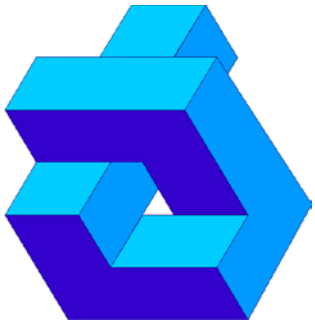
UNIVERSIDAD POLITECNICA DE MADRID



DEPARTAMENTO DE AUTOMATICA

INGENIERIA ELECTRONICA

E INFORMATICA INDUSTRIAL

División de Ingeniería de Sistemas

y Automática (DISAM)

Entorno para el uso de patrones de diseño en la ingeniería de sistemas basada en modelos

Entorno para el uso de patrones de diseño en la ingeniería de sistemas basada en modelos

AUTOR: Eusebio Alarcón Romero

TUTOR: Carlos Hernández Corbato
Ricardo Sanz Bravo

Agradecimientos

A mi hermana, por ser estar siempre ahí sin olvidarte de tu hermano pequeño. Por haber sido siempre un ejemplo de lucha y perseverancia.

A mis padres, por intentar hacerlo todo mejor que bien. Por confiar siempre en mí y hacer posible todo este tiempo en Madrid.
A mi tía, porque "madrina no hay más que una".

A Ana, por cada día en estos últimos 5 años. Por compartir tu alegría en los buenos momentos. Por darme fuerzas y sonreír en los difíciles. Por ser amiga en todos.

A Emilio, David, Javi, Alvaro, Caty y Alfonso. Por tantas noches en el Kutre, experimentos de comida y proyectos absurdos. Porque después de tanto tiempo no me canso de vosotros. Pero principalmente, por hacer que tres meses fuesen un "hasta mañana".

A Esther, Melani, Jenny, Mari y Cris. Porque los comienzos y lo nuevo no son fáciles, pero sí intensos. Por los cumples veraniegos, las noches en la playa y aguantar a unos tíos raros. Por no ser algo pasajero, y dejar de ser las "turismas".

A Pablo y Alfonso, por las largas horas de provecho en clase y la perfección en los trabajos en grupo. Por las sangrías en el Santa Elena, la cafetería después de cada examen y las cenas de especialidad. Por la primera práctica de Máquinas II.

A Laura, Bea, Adri, Soto, Paco y demás amigos de la resi. Por el largo viaje que comenzamos y acabamos juntos. Por la 117 y las fiestas en la sexta planta. Por todos los momentos inolvidables que puedes vivir con alguien que acabas de conocer.

A Carlos, por toda su dedicación y ayuda para realizar este proyecto.
A Adolfo, por no ser capaz de dejarme un problema sin resolver.
A Ricardo, por darme la oportunidad de trabajar con grandes personas.

A todos, muchas gracias.

Resumen

La posibilidad de modelar la estructura de patrones de diseño es una opción muy interesante para los diseñadores. Permite especificar patrones en un determinado dominio de aplicación de forma sencilla, sin la necesidad de centrarse en los detalles de la aplicación particular. Este interés cobra mayor fuerza en las realizaciones de la ingeniería actual, que tienden a ser más grandes y complejas, siendo necesario el uso de la ingeniería de sistemas basada en modelos. Sin embargo no existente un acuerdo a la hora de abordar el problema de capturar los patrones de manera formal, lo que complica su reutilización por otros desarrolladores en la realización de otros sistemas técnicos.

En el presente proyecto fin de carrera, se plantea el anterior problema, en el ámbito del diseño de sistemas de control basados en modelos. Dada la complejidad de los sistemas actuales de control es necesario el modelado en su diseño, y son necesarias herramientas para esta nueva metodología. Haciendo uso del entorno de desarrollo Rational Software Architect y del lenguaje unificado de modelado UML se ha diseñado un lenguaje para la especificación de patrones de diseño, y se ha desarrollado una herramienta que permite su aplicación durante el modelado en la fase de diseño de los sistemas.

El lenguaje de especificación de patrones se basa en el estudio realizado de distintas propuestas que hacen uso de las colaboraciones parametrizadas de UML y los estereotipos UML. Por otro lado, la herramienta encargada de realizar las transformaciones, interactúa con el entorno de modelado en el que trabaja el diseñador a través de la plataforma de modelado proporcionada por Rational; y junto con las librerías para UML2 de eclipse, puede manejar los distintos modelos implicados en el proceso, interpretar los patrones, y realizar las transformaciones oportunas sobre los modelos a los que se le aplican los patrones. En los distintos capítulos del proyecto, se presentarán los estudios realizados sobre la materia y las herramientas consideradas hasta llegar a la solución presentada.

ÍNDICE

Índice de figuras

Índice de tablas

Índice de códigos

Capítulo 1

Introducción

1.1. Sistemas de control en la ingeniería actual

Actualmente, los proyectos de ingeniería se enfrentan al problema del crecimiento del tamaño y complejidad de los sistemas. Los sistemas integran componentes mecánicos, electrónicos y cada vez más software, y esta integración supone también una mayor complejidad en la definición y en la verificación de los mismos. Los sistemas de control son un ejemplo de este aumento de complejidad, en los que, el papel del software es cada vez mayor.

Por un lado, la creciente complejidad y heterogeneidad de estos sistemas presenta nuevas dificultades para la ingeniería de los mismos, demandando nuevos métodos de descripción de los mismos que permitan una visión más holística. Por otro, estos sistemas de control presentan nuevos retos para garantizar los requisitos de robustez y disponibilidad que demanda la tecnificada sociedad actual. Ya no es posible mantener esa responsabilidad en operarios humanos: el flujo informacional para la toma de decisiones en estos sistemas es de tal volumen que su tratamiento ha de ser automatizado, se hace imprescindible dotar a estos sistemas de mayores niveles de autonomía

La Ingeniería de Sistemas plantea una aproximación multidisciplinar que se está orientando a un enfoque basado en modelos frente al tradicional enfoque documental, a la hora de describir los sistemas.

Los modelos son representaciones parciales de los sistemas que utilizan una notación y una semántica claramente definidas para describir de forma rigurosa ciertos aspectos de los mismos, en contraposición al lenguaje natural empleado para las descripciones tradicionales de los sistemas.

Son necesarios pues sistemas más autónomos y métodos de ingeniería basados en la ingeniería de sistemas y en los modelos. El proyecto Asys, en el que se enmarca este proyecto, plantea el desarrollo de una metodología de desarrollo de estas características.

1.2. Marco de desarrollo del proyecto

El presente proyecto fin de carrera se enmarca dentro del proyecto ASys (Autonomous Systems) que se desarrolla en el laboratorio de sistemas autónomos ASLab (Autonomous Systems Laboratory) perteneciente a la división de ingeniería de sistemas y automática DISAM.

Proyecto ASys

El proyecto ASys es un programa de investigación a largo plazo, dirigido al desarrollo de una tecnología para el desarrollo sistemático de sistemas autónomos. Éste incluye métodos para el análisis de requisitos, arquitecturas de la autonomía y activos reutilizables, como herramientas, ontologías, etc.

