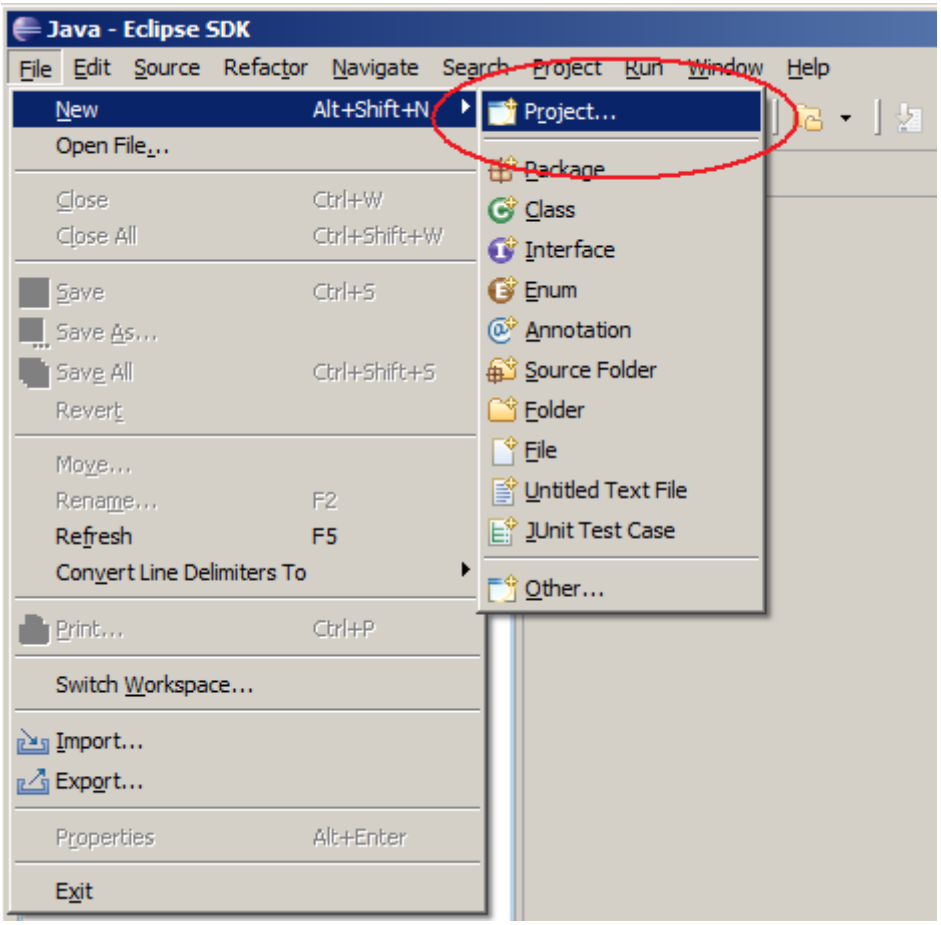
A la izquierda puede verse el explorador de paquetes. Esta vista es la que muestra los proyectos del workspace y los archivos de los mismos. Como en el

ejemplo se muestra un nuevo workspace, todavía no existe ningún proyecto a visualizar.

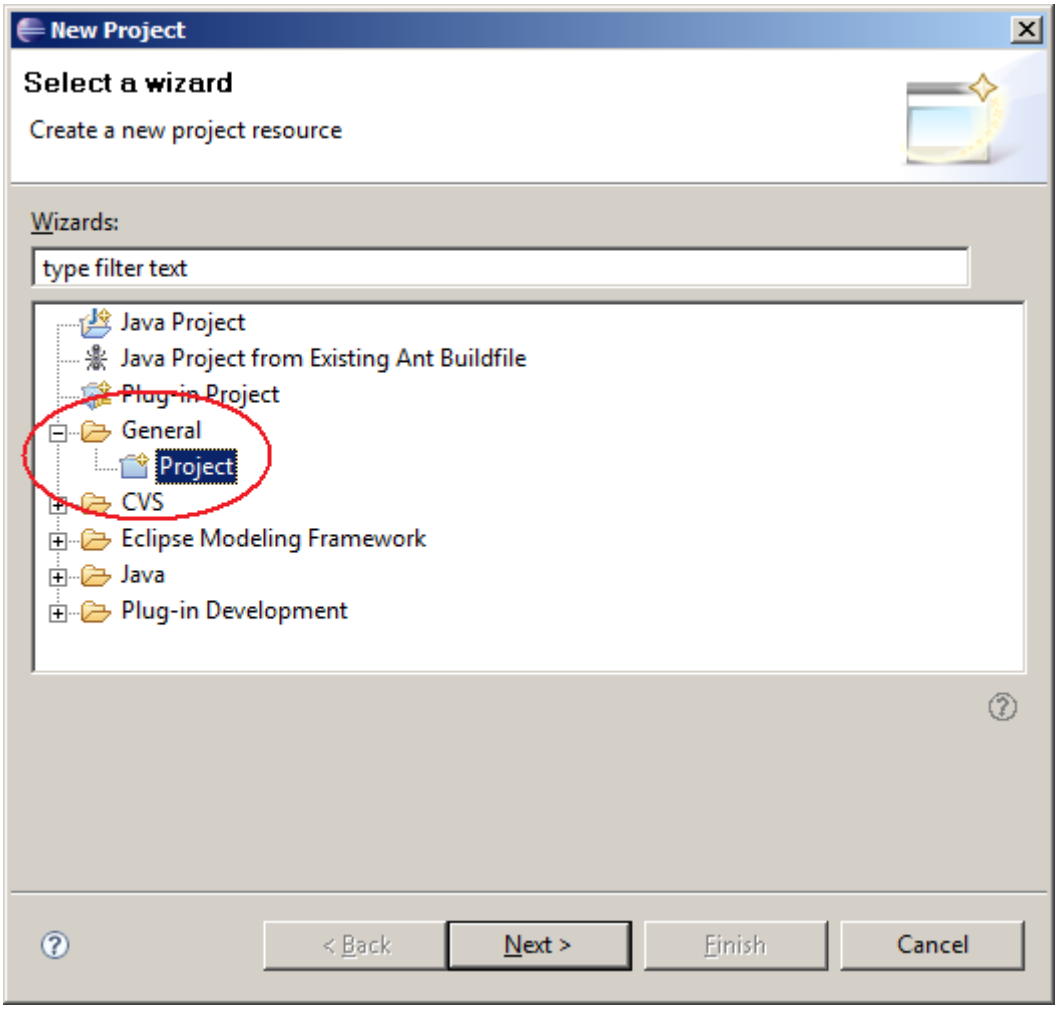
En el centro se encuentra el área de trabajo. En esta área se abrirán los diferentes editores que posee ePlatero.

Sobre la parte inferior, se encuentran diferentes vistas con información complementaria suministrada por los diferentes editores.

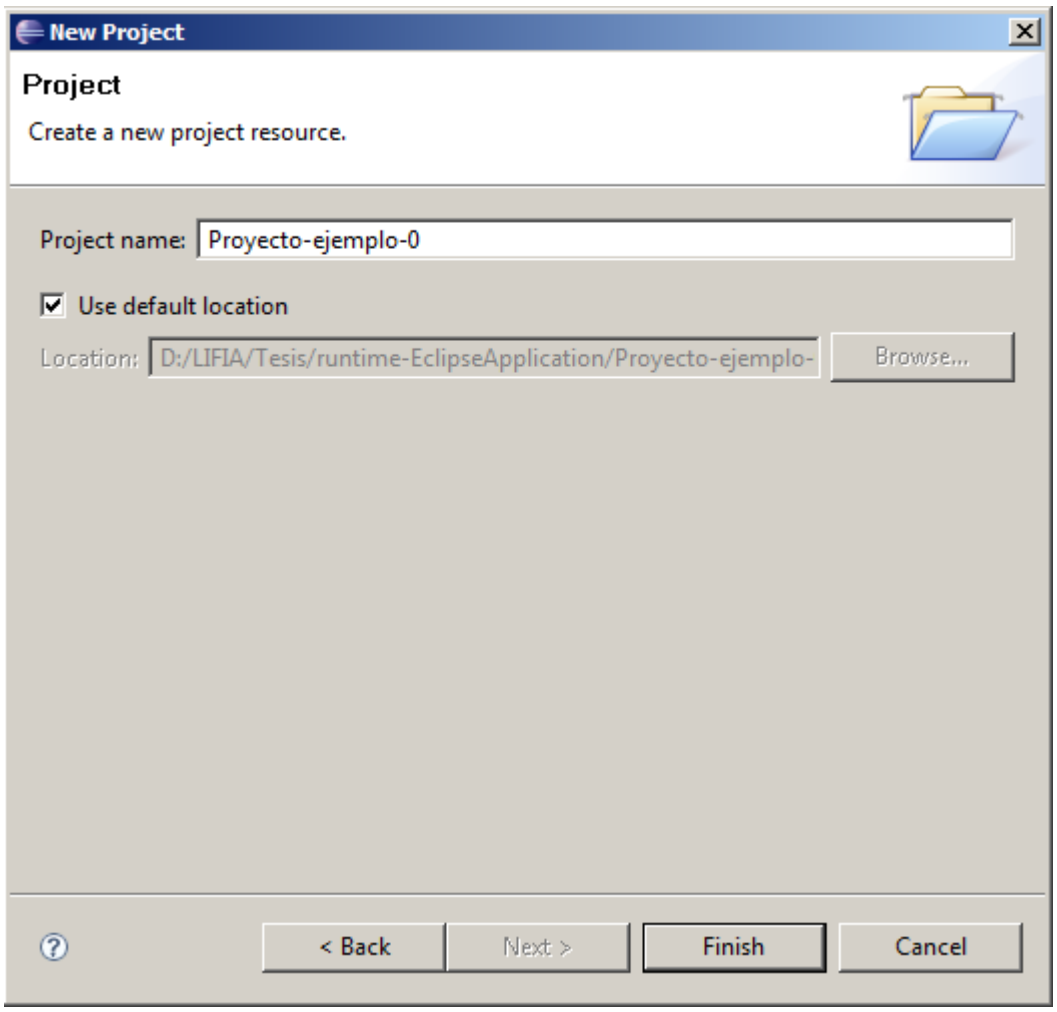
Para comenzar a trabajar con los editores de ePlatero, es necesaria la creación de un proyecto. Para ello, hay que utilizar el menú **File>New>Project**



Esto brinda acceso al diálogo de nuevo proyecto. El tipo de proyecto necesario para la edición de modelos en ePlatero es **General>Project**