ASys y la ingeniería de sistema autónomos

Durante su operación, los sistemas pueden necesitar soportar perturbaciones externas, cambios en la especificación original, o dinámicas inesperadas que no siempre se pueden predecir. Los ingenieros de producción y automáticos anhelan los sistemas autónomos, capaces de trabajar por su cuenta. Sin embargo, se considera que esta autonomía está limitada (por ejemplo, por el ingeniero que lo ha desarrollado). Esto es, en cierto sentido, una gran diferencia entre la autonomía natural y la artificial. Los sistemas de autonomía natural se consideran más sistemas autónomos en el sentido etimológico de la palabra (es decir, siguiendo sus propias leyes de comportamiento). En el otro lado, los sistemas de autonomía artificial se comportan de forma autónoma pero sólo hasta cierto grado, estando limitados por restricciones impuestas externamente (por ejemplo, en relación con la seguridad, la economía o el impacto medioambiental).

La decisión estratégica en el proyecto ASys es la consideración de todo el dominio de la autonomía para cualquier sistema autónomo indiferentemente de su aplicación particular. Esto implica la consideración de una gran variedad de sistemas, desde aplicaciones basadas en robots a procesos continuos o sistemas software.

El método de un enfoque progresivo en el dominio permitirá la derivación de activos progresivamente centrados en el dominio que son, al mismo tiempo, compatibles con las abstracciones y la capacidad de proveer valor funcional. La estrategia seguida para incrementar la autonomía de un sistema será la explotación de una clase de patrón particular: lazos de control cognitivo, que son lazos de control basados en el conocimiento.

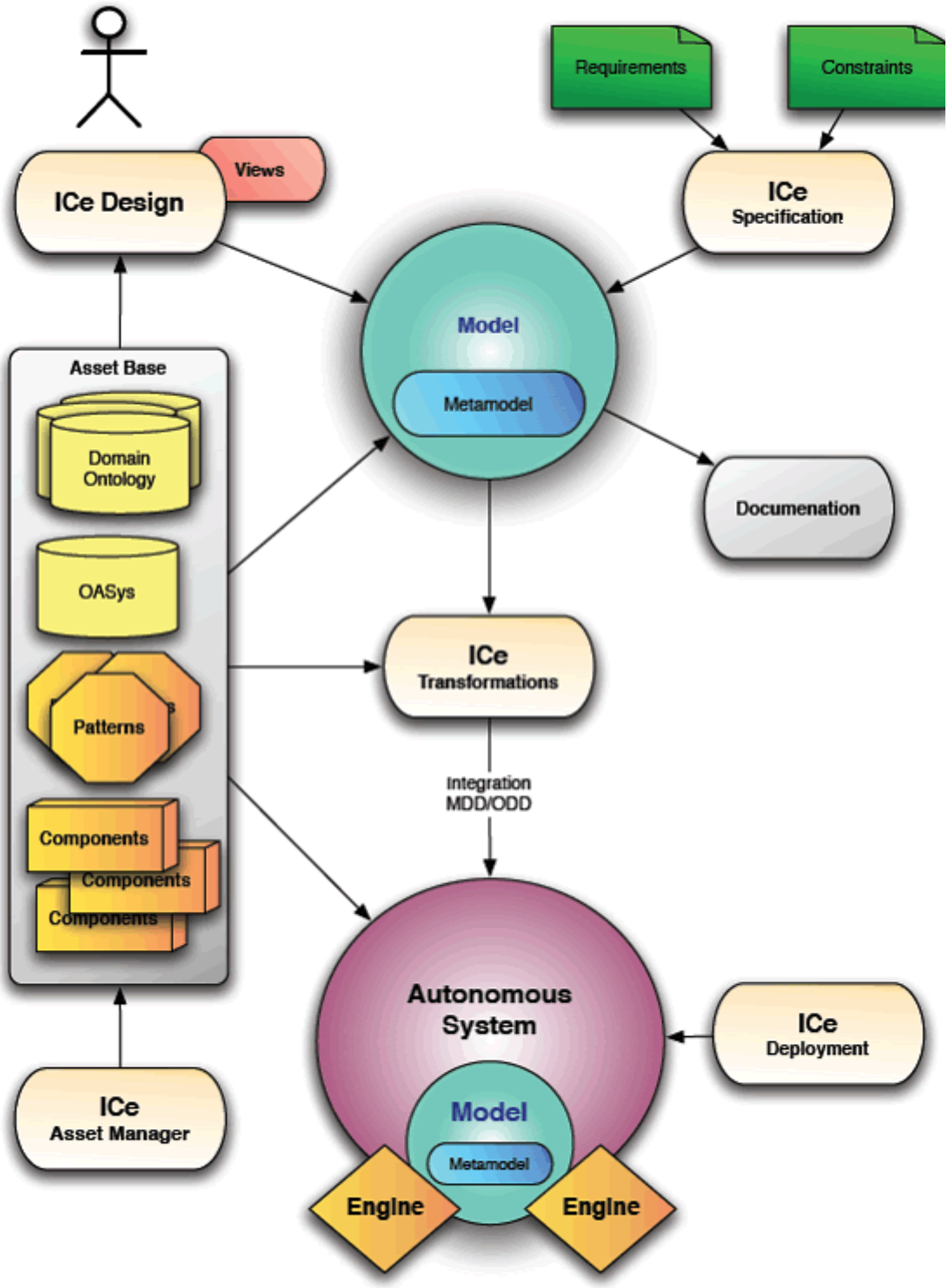


Figura 1.1 Elementos en el Proyecto de investigación ASys

El programa de investigación considera diferentes elementos para materializar las ideas previas (Figura 1.1): un enfoque de diseño centrado en la arquitectura, una metodología para diseñar sistemas autónomos basados en modelos, una base de activos que aportan elementos modulares para que realicen los roles especificados en las arquitecturas de los patrones.

Los procesos de ingeniería cubren desde las especificaciones y conocimientos iniciales, hasta el producto final, que es el sistema autónomo. La primera etapa del proyecto de investigación se centra en ontologías, como una conceptualización común para describir el área de conocimiento. Son tratados tanto el estudio de ontologías de dominios existentes, como el desarrollo de una ontología para el dominio de los sistemas autónomos OASys [4]. Uno de los objetivos centrales es producir una metodología que explote estas ontologías para generar modelos basados en el conocimiento que contienen. Esto se realiza en la fase de diseño como se refleja en la parte superior de la figura 1.1.

Una piedra angular del proyecto ASys es el uso de patrones de diseño como el principal vehículo para la explotación de arquitecturas reutilizables. Los patrones de diseño presentan soluciones recurrentes a problemas de diseño en un cierto contexto. Los patrones de ASys podrían clasificarse en dos categorías: patrones de arquitectura, que expresan la organización estructural de un sistema autónomo, y patrones del dominio, que describen un mecanismo para resolver un problema concreto pero recurrente en un contexto particular.

Los patrones en ASys no serán usados independientemente de la ontología OASys. Los patrones del dominio describirán interacciones entre los componentes del sistema y el entorno, mediante el uso de la conceptualización de la ontología, que representa soluciones de diseño para que el comportamiento del sistema cumpla los requisitos de ingeniería. Ellos modelarán las dinámicas del sistema precisas. Por otro lado, los patrones de arquitectura harán lo mismo para las dinámicas y la organización interna del sistema, describiéndolas en términos de interacción entre los elementos ontológicos que conceptualizan el propio sistema. La definición y la consolidación de patrones de arquitectura que capturan formas de organizar componentes en subsistemas funcionales serán críticas. Como una estrategia de generalización, se evaluarán en diferentes bandos de pruebas los diferentes elementos considerados en el proyecto ASys.

Por lo tanto, todos los patrones del sistema no sólo se especificarán partiendo de los conceptos de OASys, sino que eventualmente formarán parte de la propia ontología, modelando las relaciones y las interacciones entre ellos.

Los modelos constituyen el núcleo para los sistemas autónomos en el proyecto ASys. Los requisitos de usuario y diseñador, y las limitaciones impuestas por el propio sistema guiarán el desarrollo de los modelos. Las metodologías de desarrollo de modelos en ASys se ocuparán de esta caracterización y del desarrollo de los modelos. La siguiente etapa es extraer de los modelos construidos una perspectiva particular de interés para los sistemas autónomos. Las perspectivas funcionales y estructurales

unificadas se consideran críticas para la investigación en la medida en la que proveen conocimiento sobre las intenciones y el comportamiento de los sistemas autónomos.

Los modelos obtenidos serán explotados por medio de máquinas de aplicación comercial o módulos de ejecución de modelado personalizado. Considerando las necesidades metacognitivas de los sistemas autónomos, los metamodelos son abordados en la investigación. El enfoque progresivo en el dominio se utilizará para tratar diferentes niveles de abstracción entre metamodelos y modelos. Se obtendrá un producto de documentación adicional en relación con los modelos construidos para su actualización, intercambio, y consulta.

El enfoque del proyecto ASys es conceptual y centrado en la arquitectura. La conveniencia del control existente y las arquitecturas de control cognitivo se determinarán en términos de cómo encajan con las ideas de investigación de ASys y los productos desarrollados (ontologías, modelos, perspectivas, máquinas). Se considerarán también las posibles adaptaciones y extensiones para las arquitecturas analizadas.

La definición y la consolidación de patrones de arquitectura que capturan formas de organizar componentes en subsistemas funcionales serán críticas. Como una estrategia de generalización, se evaluarán en diferentes bandos de pruebas los diferentes elementos considerados en el proyecto ASys.

1.3. Motivación y objetivos del proyecto

ASys plantea un marco para el desarrollo de sistemas autónomos, pero como solución particular dentro del problema del diseño de sistema de control complejos y robustos. Es en este marco general donde se plantea el uso sistemático del desarrollo dirigido por modelos MDD[1], el modelado y los patrones, se ha realizado el proyecto, que ha consistido en el desarrollo de una metodología de diseño MDD basada en patrones, y las herramientas que le den soporte. Como resultado del trabajo realizado en este proyecto se ha desarrollado una de las herramientas fundamentales de entre los activos de ASys, que permite la especificación de patrones y su uso, en combinación de las ontologías de dominio y OASys, para producir los modelos del sistema autónomo en la fase de diseño

1.3.1. Objetivos del proyecto

El propósito del presente proyecto es el desarrollo de una herramienta para el diseño de sistemas de control mediante el uso de la metodología Model-Driven Development. La herramienta constará de dos componentes: una aplicación software que permitirá

[1] Consultar Punto 2.4 de la memoria

aplicar patrones de diseño en el desarrollo de los modelos UML del sistema y una librería con algunos patrones básicos de control ya diseñados.

La idea es que dado un modelo inicial del sistema a diseñar en su fase de análisis, el diseñador pueda aplicar ciertas soluciones de diseño de manera automática. La herramienta le ayudaría en la fase de diseño, permitiéndole aplicar patrones dados al modelo de su sistema, tal y como resulte de la fase de análisis. La aplicación interactuará con el modelo y el patrón para generar un modelo que será el resultado de la aplicación del patrón al modelo inicial.

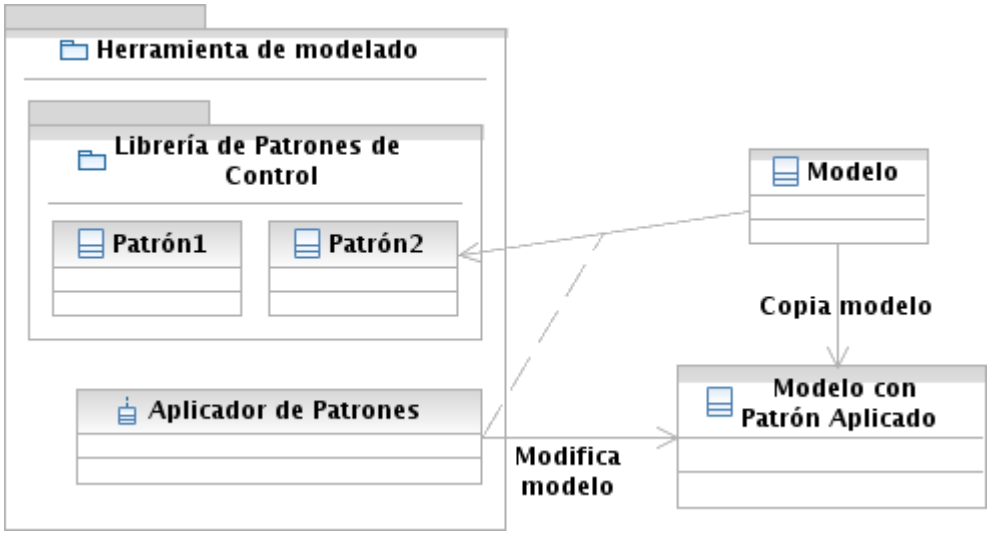


Figura 1.2 Estructura y funcionamiento de la herramienta objetivo del proyecto

El usuario podrá trabajar con sus modelos a través de la interfaz ofrecida por su herramienta de desarrollo, y podrá hacer uso de los patrones de forma sencilla siguiendo las indicaciones para su aplicación. Además, el usuario podrá especificar también sus propios patrones. En el capítulo de análisis del problema se presentará de forma detallada la especificación de requisitos de la herramienta.

Así los principales objetivos son:

- Diseño de una solución para la especificación de patrones mediante modelos.
- Desarrollo de una herramienta que de soporte en la aplicación de los patrones durante el modelado de sistemas.
- Desarrollo de un manual de usuario de la herramienta.
- Diseño de una librería de patrones.
- Documentación del trabajo realizado.