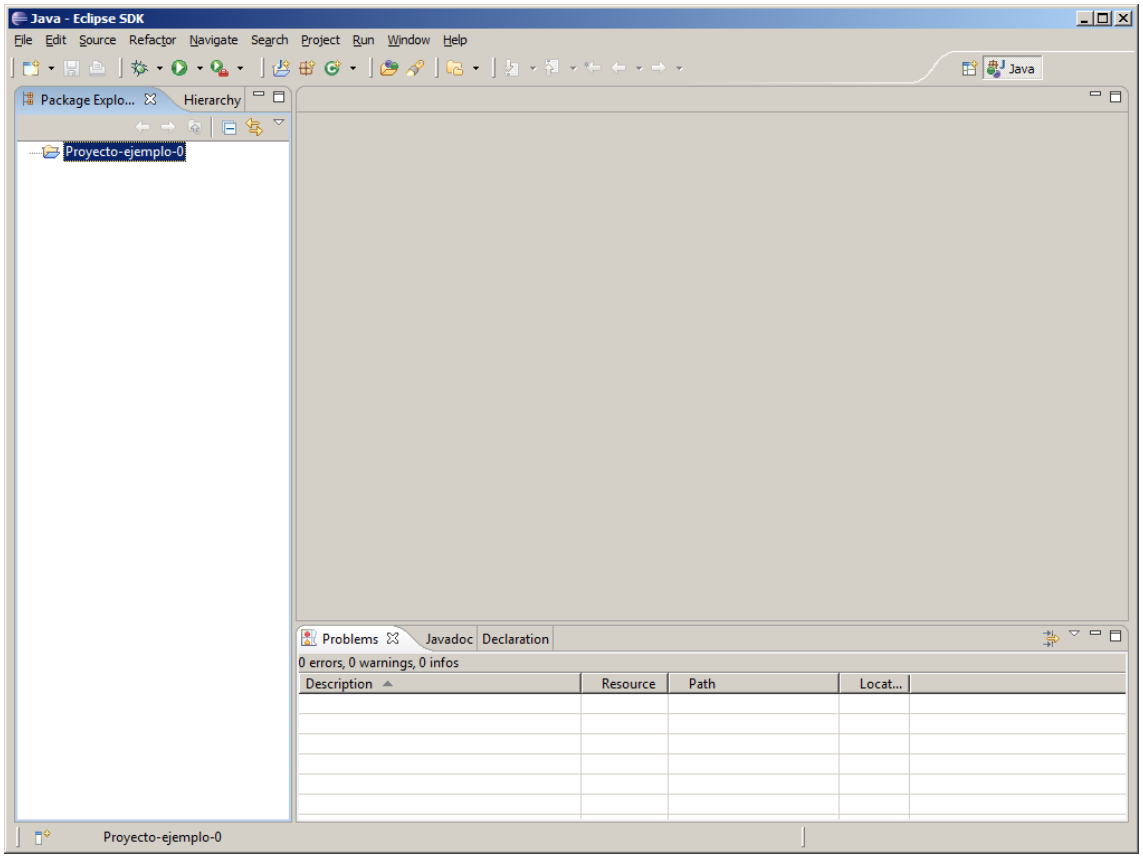


A continuación, se mostrará una ventana en la que se deberá ingresar el nombre del nuevo proyecto, en este ejemplo *Proyecto-ejemplo-0*, tal como se muestra en la imagen siguiente.



Al presionar *Finish* el proyecto será creado.

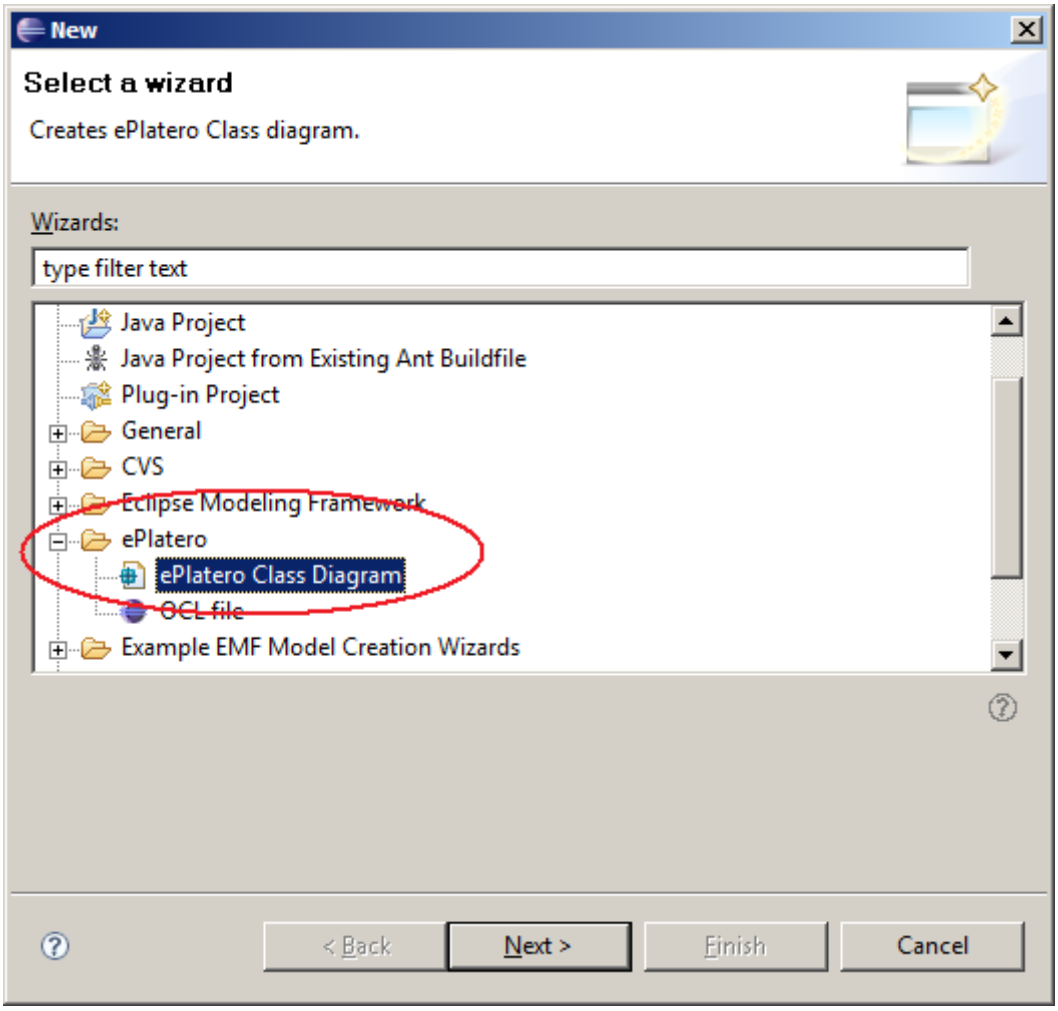
En la imagen siguiente se muestra ePlatero con el nuevo proyecto creado.



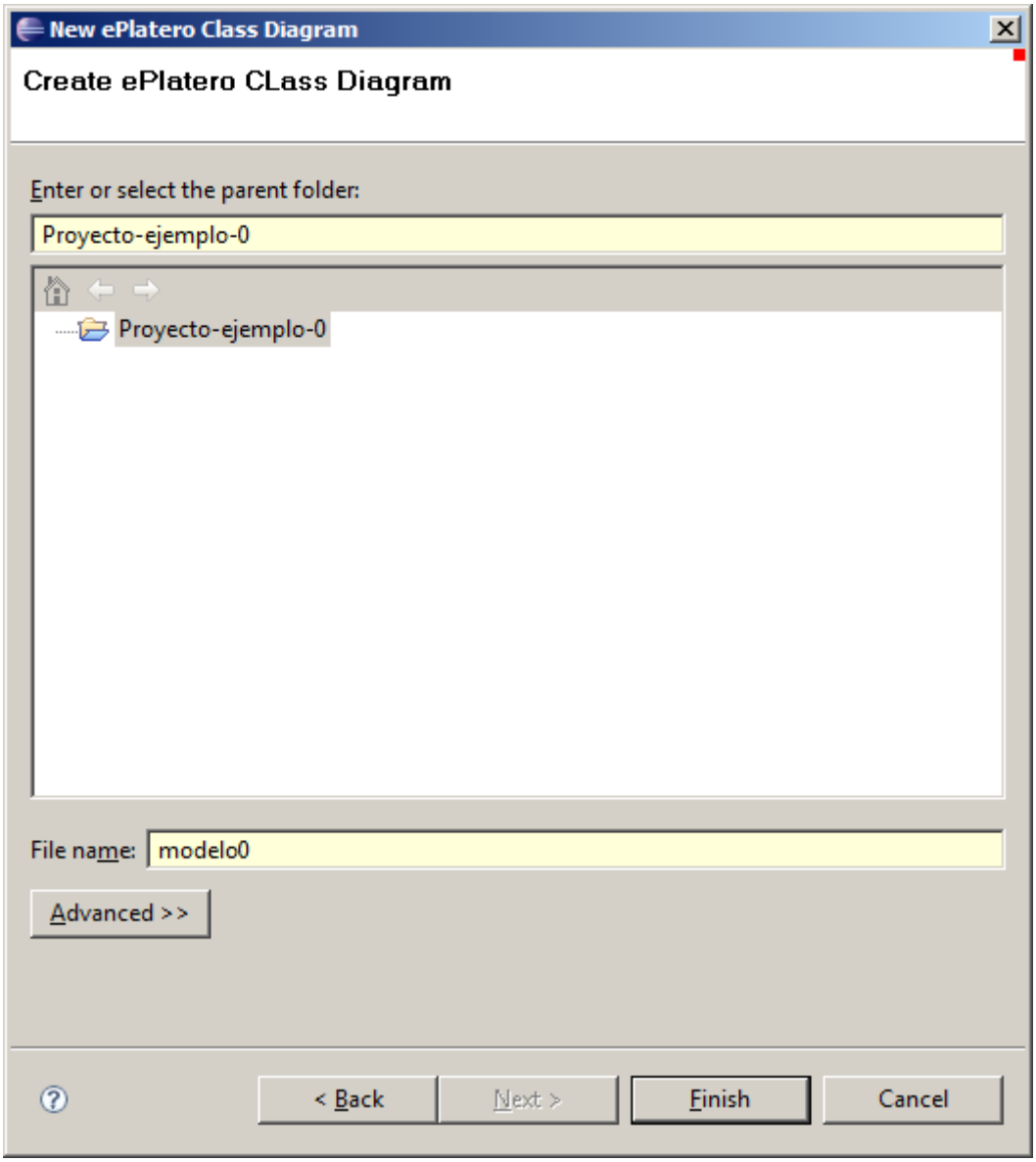
Puede verse como el nuevo proyecto se encuentra dentro del **Package Explorer**.

El próximo paso es la creación de los modelos UML y archivos de restricciones OCL sobre los cuales se aplicarán las refactorizaciones.

Para crear un modelo UML, hay que hacer click derecho sobre el proyecto e ingresar a la opción **New>Other…** Luego se selecciona la opción **ePlatero>ePlatero Class Diagram,** como se muestra en la siguiente imagen.