1.3.2. Alcance del proyecto

El alcance del proyecto consiste en el logro de una serie de hitos durante una etapa de documentación y análisis del problema, el desarrollo de unos productos concretos durante la etapa de investigación y desarrollo, y la realización de la documentación que recoja el trabajo realizado. Paso por paso, el alcance del proyecto incluye las siguientes tareas:

- Estudio de la base teórica de las metodologías sobre las que se fundamenta el proyecto.
- Estudio del estado de investigación en la materia en la que se va a trabajar.
- Familiarización con el entorno y las herramientas desarrollo, así como el estudio de sus posibilidades.
- Análisis de las soluciones disponibles para cumplir los requisitos del proyecto.
- Desarrollo de la aplicación para la aplicación de los patrones.
- Desarrollo de la librería de patrones de control.
- Realización de pruebas y verificación.
- Desarrollo de un manual de usuario.
- Desarrollo de la memoria final del proyecto.

1.3.3. Objetivos personales

El desarrollo de este proyecto fin de carrera me supone como objetivo principal el tener que enfrentarme a una metodología de trabajo y las materias sobre las que se fundamenta, sin tener conocimientos previos. De igual manera, el entorno de trabajo es totalmente nuevo para mí pues nunca he usado Linux, el entorno eclipse de programación o un repositorio de control de versiones. Así pues, cada paso que dé será para aprender e indagar nuevos conocimientos siempre interesantes.

Es emocionante poder trabajar con una herramienta como Rational Software Architect que me permite, entre otras cosas, iniciarme en el diseño de modelos UML de sistemas, trabajar con patrones y transformaciones, mejorar mis habilidades de programación estructurada en Java, y principalmente comprender una metodología de trabajo del desarrollo basado en modelos.

Considero también de interés el proyecto por ser, por un lado my tangible y clásico en cuanto a la existencia de requisitos de cliente, y a la vez muy innovador porque el desarrollo de herramientas de modelado al nivel que se aborda está en la frontera del estado del arte. Además, involucra una disciplina tan crucial para la tecnología actual como la ingeniería de sistemas.

1.4. Estructura del documento

Estado del Arte. Se explica la metodología de trabajo del desarrollo basado en modelos, así como los elementos de los que hace uso: modelos, UML, patrones, etc. También, se presentan las prácticas habituales a la hora de abordar el problema de la especificación y aplicación de patrones.

Herramientas y entorno de desarrollo. Se presenta el entorno usado para la realización del proyecto, así como la explicación del funcionamiento de las herramientas investigadas para desarrollar el proyecto.

Análisis del problema. Se analizan los requisitos del proyecto y las soluciones disponibles para su realización.

Solución adoptada. Se presenta el diseño de la solución que se ha planteado y su implementación.

Ejemplos de uso. Se muestran dos ejemplos para ilustrar el funcionamiento de la herramienta.

Conclusiones. Se hace un repaso y valoración del trabajo realizado, y de los productos que se han desarrollado.

Planificación. EDP y diagrama de Gantt que representan el trabajo llevado a cabo durante la realización del proyecto.

Anexos. Manual de usuario y documentación relativa al código de la herramienta desarrollada.

Capítulo 2

Estado del arte

2.1. Modelado

El modelado software tiene como objetivo la plasmación y la captura en modelos del sistema o aplicación a desarrollar. Los modelos han de ser capaces de transmitir la idea del sistema a todos los implicados en este proceso de desarrollo, desde quien concibe la idea hasta el último programador. Así pues, entendemos los modelos como representaciones abstractas y simplificadas, construidos mediante diagramas, esquemas, símbolos…, de una aplicación o sistema; son una descripción desde un punto de vista particular, en la que se resalta las características que resultan de mayor interés y se omite los detalles irrelevantes para conseguir una mayor claridad en la interpretación.

El concepto de modelado software mediante el uso de elementos visuales existe desde hace unos años pero aún así, todavía, hay veces en las que no se entiende bien la necesidad del modelado y el uso de las herramientas existentes para llevarlo a cabo. Se nos presenta entonces la cuestión, *¿por qué es necesario el modelado?*

El crecimiento de la tecnología y los sistemas informáticos ha conllevado un aumento de la complejidad del desarrollo software. Empezando por el ensamblador, la programación ha ido aumentando su complejidad a la par que se evolucionaba el dominio objeto de representación. Programación estructurada, orientada a objetos, orientada a componentes, orientada a servicios… Ante esta visión de complejidad, surge la necesidad de elevar el nivel de abstracción, de manera que la programación y el código dejen de ser el centro de atención.

Como en la ingeniería software, en los procesos de ingeniería actuales, que son de una gran complejidad y abarcan diferentes disciplinas, surge la necesidad de expresar los sistemas en un nivel de abstracción más alto para después bajar hasta un nivel de mayor detalle.

Actualmente existe un lenguaje genérico para modelar cualquier sistema, el SysML (Systems Modeling Language). El problema es que es una especificación reciente y no tan consolidada en metodologías y herramientas como UML (Unified Modeling Language). Debido a esto, se ha optado por el UML para la realización de este proyecto. Dado que SysML comparte el metamodelo MOF con UML, sería sencillo reconvertir en entorno desarrollado para SysML.

Los modelos, expresados en el lenguaje unificado de modelado UML serán fundamentales para el desarrollo de este proyecto, ya que son la base para el planteamiento del proyecto ASys y para capturar nuestros sistemas de control.

2.2. UML

El UML (Unified Modeling Language) es un lenguaje de modelado de sistemas software respaldado por el OMG (Object Managment Group), consorcio dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos. Actualmente, es el lenguaje de modelado más conocido y utilizado. Es un lenguaje gráfico para visualizar, especificar, construir y documentar artefactos de un sistema. UML ofrece un estándar para describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como procesos y métodos del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes reutilizables.

El UML es único al tener una representación de datos estándar. Esta representación se llama metamodelo. El metamodelo es una descripción de UML en UML. Describe los objetos, los atributos y las relaciones necesarias para representar los conceptos de UML sin una aplicación software.

Figura 2.1 Logos de OMG y de UML

Origen del UML

El lenguaje UML comenzó a gestarse en octubre de 1994, cuando James Rumbaugh se unió a la compañía Rational fundada por Grady Booch, ambos, reputados investigadores de el área de la metodología software.

El objetivo era unificar dos métodos que habían desarrollado: el método Booch, para la descripción de conjuntos de objetos y sus relaciones, y el OMT (Object Modelling Tool), técnica de modelado orientada a objetos. El primer borrador apareció en octubre de 1995. En esa misma época otro reputado investigador, Ivar Jacobson, se unió a Rational y se incluyeron ideas suyas como el método OOSE (Object-Oriented Software Engineering), metodología de casos de uso. Estas tres personas son conocidas como los "tres amigos". Además, este lenguaje se abrió a la colaboración de otras empresas para que aportaran sus ideas. Todas estas colaboraciones condujeron a la definición de la primera versión de UML.