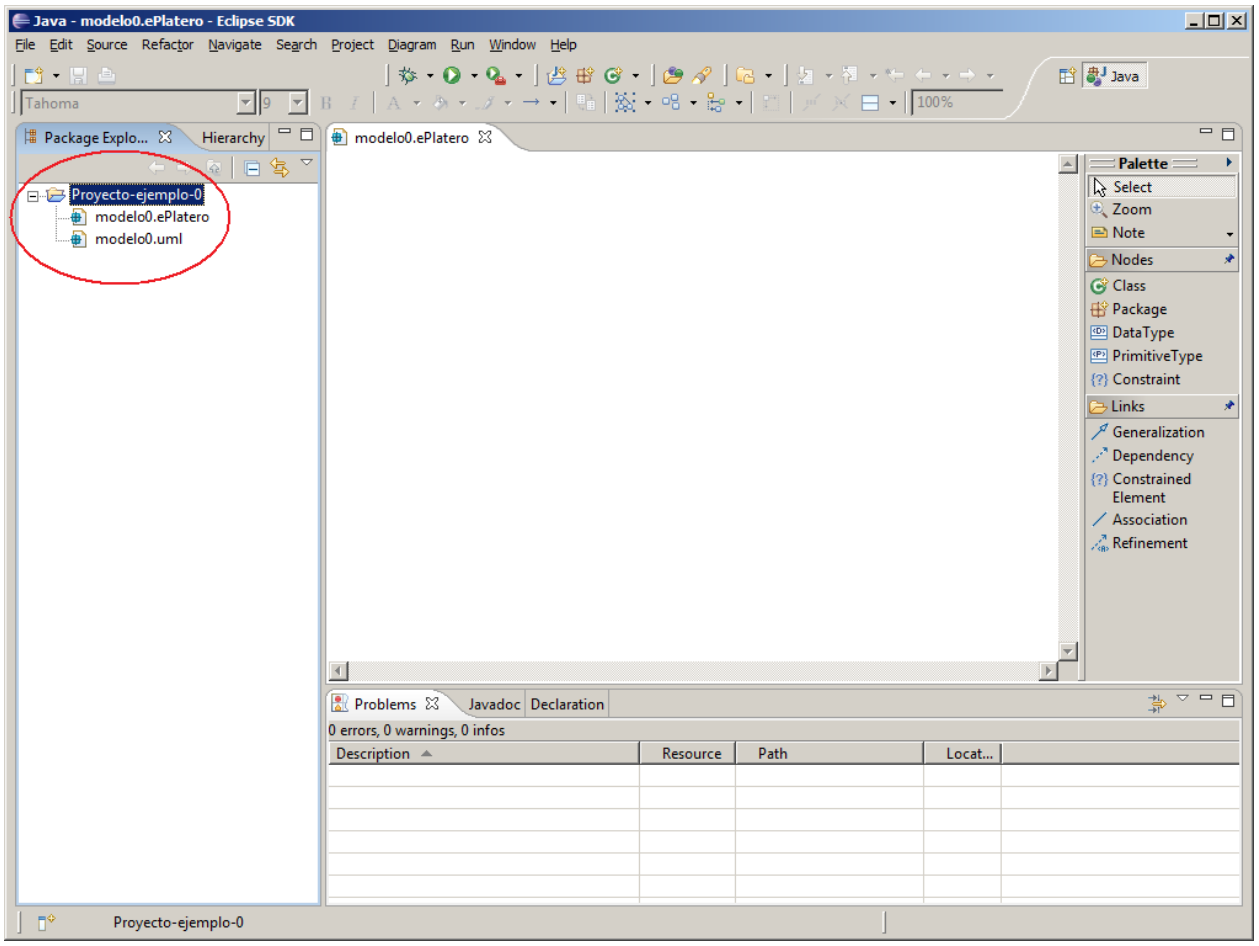


En el dialogo siguiente, se debe seleccionar el proyecto en el cual crear el archivo y el nombre del mismo.

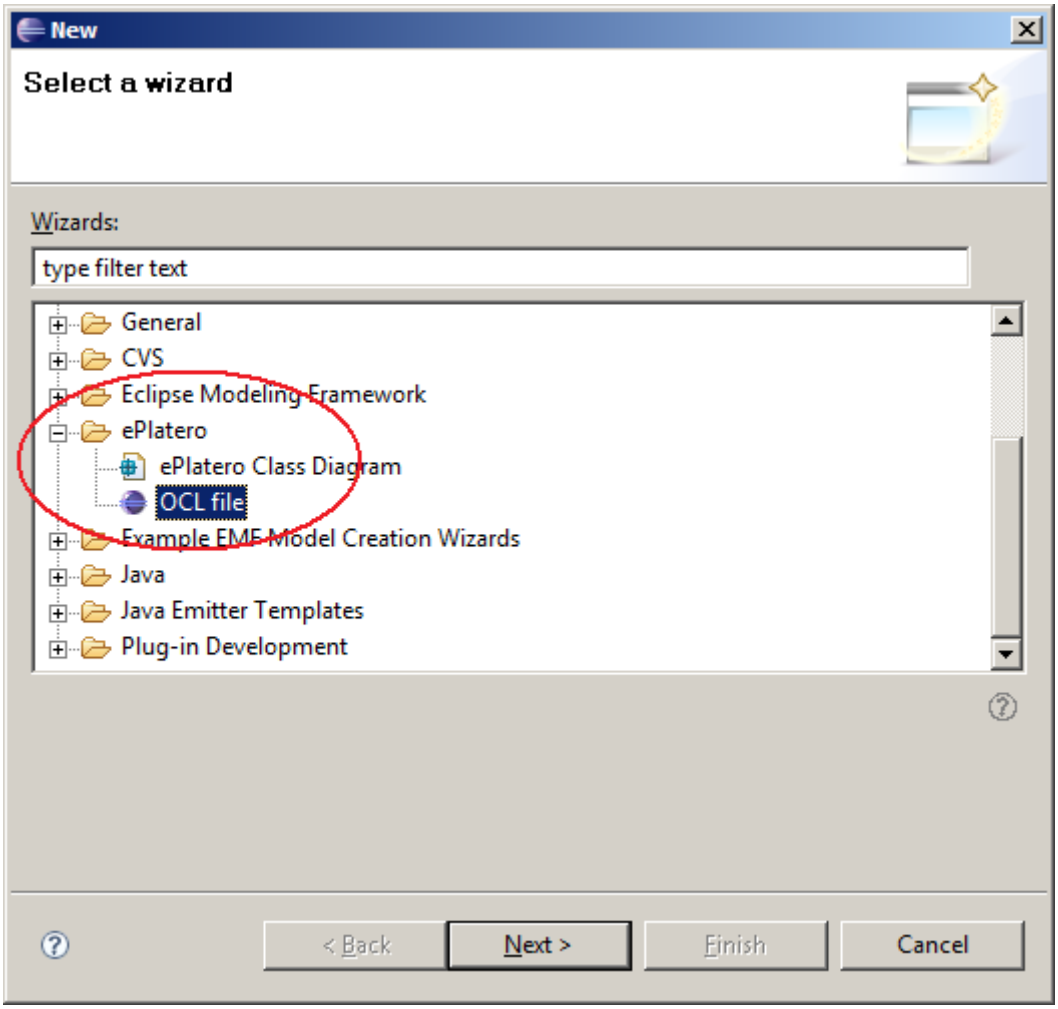


Al presionar **Finish**, dos archivos son creados:

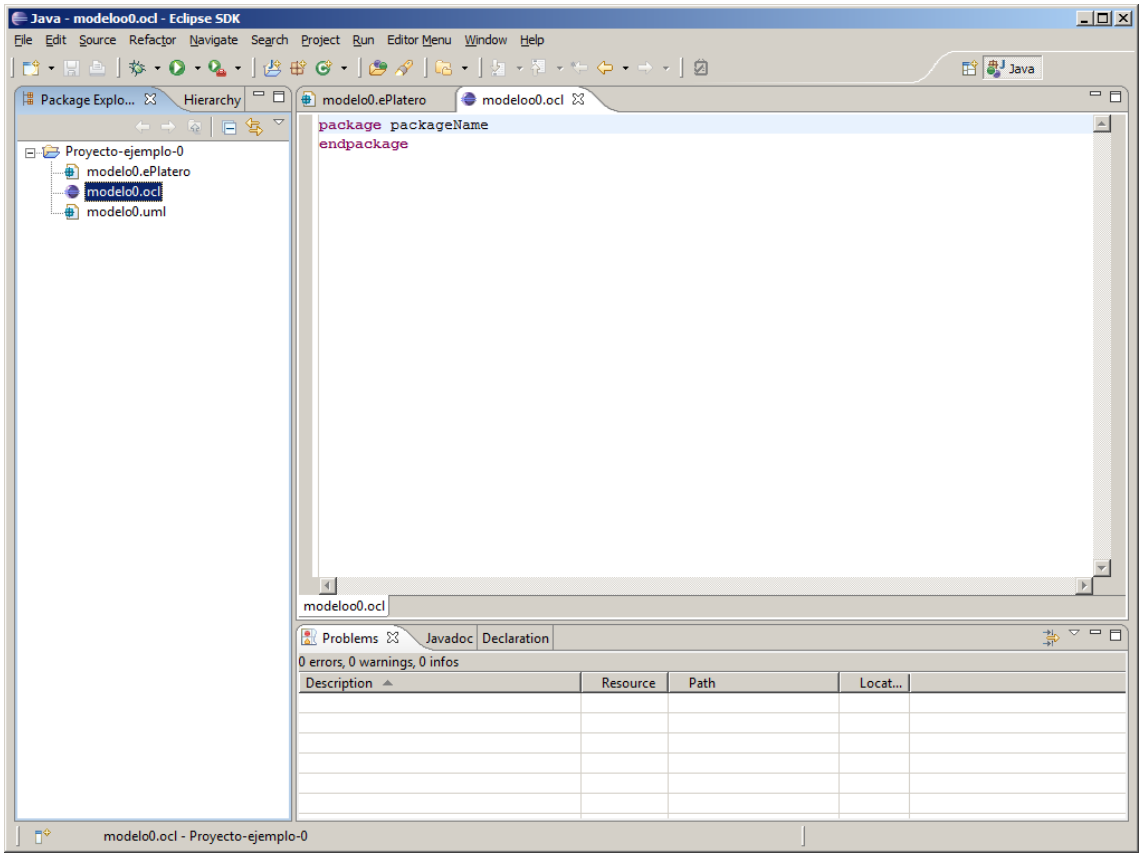
- un archivo .uml
- y otro .eplatero.



A continuación hay que realizar el mismo procedimiento, pero en este caso creando un archivo OCL



El workspace ya está listo para desarrollar los ejemplos, y luce de la siguiente manera:



11.2 Casos de ejemplo

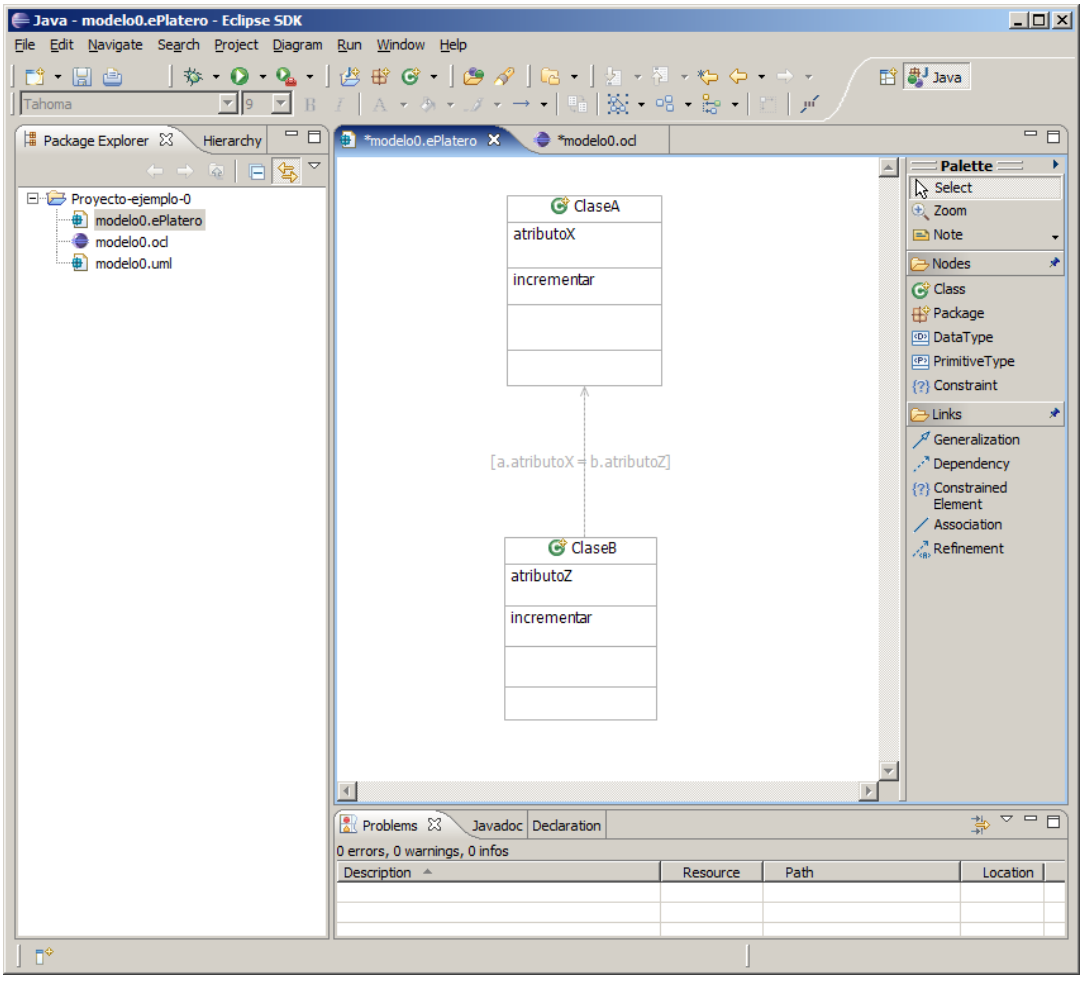
En esta sección se mostrará la ejecución en ePlatero con cada una de las reglas de refactorización definidas en la sección 8.1. Para cada regla se mostrará la representación de su diagrama UML en el editor del ePlatero junto con las restricciones OCL. A continuación se procederá a refactorizar las restricciones y mostrar el resultado final.

11.2.1 Renombrar Atributo

El diagrama UML de la definición de esta regla muestra una clase, *ClaseA*, la cual tiene definido el atributo *atributoX*.

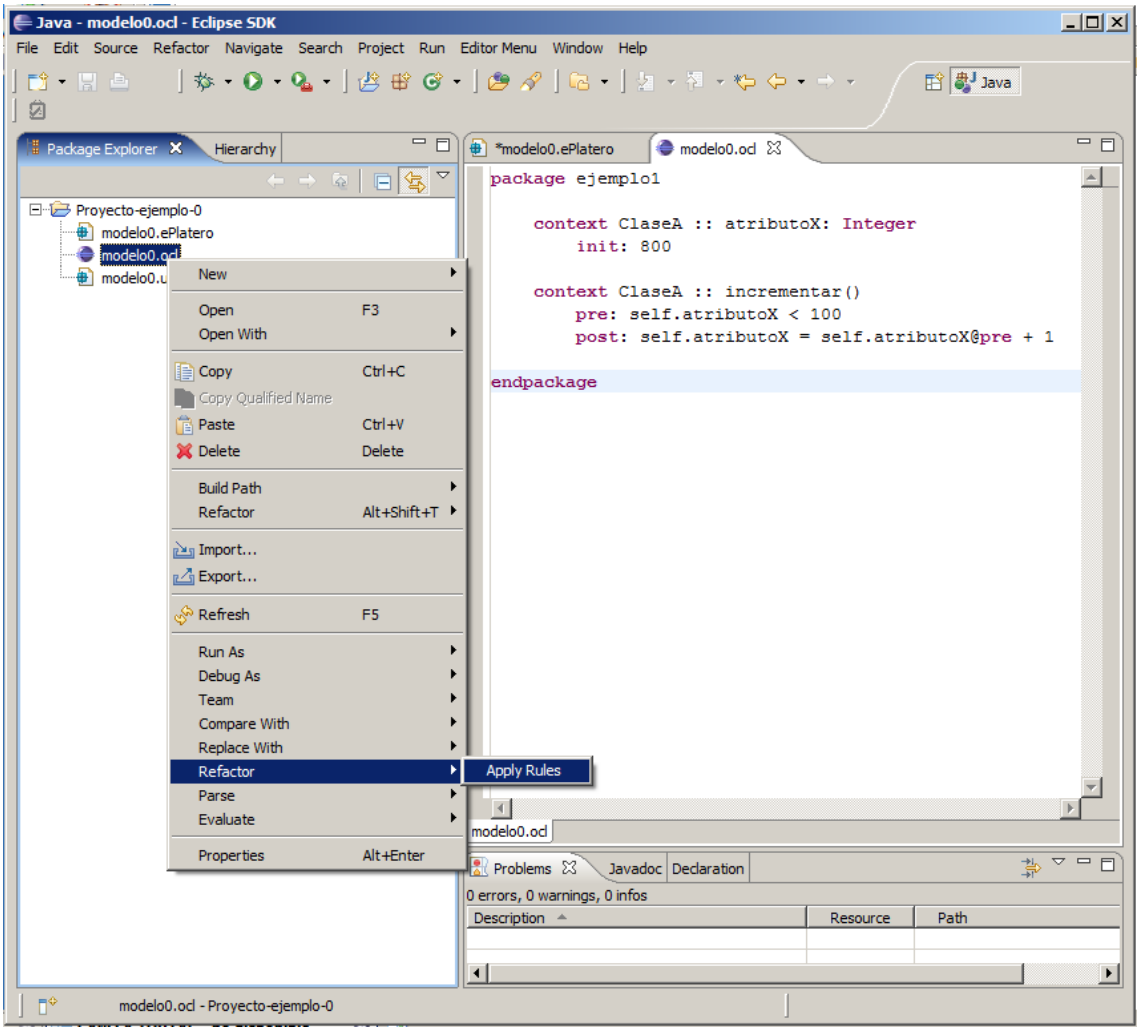
La refactorización del UML especifica que el *atributoX* se debe renombrar a *atributoZ*.

A continuación se muestra la herramienta ePlatero con este ejemplo representado.



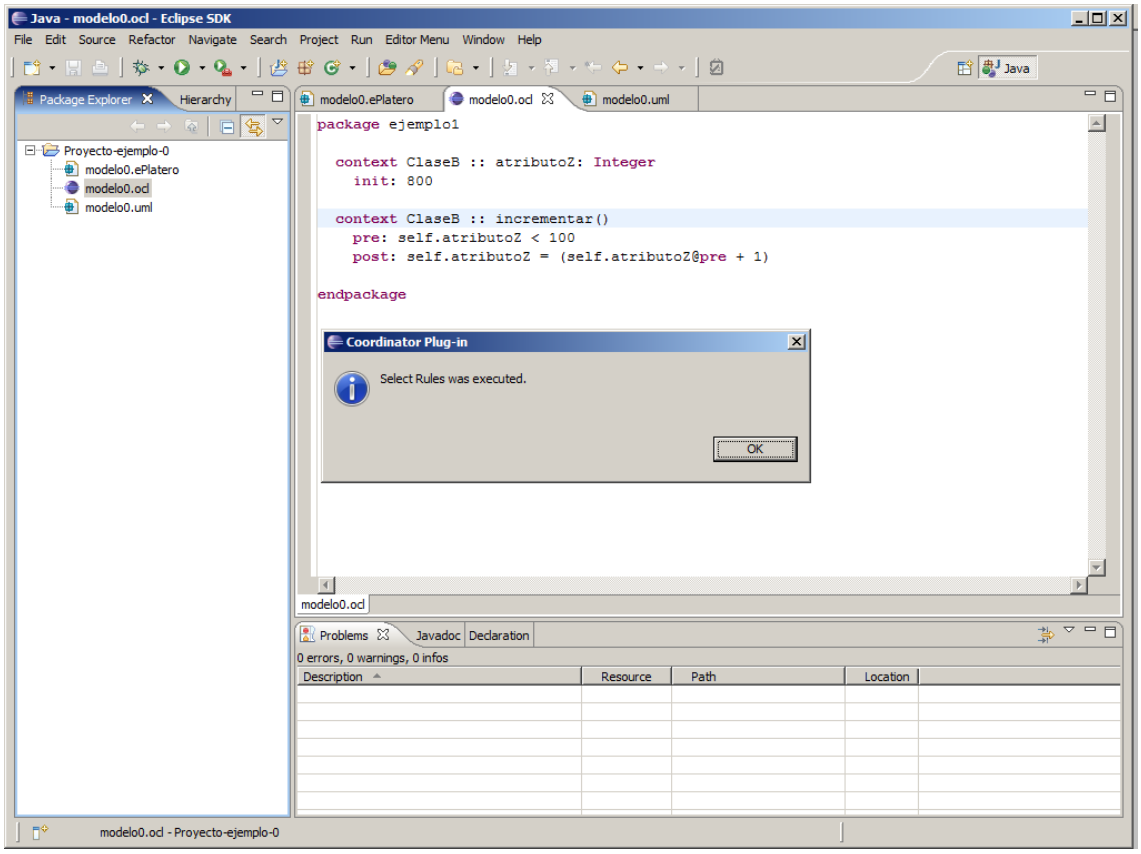
El siguiente paso es definir las restricciones OCL. Para este ejemplo, se define una restricción de valor inicial y una operación con pre y post condiciones.

La siguiente figura muestra las restricciones configuradas en ePlatero y el menú sobre el cual se invoca a la refactorización de OCL.



La aplicación requerirá especificar el diagrama UML sobre el cual se ejecutará la refactorización, en este caso *modelo0.uml*.

Luego de la refactorización, el contenido del archivo de restricciones OCL es actualizado con las restricciones refactorizadas. El plugin informa que el proceso ha finalizado correctamente.