Esta primera versión se ofreció a un grupo de trabajo para convertirlo en 1997 en un estándar del OMG. Este grupo propuso una serie de modificaciones y una nueva versión de UML (la 1.1), que fue adoptada por el OMG como estándar en noviembre de ese mismo año. Desde aquella versión ha habido varias revisiones que gestiona la *OMG Revision Task Force*, siendo la revisión más significativa UML 2.0 que fue adoptada por el OMG en 2005. La última versión publicada fue UML 2.3, en mayo de 2010.

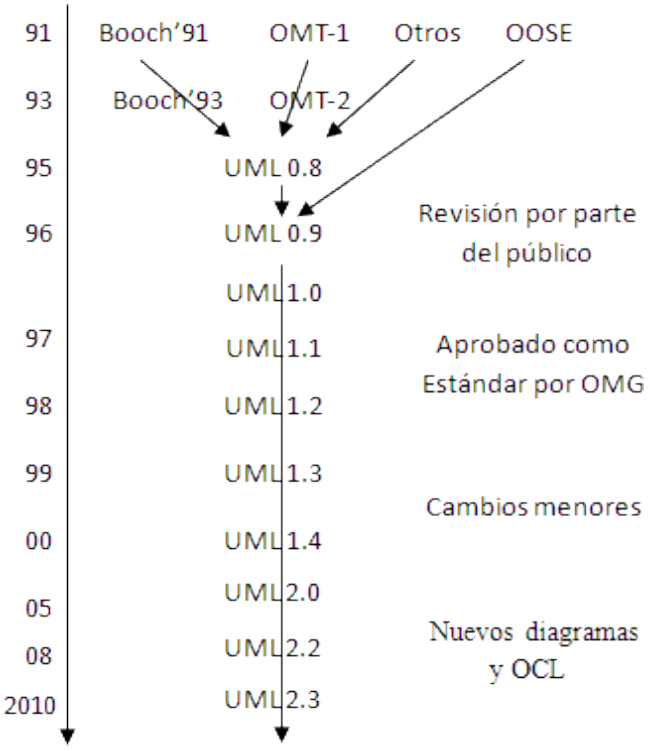


Figura 2.2 Evolución de UML

Hay cuatro partes en la especificación de UML 2.x:

- La superestructura que define la notación y la semántica para sus diagramas y elementos del modelos
- La infraestructura que define el núcleo del metamodelo sobre el que se basa la superestructura.
- El lenguaje OCL (Object Constraint Language) para escribir restricciones sobre los modelos.
- El diagrama de intercambio de UML, que define cómo dos diseños de diagramas UML se intercambian.

De las tres metodologías de partida, las de Booch y Rumbaugh pueden ser descritas como centradas en objetos, ya que sus aproximaciones se enfocan hacia el modelado de los objetos que componen el sistema, su relación y colaboración. Por otro lado, la metodología de Jacobson es más centrada a usuario, ya que todo en su método se deriva de los escenarios de uso. UML se ha ido fomentando y aceptando como estándar desde el OMG, que es también el origen de CORBA, el estándar líder en la industria para la programación de objetos distribuidos.

Diagrama de clases

El diagrama de clases es uno de los diagramas usados en UML para modelar elementos estáticos. El propósito principal del diagrama es representar las clases dentro de un modelo. En una aplicación orientada a objetos, las clases tienen atributos, operaciones y relaciones con otras clases. El diagrama de clases de UML puede representar todos estos elementes fácilmente. El elemento fundamental del diagrama de clases es el icono que representa a una clase (Figura 2.3).

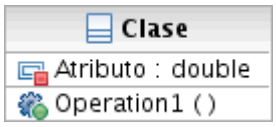


Figura 2.3 Representación de una clase en UML

Es un rectángulo dividido en tres partes: el nombre de la clase, los atributos y las operaciones. Los atributos y las operaciones se pueden omitir en la representación si no son útiles en una representación particular. Cada atributo va seguida de dos puntos y el tipo del atributo. En las operaciones, se puede indicar los tipos de los argumentos y los resultados también. Además, se puede indicar la visibilidad del los atributo y las operaciones con distintos signos delante de los nombres.

Las clases puedan estar relacionadas de distintas maneras, y de acuerdo al significado de estas relaciones, se utilizan distintas representaciones. La conexión conceptual entre clases se denomina asociación (Figura 2.4). El ellas se puede indicar el

nombre de la relación que refleje el significado de la relación, la dirección es que se da la relación y los roles que juegan cada clase, así como su multiplicidad.

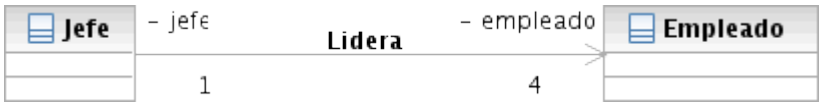


Figura 2.4 Representación de una asociación en UML

El concepto conocido en orientación a objetos como herencia se puede representar en UML mediante la relación de generalización (Figura 2.5). El triángulo sin rellenar señala la clase padre o antecedente de la que se hereda.

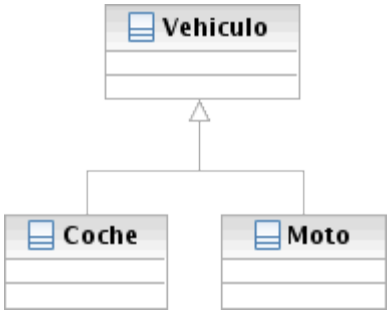


Figura 2.5 Representación de generalización en UML

Para indicar que una clase utiliza a otra se usa una relación de dependencia (Figura 2.6). La flecha señala la clase de la que se depende.

Figura 2.6 Representación de una dependencia en UML

A veces, una clase consta de otras. Este tipo de relación se conoce como agregación. (Figura 2.7). Los componentes y la clase que constituyen son una asociación que conforma un todo. El rombo sin relleno es la clase que representa todo.

Figura 2.7 Representación de una agregación en UML

En las agregaciones, cada componente no tiene porqué pertenecer a un único todo. No es así en las composiciones (Figura 2.8), un tipo especial de agregación en la que cada componente sólo puede pertenecer a un todo. El rombo negro indica el todo de la composición.

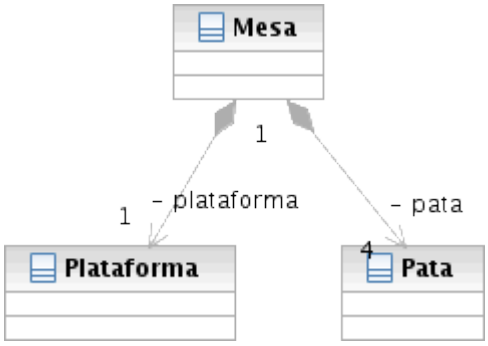


Figura 2.8 Representación de una composición en UML

A veces se quiere especificar una serie de operaciones independientemente de cómo se implementen y quién lo haga. Para capturar este tipo de operaciones reutilizables se utiliza la interfaz (Figura 2.9). Para indicar que una clase implementa una interfaz se utiliza un tipo de relación denominado realización. El triángulo sin relleno indica la interfaz.