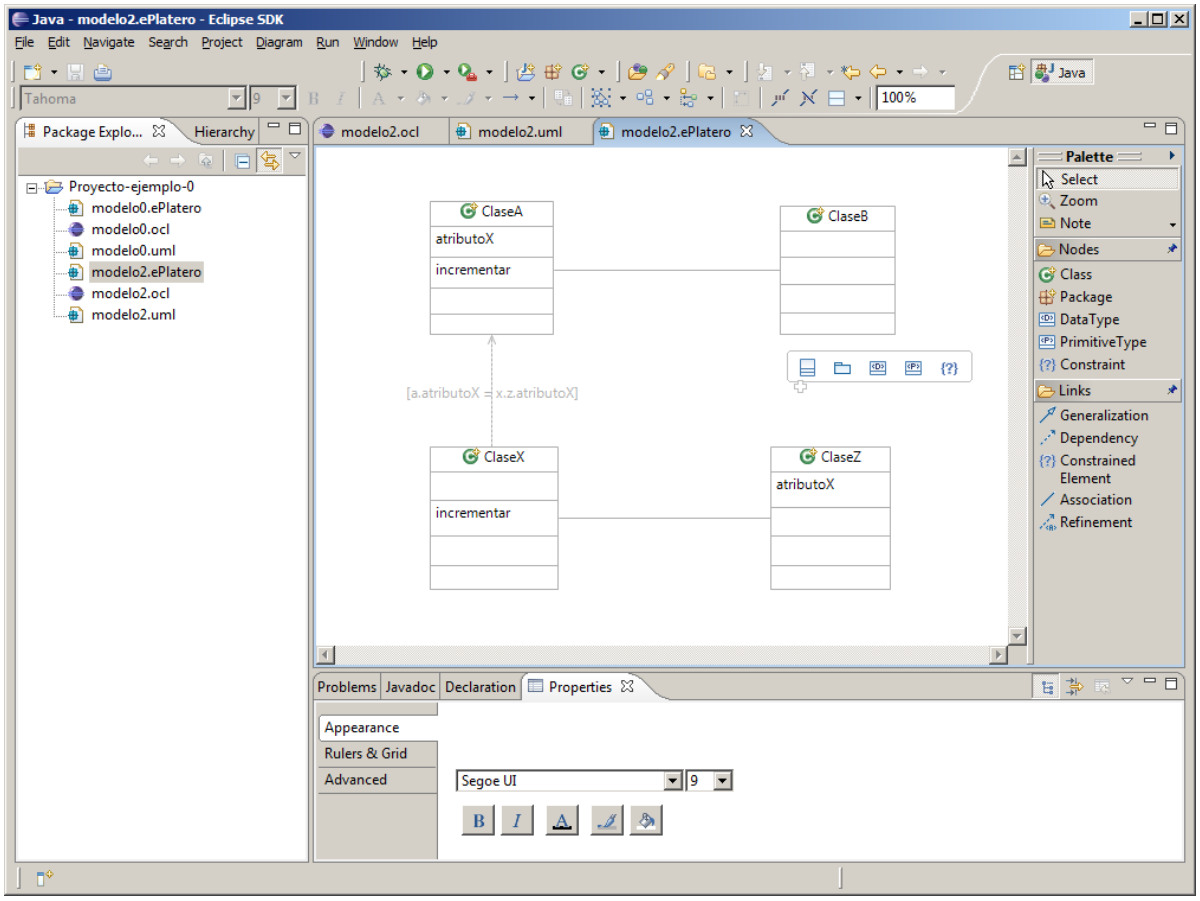
Como puede verse en la imagen, el contenido del archivo OCL fue actualizado para ser consistente con el diagrama UML refactorizado, es decir, la clase *ClaseB*.

La restricción de valor inicial del *atributoX* de la clase *ClaseA* se transformó en una restricción de valor inicial del atributo correspondiente de la clase *ClaseB*, es decir, *atributoZ*. Los valores contenidos en esa restricción permanecieron inalterados.

Además, la operación *incrementar* sufrió cambios en sus pre y post condiciones. La pre-condición que recaía sobre el *atributoX*, pasó a utilizar el *atributoZ*. También la post-condición fue actualizada de la misma forma.

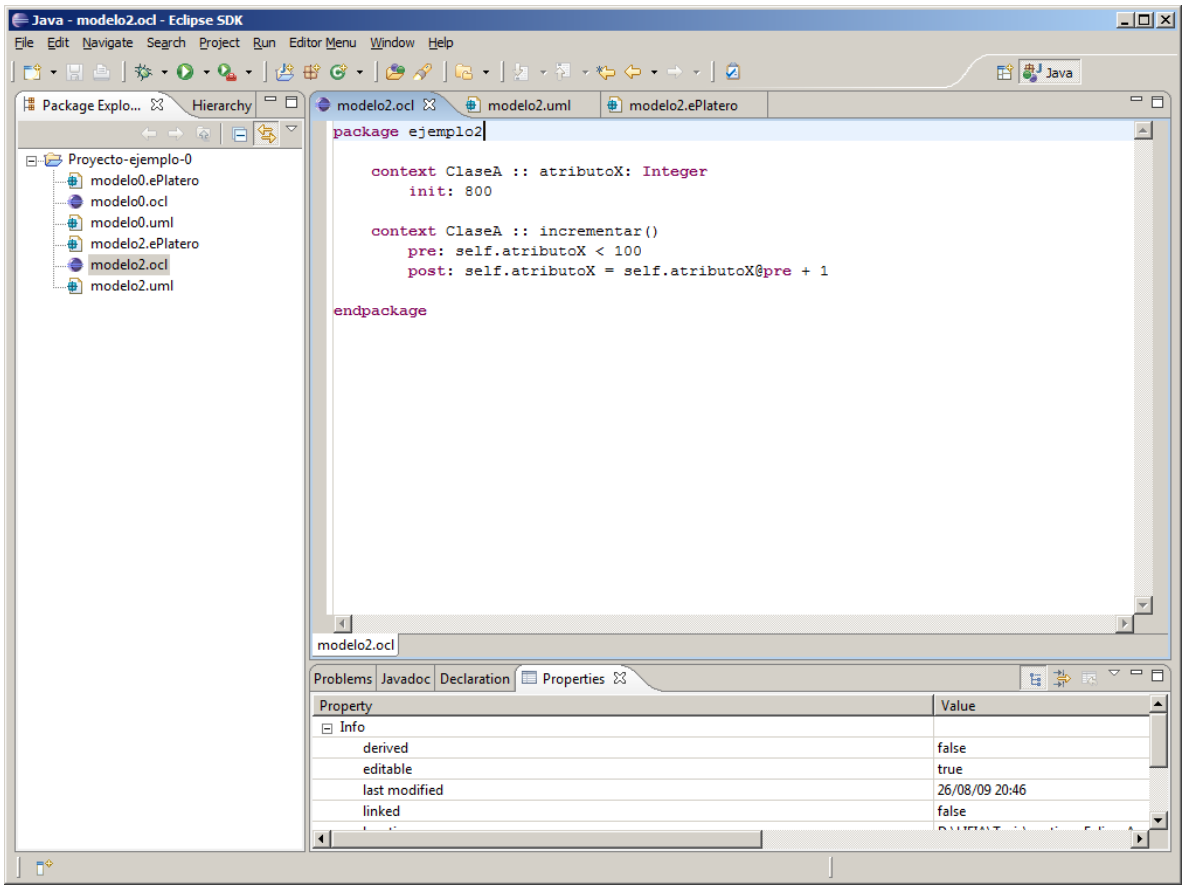
11.2.2 Mover Atributo

Para esta regla, el ejemplo muestra dos clases unidas por una relación de 1 a 1. Una de las clases tiene definido un atributo que en la refactorización pasa a pertenecer a la otra clase. La representación de este ejemplo en ePlatero se muestra a continuación.



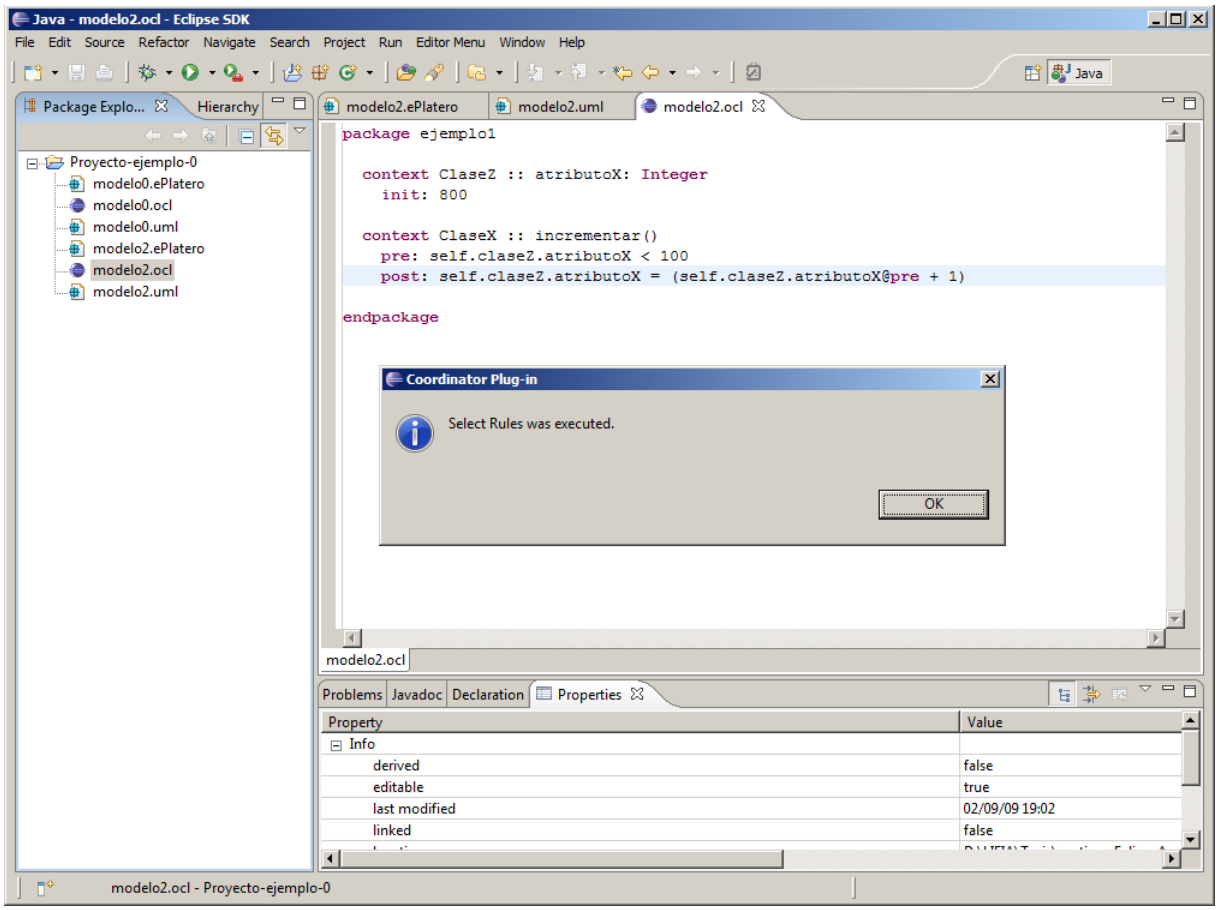
La refactorización mueve el *atributoX* desde *ClaseA* a la clase asociada, en este caso llamada *ClaseZ*. Las restricciones OCL que apunten a dicha variable, tienen que ser cambiadas de contexto por *ClaseZ* o referenciadas mediante una navegación en la relación de *ClaseX* a *ClaseZ*.

A continuación se muestran configuradas en ePlatero las restricciones a utilizar en el ejemplo:



Igual que en el ejemplo del inciso anterior, se incluyó una restricción de valor inicial y las restricciones sobre una operación.

En la siguiente imagen puede verse el resultado de la refactorización de las restricciones.



Luego de la refactorización, la restricción de inicialización de la variable *atributoX* de la clase *ClaseA*, pasó a pertenecer a la clase *ClaseZ*, que corresponde a la clase asociada correspondiente de la refactorización.

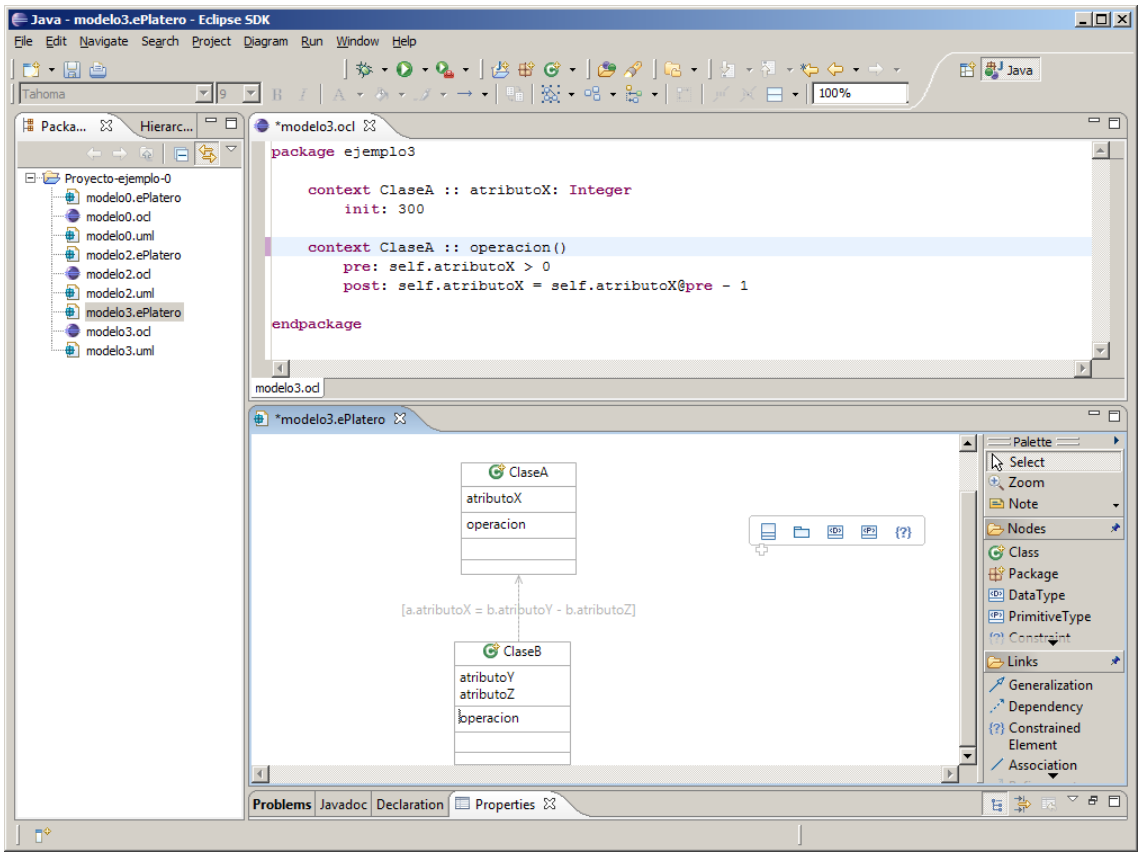
La definición de la operación dejó de utilizar la variable de la clase *ClaseA* (*ClaseX* de la refactorización) para utilizar la relación para acceder a la misma. Se actualizaron las pre y post condiciones.

11.2.3 Refactorizar Atributos

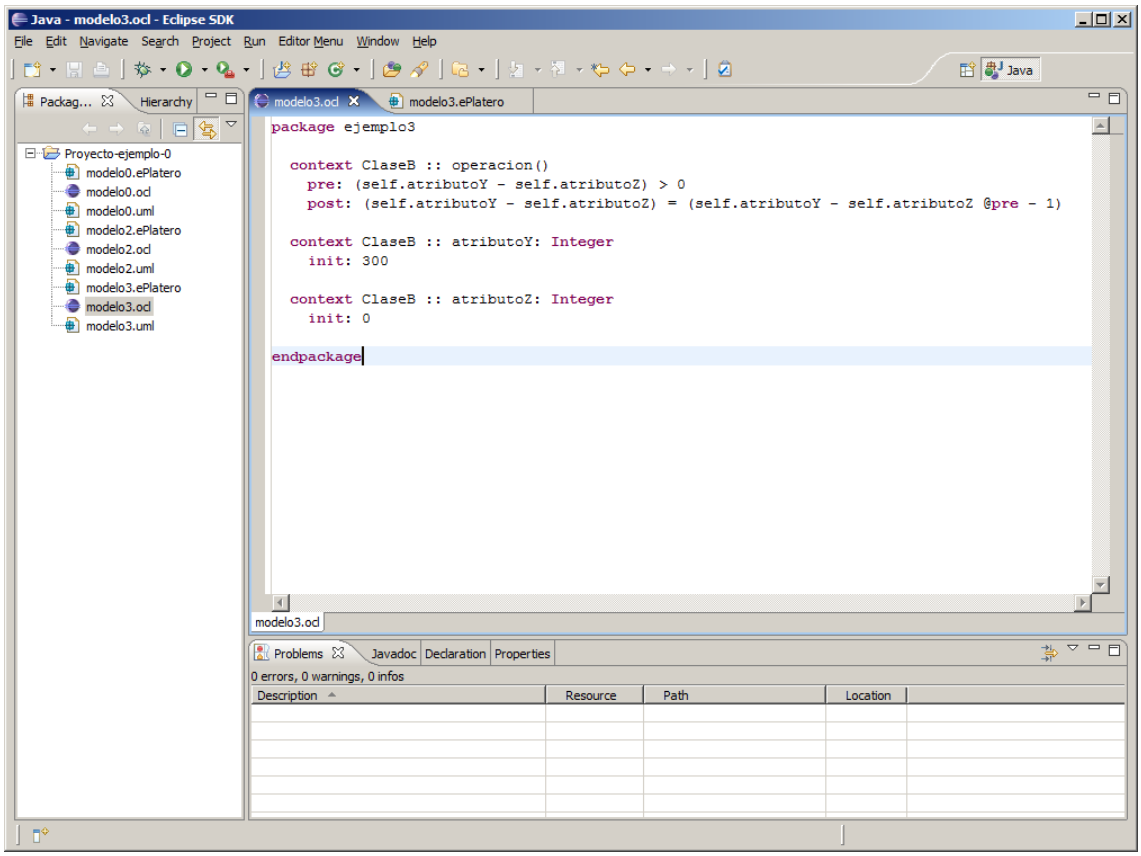
Para demostrar el funcionamiento de esta regla, se eligió un ejemplo muy simple. En el mismo se muestra una clase con un atributo llamado *atributoX* y la clase refactorizada que en lugar de dicho atributo tiene dos nuevos atributos llamados *atributoY* y *atributoZ*.

Las restricciones OCL a refactorizar son una expresión de valor inicial y una definición de condiciones sobre una operación.

La configuración completa se muestra en la siguiente figura:



Luego de la refactorización, el archivo de restricciones OCL queda de la siguiente manera:



La restricción de valor inicial de la variable *atributoX* ha desaparecido. En su lugar, se crearon expresiones de valor inicial para las variables *atributoY* y *atributoZ*. Los valores iniciales de las nuevas variables fueron calculados por la herramienta de forma tal que se mantiene la relación entre las mismas.

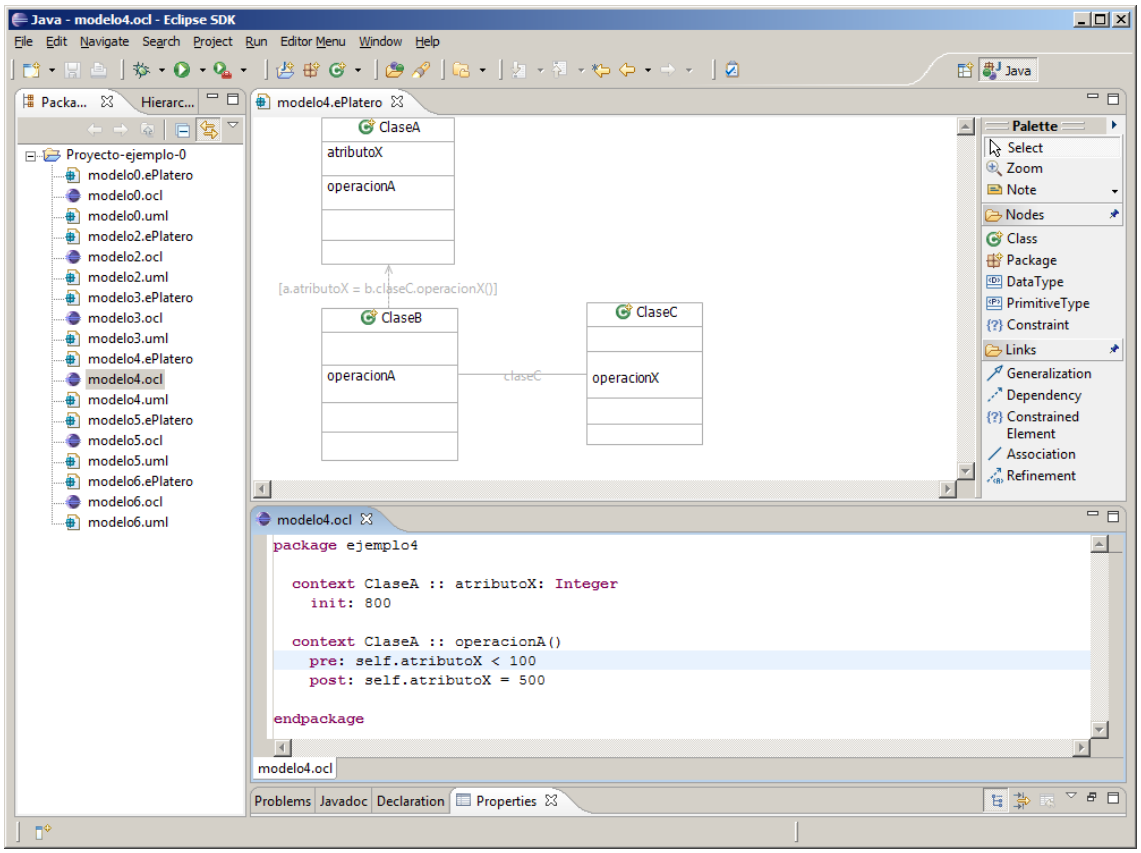
Además de ello, fue refactorizada la operación definida. Las pre y post condiciones fueron actualizadas. Las referencias a la variable *atributoX* fue reemplazada por la equivalencia definida en el mapeo (en este caso una diferencia entre las nuevas variables).

11.2.4 Transformar Atributo en Valor Calculado

Para el ejemplo correspondiente a esta regla se utilizará un modelo en el que están representadas tres clases: La clase *ClaseA* que posee la variable *atributoX* y dos operaciones, la clase *ClaseB* que es la refactorización de *ClaseA* y a la cual mediante la refactorización se le ha eliminado el atributo y ha quedado solamente con las dos operaciones y una nueva clase agregada en la refactorización llamada *ClaseC*, la cual posee la operación *operacionX*.