Figura 2.9 Representación de interfaz y realización en UML

Así se presenta de forma resumida los principales elementos, y su forma de representación en los modelos sobre los que se trabajará en el presente proyecto.

2.3. Patrones

Cuando se trabaja sobre un problema particular, es común que se utilice la experiencia adquirida anteriormente en la resolución de otros problemas similares para encontrar una solución al nuevo problema. Con la apreciación de continuos problemas recurrentes es como empieza a surgir el concepto de patrones en la ingeniería de software. Así pues, un patrón, es una solución a un problema recurrente que se pueda presentar en distintos ámbitos de la ingeniería de software. Como se comentó en el punto 1.2, los patrones de diseño tienen una gran importancia en el proyecto ASys, pues son uno de los principales mecanismos usados para la explotación de arquitecturas reutilizables.

Primeros patrones

En 1987, Ward Cunningham y Kent Beck, ambos programadores, trabajaban con el *Tektronix's Semiconductor Test Systems Group* que estaba teniendo problemas para terminar un diseño. Decidieron probar el material sobre patrones que habían estado estudiando. Al igual que el arquitecto Christopher Alexander dijo que los ocupantes de un edificio deberían diseñarlo, Ward y Kent dejaron que los representantes de los usuarios terminaran el diseño.

Ward le dio a los a los diseñadores noveles cinco patrones de lenguaje que les ayudaría a aprovechar las ventajas del lenguaje de programación que usaban. Ward y Kent, orgullosos de su trabajo, publicaron un artículo en OOPSLA-87 titulado *Using Pattern Languages for OO Programs.*

No sería hasta principios de la década de 1990 cuando los patrones alcanzarían un gran éxito en el mundo de la informática a partir de la publicación del libro *Design Patterns* escrito por el grupo Gang of Four, compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides, en el que se recogían 23 patrones de diseño comunes.

Uso de patrones

Los patrones comienzan como una idea o la mejor práctica en muchos proyectos. Los patrones se pueden dar en diferentes niveles y asistir en la creación de artefactos de acuerdo con las mejores prácticas en cada etapa del ciclo de vida del desarrollo de la solución.

El objetivo de los patrones es:

- Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores.
- Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Existen distintos tipos de patrones dependiendo del ámbito de desarrollo de sistemas software. Así podemos encontrar:

- Patrones de arquitectura. Expresan esquemas de la organización estructural de sistemas software. Proveen de un conjunto de subsistemas predefinidos,

especifican sus responsabilidades, e incluyen reglas y directrices para organizar las relaciones entre ellos.

- Patrones de diseño. Describen una estructura recurrente de componentes comunicados que resuelve un problema general de diseño en un contexto particular.
- Idiomas. Son patrones de bajo nivel específicos para lenguajes de programación. Describen como implementar aspectos particulares de componentes o relaciones entre ellos con las características de un lenguaje concreto.

En este proyecto se tratará con patrones de diseño y que capturan sistemas de control.

2.4. Model-Driven Development

El proyecto ASys plantea el uso sistemático del desarrollo dirigido por modelos *(Model-Driven Development)*, junto con los elementos descritos en los puntos anteriores, de los que esta metodología se sirve para su funcionamiento. El MDD es un paradigma de desarrollo software apoyado e impulsado por la metodología *Model-Driven Architecture* (MDA) [9], una propuesta de diseño software lanzada por el OMG.

La MDA proporciona un conjunto de líneas a seguir para la estructuración de las especificaciones que son expresadas como modelos. La MDA define la funcionalidad de los sistemas usando un modelo independiente de la plataforma PIM (Platform Independent Model) y un lenguaje específico del dominio DSL (Domain Specific Language). Después, dado un modelo de definición de la plataforma PDM (Platform Definition Model), el PIM se traslada a distintos modelos específicos de la plataforma PSMs (Platform Specific Models) que los computadores pueden correr. Estos modelos pueden usar un DLS o lenguajes de propósito general.

El MDD se centra en las transformaciones de los modelos y la generación de código. Es un estilo de desarrollo software donde los primeros artefactos software son modelos a partir de los cuales se genera código y otros artefactos. Para que esto sea posible, el acceso a los modelos tiene que poder realizarse de forma automatizada, por lo que se introduce el criterio de que un modelo deber ser legible por máquina.

Los modelos en el MDD

Los modelos software se expresan normalmente mediante el lenguaje unificado de modelado UML. El UML es un lenguaje para la especificación, la visualización, y la documentación de sistemas software. Provee de una notación visual y unas semánticas fundamentales para los modelos software. El UML también tiene un formato estándar de serialización legible por máquina, por lo que permite la automatización.

En el MDD, los modelos no se usan sólo como bocetos o planos, sino como artefactos primarios a partir de los cuales implementaciones eficientes se generan mediante la aplicación de transformaciones. En el MDD, los modelos de aplicación orientados en el dominio son el primer foco cuando se desarrollan nuevos componentes software. El código y otros artefactos del dominio de destino se generan usando trasformaciones diseñadas con la participación tanto de expertos en modelado como de expertos en el dominio de destino.

El MDD tiene el potencial de reducir considerablemente el coste del desarrollo de soluciones y mejorar la consistencia y calidad de las mismas. Hace esto automatizando patrones de implementación con transformaciones, las cuales eliminan trabajo repetitivo de desarrollo a bajo nivel. En lugar de aplicar repetidas veces los conocimientos técnicos manualmente al construir artefactos solución, el conocimiento se codifica directamente en las transformaciones. Se consiguen así ambas ventajas, consistencia y mantenibilidad. Una transformación es reaplicada rápidamente para generar artefactos solución que reflejan un cambio en la arquitectura de implementación.

Los patrones en el MDD

Un patrón es una solución a un problema recurrente dentro de un contexto dado. Los patrones encapsulan el tiempo, la habilidad, y el conocimiento de un diseñador para resolver un problema software. Y cuando se usa repetidamente en diferentes proyectos, un patrón se establece como la mejor de las prácticas.

El MDD libera el potencial de los patrones para crear soluciones bien diseñadas, y los patrones proveen el contenido para el MDD.

2.5. Transformaciones en Model-Driven Development

La idea principal de MDD es que el modelo se puede ir transformando en especificaciones más concretas del sistema, hasta, idealmente, resultar en el propio sistema. Son pues conceptos claves en MDD los modelos y las transformaciones.

El término transformación hace referencia a la acción o procedimiento mediante el cual algo se modifica, altera o cambia de forma manteniendo su identidad.

En el ámbito de la ingeniería de software, una transformación es un mecanismo que toma un conjunto de elementos y los cambia a otro nuevo conjunto de elementos de destino. Hay un conjunto de cambios que se pueden configurar. La relación entre los elementes fuente y los de destino está definida en una serie de reglas contenidas en la transformación.

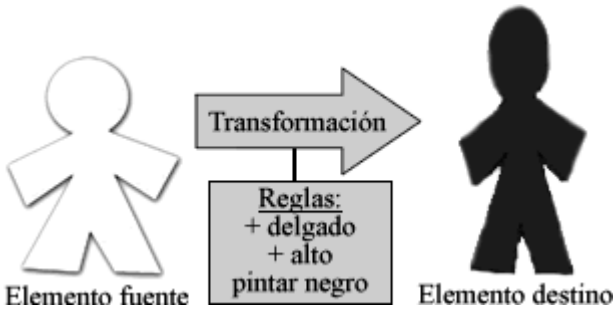


Figura 2.10 Ejemplo de transformación

Las transformaciones te permiten convertir modelos en código, convertir código en modelos, convertir código en código, y convertir modelos en modelos en diferentes niveles de abstracción. Además, las transformaciones pueden implementar patrones para convertir elementos de una forma a otra que sigue el patrón.

2.5.1. Clasificación de los distintos enfoques de las transformaciones de modelos

2.5.1.1. Características de análisis de los distintos enfoques

Para valorar los análisis de los distintos enfoques de las transformaciones de modelos, los analistas proponen un diagrama de características (figura 2.11) [13] con el que documentar los resultados.

El diagrama muestra sólo el nivel superior de las características, de las que se realizará una breve descripción sin entrar a valorar con mayor detalle los subniveles de cada característica.

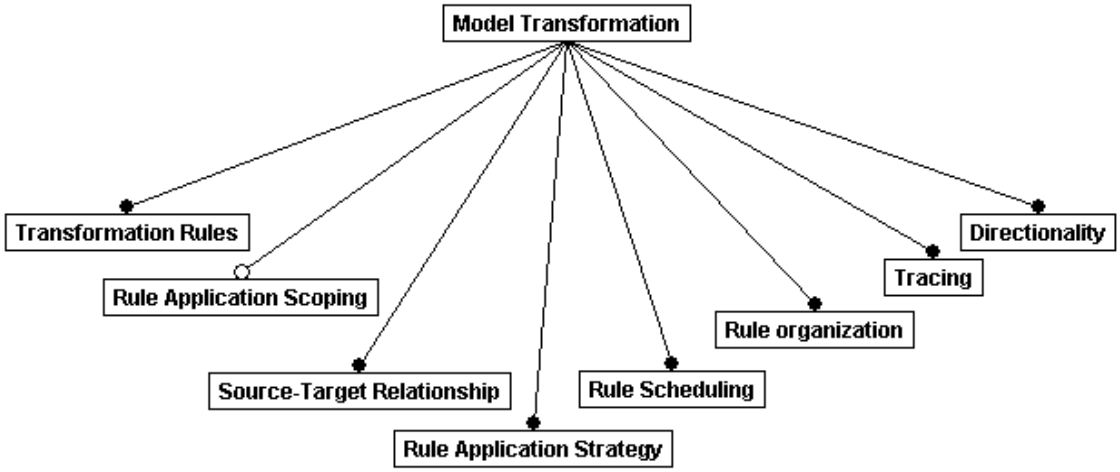


Figura 2.11 Diagrama de características en el análisis de transformaciones de modelos

Reglas de transformación. Una regla de transformación consiste en dos partes: el lado izquierdo y el lado derecho. El lado izquierdo accede al modelo fuente, mientras que el derecho se desarrolla en el modelo objetivo. Ambos lados pueden estar representados por la mezcla de variables, patrones o restricciones.

Cuatro aspectos importantes de las reglas de transformación son:

- La sintáctica de separación. Ambos lados pueden o no estar separados sintácticamente, es decir, la regla de sintaxis debe especificar cada lado como tal o no haber distinción.
- Bidireccionalidad. Una regla puede ser ejecutable en ambas direcciones.
- Parámetros de regulación. Las reglas de transformación pueden tener parámetros adicionales de control que permitan la configuración y la afinación.
- Estructuras intermedias. Algunos enfoques requieren la construcción de estructuras intermedias.

Ámbito de aplicación de la regla. Permite a una transformación restringir las partes del modelo que participan en la transformación. Algunos enfoques soportan un ámbito de aplicación flexible donde se puede configurar un ámbito pequeño dentro de un modelo sin que afecte a la totalidad.

Relación entre la fuente y el destino. Algunos enfoques encargan la creación de un nuevo modelo de destino que tiene que estar separado de la fuente. En algunos otros, la fuente y el destino son siempre el mismo modelo.

Estrategia de aplicación de reglas. Una regla necesita ser aplicada en una localización específica dentro de su ámbito fuente. Dado que puede haber más de una coincidencia para una regla dentro de su ámbito de aplicación, se necesita una estrategia de aplicación. Ésta puede ser determinista, no determinista o interactiva.

Planificador de reglas. El mecanismo planificador determina el orden en que las reglas individuales se aplican. Puede variar en cuatro áreas:

- Forma. Los aspectos de planificación se pueden expresar implícitamente o explícitamente.
- Selección de reglas. Las reglas pueden ser seleccionadas por una condición explícita. Algunas permiten una elección no determinista. Alternativamente, se puede proveer un mecanismo de resolución de conflictos basado en prioridades.
- Iteración de las reglas. El mecanismo de iteración incluye recursividad, bucles e iteración de punto fijo.
- Ajuste de fase. El proceso de transformación puede estar organizado en varias fases donde cada fase tiene un propósito específico y sólo algunas reglas pueden ser invocadas.

Organización de las reglas. Hace referencia a la composición y estructuración de múltiples reglas de transformación. Se consideran tres áreas de variación en este contexto:

- Mecanismos de modularidad. Algunos enfoques permiten empaquetar reglas en módulos. Un módulo puede importar otro módulo para acceder a su contenido.
- Mecanismos de reutilización. Ofrecen una forma de definir una regla basada en una u otros más reglas.
- Estructura organizacional. Las reglas se pueden organizar de acuerdo con la estructura del lenguaje fuente o del lenguaje de destino, o pueden tener su propia organización independiente.

Enlaces de trazabilidad. Las transformaciones pueden registrar enlaces entre sus elementos fuente y los de destino. Estos enlaces pueden ser útiles en la realización de un análisis de impacto, en la sincronización entre modelos y en la determinación del destino de una transformación.

Direccionalidad. Las transformaciones pueden ser unidireccionales o bidireccionales.

2.5.1.2. Enfoques de transformaciones de modelo a código

Las enfoques modelo a código son uno de los dos tipos generales de enfoque entre los que distinguimos (modelo a modelo y modelo a código). En general, se pueden ver como un tipo especial de transformaciones modelo a modelo; sólo es necesario proveer de un metamodelo al lenguaje de programación de destino. Sin embargo, por razones prácticas de reutilización de las tecnologías de compilación, el código se genera siempre como texto.