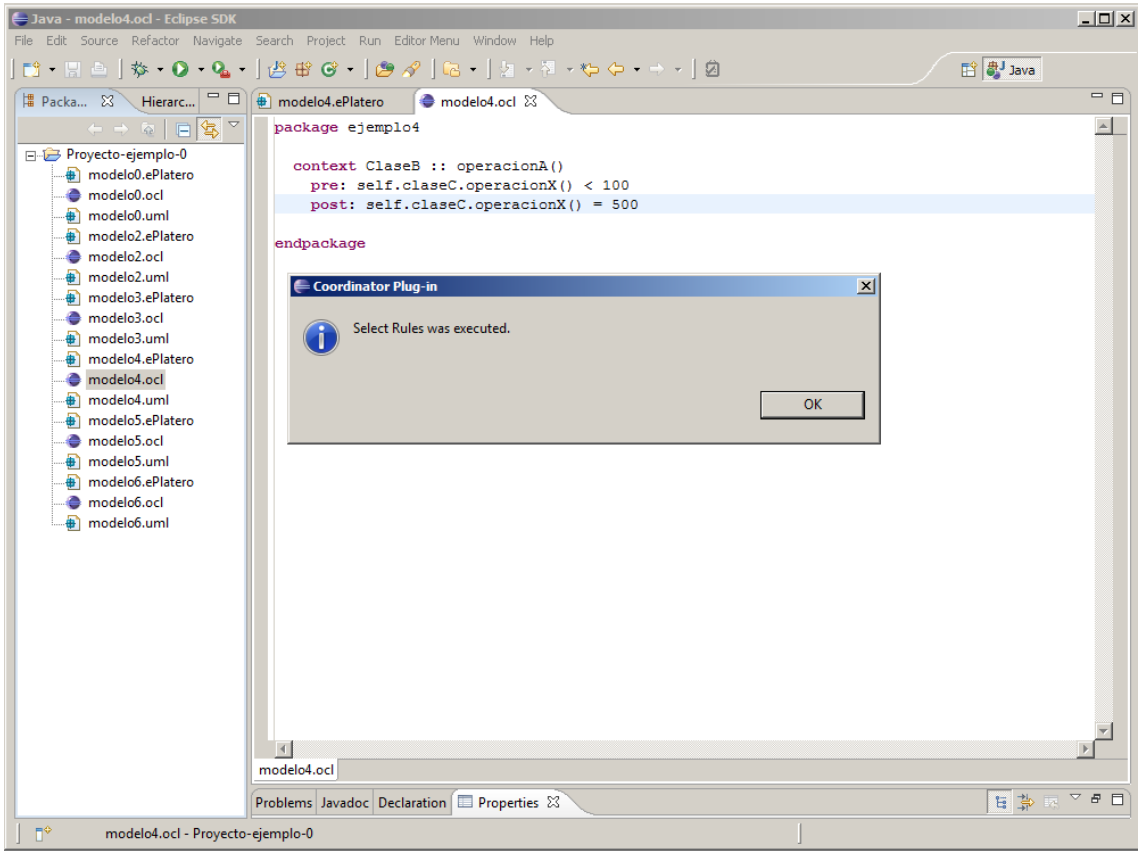
Las restricciones OCL son una expresión de valor inicial para la variable de la clase ClaseA y una operación con pre y post condiciones.

El comportamiento esperado por el modulo de refactorizaciones de OCL es que se eliminen la inicialización de la variable eliminada y que las referencias a la misma sean reemplazadas por el valor especificado en el mapeo, en este caso, una de las operaciones.



En la captura anterior puede verse la representación del ejemplo planteado en este capítulo.

A continuación se muestra el resultado de las reglas OCL luego de la refactorización:

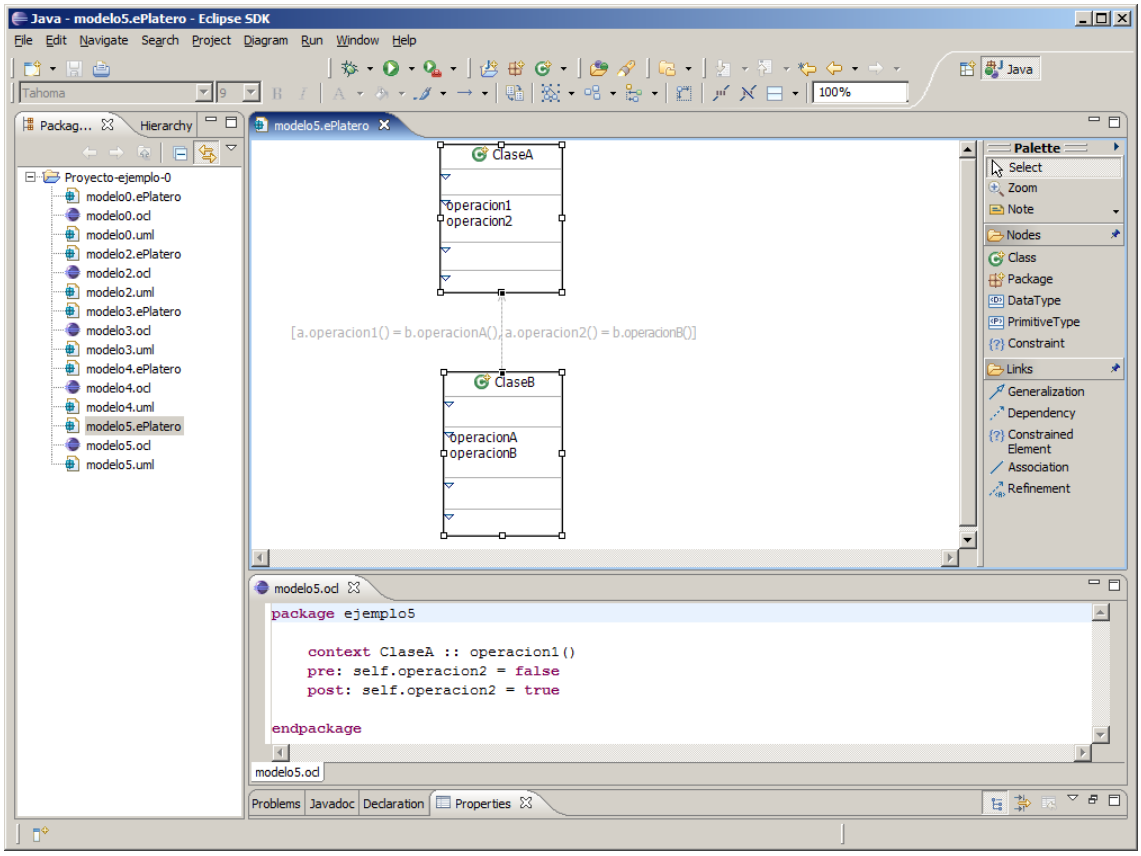


Como puede verse, la ejecución del plugin produjo el resultado esperado.

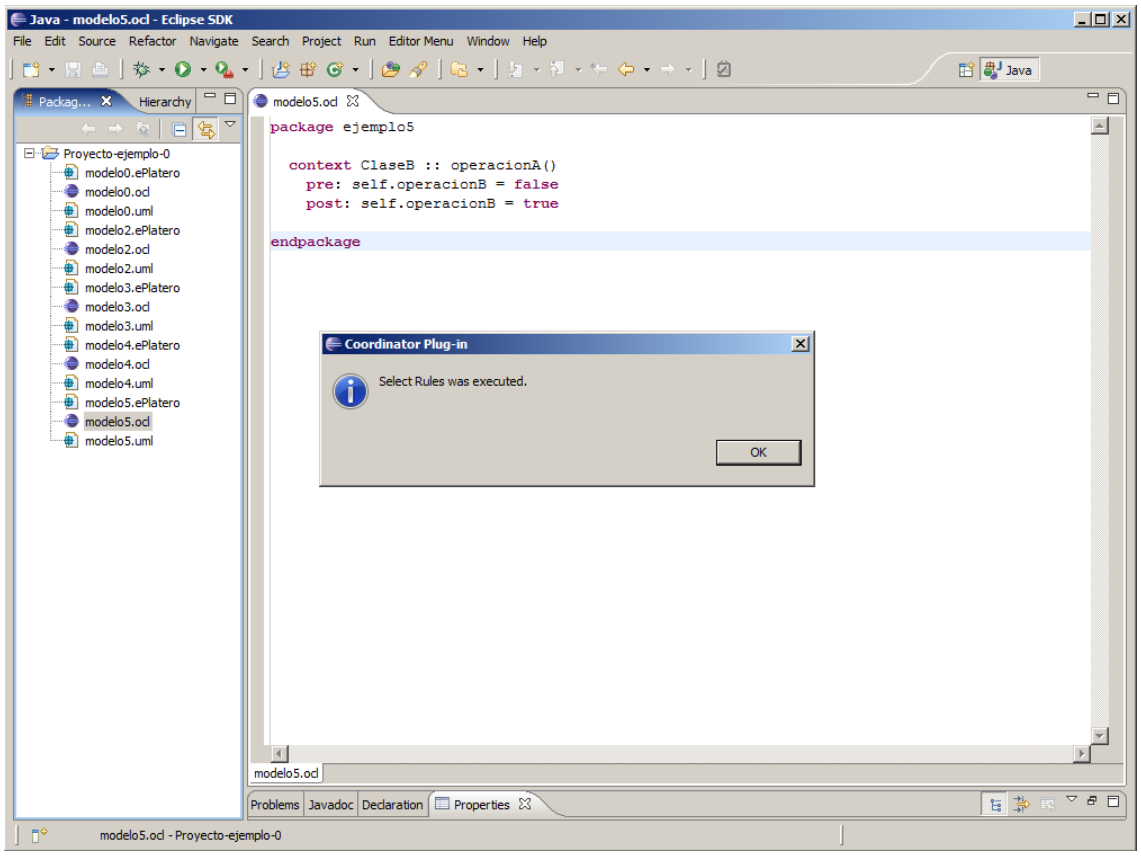
11.2.5 Renombrar Operación

Esta regla tiene como objetivo cambiar el nombre de una operación. Para demostrar el funcionamiento de la herramienta se utilizará un ejemplo en el que se define una clase con dos operaciones: *operacion1* y *operacion2*. Luego de la refactorización, ambas operaciones cambian de valor.

El diagrama del ejemplo se complementa con una restricción OCL sobre una de las operaciones, y utiliza la referencia a la otra operación en las pre y post condiciones.



El comportamiento esperado al aplicar esta regla en la refactorización es que tanto la definición de la operación como las pre y post condiciones se actualicen. A continuación se muestra el resultado de la ejecución de las reglas.



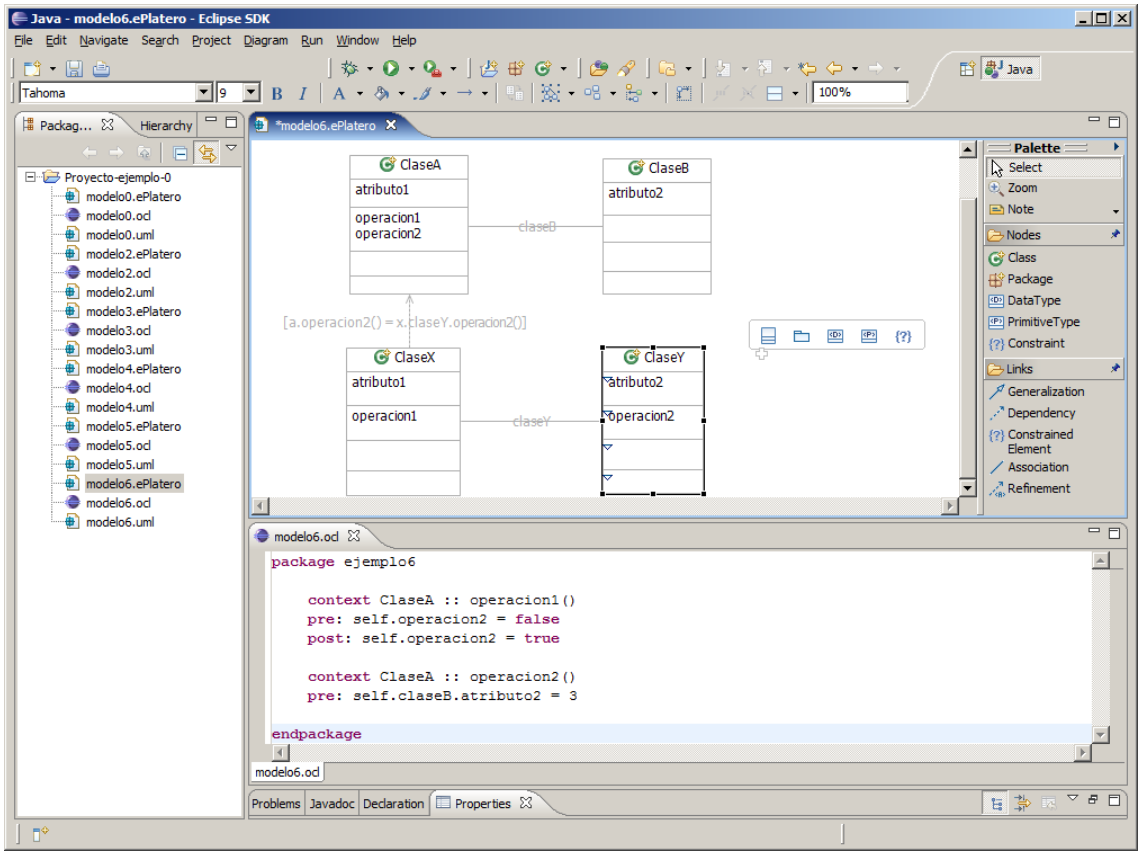
Como puede verse, las operaciones se renombraron en todas sus apariciones.

11.2.6 Mover Operación

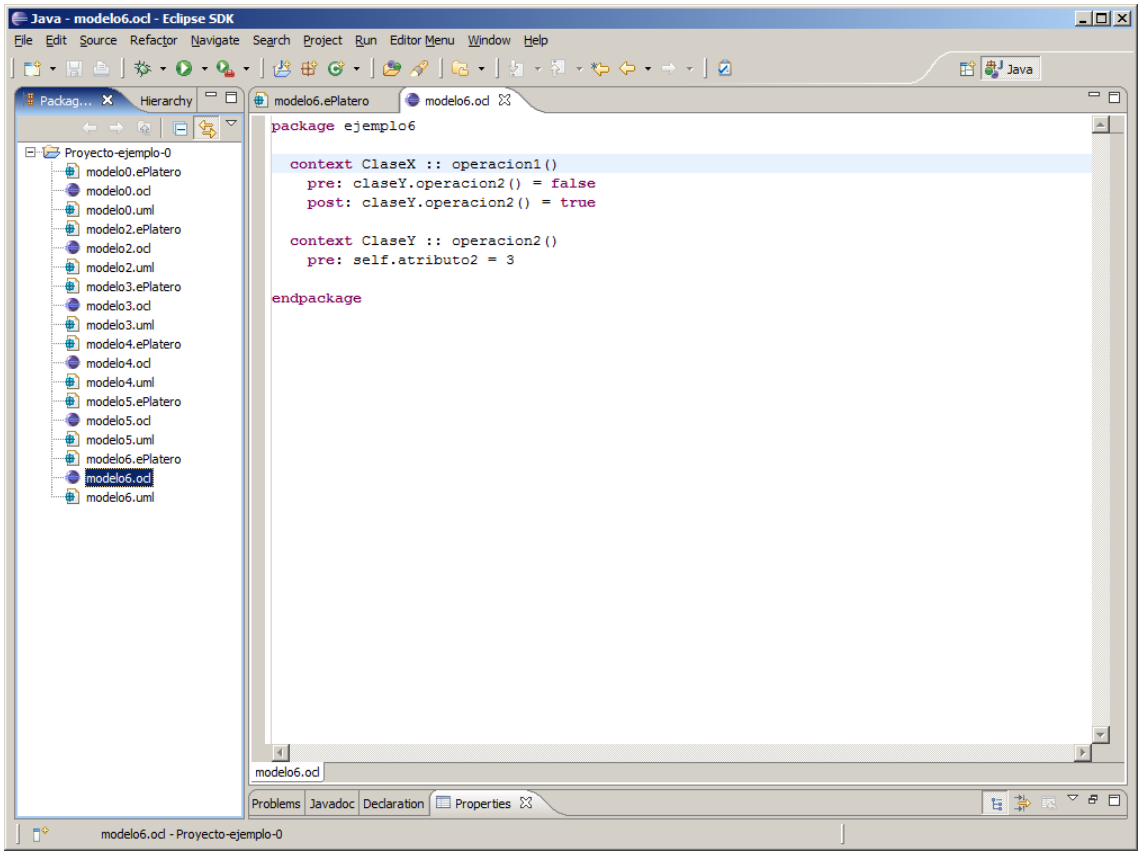
Para ejemplificar el funcionamiento de esta regla, se usó un diagrama en el cual hay definidas dos clases: *ClaseA* y *ClaseB*. *ClaseA* tiene dos operaciones, una de las cuales es movida a *ClaseB* en la refactorización.

Las restricciones OCL que se definieron son dos, una sobre la operación movida y otra sobre la que se conserva en *ClaseA*, pero que utiliza a la operación *operacion2* en las pre y post condiciones.

La representación del ejemplo en ePlatero puede verse a continuación.



Al realizar la refactorización, el resultado es el siguiente:



La operación *operacion2* fue movida de contexto a *ClaseY*.

La operación *operacion1* se conserva en *ClaseX* pero sus pre y post condiciones que utilizaban *operacion2* fueron redefinidas para adaptarse al cambio en la misma.

Capítulo 12: Conclusiones

12.1. Trabajos relacionados

A continuación se presentan algunos trabajos relacionados con el tema de la presente tesis:

- Refactoring OCL Annotated UML Class Diagrams - Slavisa Markovic and Thomas Baar.

Al igual que esta tesis, el paper mencionado trata de la refactorización del código OCL. Se basa en un conjunto de reglas de refactorización para Java escritas por Fowler en un trabajo llamado “Improving the Design of Existing Programs” y las adapta al código OCL.

A diferencia de esta tesis, el paper define reglas para el código OCL, pero no tiene en cuenta la relación entre estas reglas y la refactorización de diagramas UML.

- Definition and Correct Refinement of Operation Specifications - Thomas Baar, Slavisa Markovic, Frederic Fondement, and Alfred Strohmeier.

Este paper trata de la refactorización de modelos. En particular analiza como los cambios en la estructura de los diagramas UML impacta en las restricciones OCL asociadas. Un análisis similar se realizó en el capítulo 7 de esta tesis.

Este paper no se ocupa de reglas para refactorizar los diagramas UML ni el código OCL.

- El modelo relacional en el marco de las transformaciones de modelos.
Jerónimo Irazabal
Tesina de Licenciatura en Informática – Plan 90
Marzo 2009

Esta tesis encara el tema de la refactorización de modelos igual que en esta tesis. Se define también un lenguaje para basar las transformaciones. La diferencia es que la tesis se ocupa de la transformación de modelos relacionales, en lugar de diagramas UML.

12.2. Conclusiones

El desarrollo de software dirigido por modelos (MDA) es una técnica de desarrollo de software que está cobrando mucha fuerza en los últimos años. Mediante el uso de modelos y refactorizaciones de los mismos pretende combatir muchos de los problemas actuales del desarrollo de software.

El elemento principal de MDA son los modelos. Estos son definidos en UML y necesitan de un lenguaje de restricciones para completar su significado. El lenguaje utilizado para ello es OCL. Con la combinación de UML y OCL se consiguen modelos precisos y completos, con el nivel de detalle requerido por MDA.

Sin embargo, para que la industria adopte este proceso de desarrollo es necesario contar con herramientas adecuadas. La herramienta CASE ePlatero es un excelente ejemplo de este tipo de herramientas. ePlatero es una herramienta muy potente, especialmente para el modelado de diagramas UML y los refinamientos de los mismos. Sin embargo, no proveía soporte para el refinamiento de diagramas con código OCL asociado.

En esta tesis se definió la forma de refactorizar diagramas UML con restricciones OCL. En particular, se creó un catálogo de reglas de refactorización para restricciones OCL.

Las reglas se generan a partir de las reglas de transformación de UML de ePlatero. Al aplicar las reglas tanto a los diagramas UML como a las restricciones OCL, se logra que los diagramas se conserven consistentes.

El catalogo de reglas tiene la ventaja de ser fácilmente extensible. En esta tesis se definieron varias reglas, y se sentaron las bases para la ampliación del catálogo en trabajos futuros.

Además de ello, se implementó la extensión de ePlatero que agrega la capacidad de refactorizar diagramas UML con restricciones OCL asociadas. El plugin fue diseñado de forma tal de ser fácilmente extensible al agregar nuevas reglas al catálogo.

Capítulo 13: Referencias Bibliográficas

[1] “An OCL-based Technique for Specifying and Verifying Refinement-oriented Transformations in MDE” C. Pons, and D. Garcia. In: Lecture Notes in Computer Science ISSN 0302-9743. volume 4199, pp. 645 ? 659, 2006. © Springer-Verlag Berlin Heidelberg 2006.

[2] “Refactoring OCL Annotated UML Class Diagrams” Slavisa Markovic and Thomas Baar. Ecole Polytechnique Federale de Lausanne (EPFL). School of Computer and Communication Sciences. Lausanne, Switzerland

[3] “Refactoring: Improving the Design of Existing Programs”. Fowler Martin. Addison-Wesley

[4] Refactoring community: Refactoring homepage - http://www.refactoring.com/

[5] “Definition and Correct Refinement of Operation Specifications” Thomas Baar, Slavisa Markovic, Frederic Fondement, and Alfred Strohmeier. Ecole Polytechnique Federale de Lausanne (EPFL). School of Computer and Communication Sciences. Lausanne, Switzerland

[6] Unified Modeling Language - http://www.uml.org/

[7] Object Constraint Language Specification - http://www.omg.org/technology/documents/formal/ocl.htm

[8] The Object Management Group (OMG) - http://www.omg.org/

[9] Eclipse - an open development platform – http://www.eclipse.org

[10] EPlatero - Desarrollo de Software Dirigido por Modelos - http://lifia.info.unlp.edu.ar/eclipse/

[11] “MDA Guide Version 1.0.1” - Joaquin Miller and Jishnu Mukerji

[12] MDA: Reusabilidad Orientada al Negocio - Valerio Adrián Anacleto http://www.epidataconsulting.com/

[13] Explore model-driven development (MDD) and related approaches: A closer look at model-driven development and other industry initiatives - Tracy Gardner and Larry Yusuf

[14] Getting Started with Modeling Maturity Levels - Anneke Kleppe, Jos Warmer

[15] La evolución de la programación hacia la ejecución y validación automática de modelos – Quirón
http://www.epidataconsulting.com/

[16] OCL 2.0. OMG Final Adopted Specification. October 2003.

[17] PAMPERO: Precise Assistant for the Modeling Process in an Environment with Refinement Orientation. C. Pons, R.Giandini, G. Pérez, P. Pesce, V.Becker, J. Longinotti, J.Cengia. In "UML Modeling Languages and Applications: UML 2004 Satellite Activities, Revised Selected Papers" . Lecture Notes in Computer Science number 3297. -- New York :Springer-Verlag. Portugal, October 11-15, 2004 . ISBN: 3-540-25081-6

[18] Implementación de técnicas de evaluación y refinamiento para OCL 2.0 sobre múltiples lenguajes basados en MOF. Diego Gracia, Claudia Pons

[19] Towards Dependable Model Transformations: Qualifying Input Test Data. Franck Fleurey, Benoit Baudry and Pierre-Alain Muller, Yves Le Traon France Télécom R&D.