En la categoría de modelo a código, se puede distinguir entre enfoques basados en visitantes y enfoques basados en plantillas:

Enfoques basados en visitantes. Un enfoque de generación de código muy básico consiste en proveer algunos mecanismos visitantes para cruzar la representación interna del modelo y escribir código en flujos de texto. Un ejemplo de esta aproximación es Jamda, que es un framework orientado a objetos que provee de un conjunto de clases para representar modelos UML, un API para manipular modelos, y un mecanismo visitante para generar código.

Enfoques basados en plantillas. La mayoría de las actuales herramientas disponibles en MDA soportan la generación de modelo a código basada en plantillas. Por ejemplo, b+m Generator Framework, JET, Codagen Architect, AndroMDA, ARCStyler, OptimalJ y XDE.

Una plantilla consiste normalmente en texto que contiene empalmes a metacódigo para acceder a información desde la fuente y realizar la selección de código y la expansión iterativa. El lado izquierdo usa lógica ejecutable para acceder a la fuente; el lado derecho combina patrones de cadenas sin tipo con lógica ejecutable para la selección del código y la expansión iterativa, y no hay separación sintáctica entre ambos lados. Los enfoques de plantillas normalmente ofrecen planificadores definidos por el usuario en la manera interna de llamar a la plantilla desde dentro de otra.

El acceso mediante lógica del lado izquierdo se puede hacer de diferentes formas. La lógica puede ser simplemente código JAVA que accede a la API provista por la representación interna del modelo fuente (ej., JMI), o podrían ser peticiones declarativas (ej., en OCL o XPath).

2.5.1.3. Enfoques de transformaciones de modelo a modelo

Las transformaciones de modelo a modelo trasladan entre los modelos fuente y los modelos de destino, los cuales pueden ser instancias de los mismo o distintos metamodelos, Todos estos enfoques soportan sintaxis tipificada de variables y patrones.

Podemos encontrar los siguientes diferentes enfoques de transformaciones modelo a modelo:

Enfoques de manipulación directa. Estos enfoques ofrecen una representación interna del modelo más algunas APIs para manipularla. Normalmente, se implementan con un framework orientado a objetos que puede proveer de alguna mínima infraestructura para organizar las transformaciones. Sin embargo, los usuarios tienen que implementar las reglas de transformación y planificarlas en su mayoría usando un lenguaje de programación como Java.

Enfoques relacionales. Esta categoría agrupa enfoques declarativos donde el principal concepto es las relaciones matemáticas (ej., AK02, QVTP, CDI, enfoques declarativos en GLR+02, y reglas de asignación en AST+1).

La idea básica es declarar los tipos de elementos fuente y de destino con una relación y especificarla usando limitaciones. En su forma pura, esta especificación no es ejecutable. Sin embargo, las limitaciones declarativas se pueden dar en semánticas ejecutables, como en la programación lógica.

Enfoques basados en transformaciones gráficas. Esta categoría se basa en el trabajo teórico en transformaciones gráficas. En particular, estos enfoques operan sobre gráficos con tipos, atribuidos y etiquetados, que son un tipo de gráficos especialmente diseñados para representar UML. Ejemplos de estos enfoques de transformaciones gráficas a transformaciones de modelos incluyen VIATRA, ATOM, GreAT, UMLX, and BOTL.

Las reglas de transformaciones gráficas consisten en un patrón gráfico en cada lado de la transformación. El patrón gráfico puede presentar en la sintaxis concreta de sus respectivos lenguajes fuente y de destino, o con el estándar de metaobjetos MOF (Meta-Object Facility). De forma similar a los enfoques relacionales, los enfoques de transformaciones gráficas son capaces de expresar transformaciones de modelos de una manera declarativa.

Enfoques basados en estructuras. Esta categoría tiene dos fases distintas: la primera concerniente con la creación de una estructura jerárquica del modelo de destino, mientras que la segunda fase ajusta los atributos y las referencias en el destino. El framework en su totalidad determina la planificación y la estrategia de aplicación; los usuarios solo se preocupan de proveer las reglas de transformación.

Un ejemplo de enfoque basado en estructuras es el framework de transformaciones modelo a modelo provisto por OptimalJ.

Enfoques híbrido. Los enfoques híbridos combinan diferentes técnicas de las categorías previas.

El lenguaje de reglas de transformación (TRL) es una composición de los enfoques imperativos y los declarativos. Algunas similitudes tiene el lenguaje de transformación Atlas (ATL) que puede ser completamente declarativo, híbrido, o imperativo en su totalidad.

XDE es un ejemplo de un enfoque altamente híbrido. XDE soporta transformaciones modelo a modelo a través de su mecanismo de patrones. La mezcla del concepto de colaboración parametrizada, que es el mecanismo de UML para modelar patrones de diseño, con el objetivo de transformaciones modelo a modelo envuelve el mecanismo de patrones en una mayor complejidad haciéndolo un enfoque híbrido.

Otros enfoques de modelo a modelo. El framework de transformaciones definido en el Common Warehouse Metamodel del OMG, y las transformaciones usando XSLT.

2.6. Definición de patrones de diseño en Model-Driven Development

La integración de los patrones de diseño en un lenguaje de modelado es una idea atractiva. Un elemento conceptual simple de modelado que permita indicar explícitamente las clases de un patrón de diseño podría ayudar a los diseñadores en muchos sentidos. Además de la ventaja directa de una mejor documentación y la consecuente mejora en el entendimiento de un modelo, señalando la existencia de un patrón de diseño, los diseñadores pueden abstraerse de detalles de diseño conocidos y concentrarse en tareas más importantes.

2.6.1. Modelado de la estructura de un patrón de diseño usando colaboraciones parametrizadas

El lenguaje unificado de modelado UML ha mejorado el concepto de diseño *colaboración* para proveer de un mejor soporte para el diseño de patrones. De hecho, los dos niveles conceptuales provistos por las colaboraciones (colaboraciones parametrizadas y usos de colaboración) encajan perfectamente para el modelado de patrones de diseño. A nivel general, una colaboración parametrizada es capaz de representar la estructura de la solución propuesta por el patrón, que es enunciada en términos genéricos. La aplicación de la solución en un contexto particular (instancia de un patrón) puede ser representada por usos de colaboración.

La propuesta oficial de UML: Colaboraciones parametrizadas.

De acuerdo con el manual de usuario de UML, se requiere seguir tres pasos para modelar un patrón de diseño:

1. Identificar una solución común para el problema común y materializarlo como un mecanismo.
2. Modelar el mecanismo como una colaboración, es decir, un espacio de nombres que contiene su estructura, además de sus aspectos de comportamiento.
3. Identificar los elementos del patrón de diseño que deben estar ligados a elementos en un contexto específico e interpretarlos como parámetros de la colaboración.

Una colaboración se define en términos de roles. Los aspectos estructurales de una colaboración se especifican usando roles de clasificador, los cuales son marcadores de posición para objetos que interactuarán para lograr los objetivos de la colaboración. Como un marcador de posición, un rol es similar a una variable libre o a un parámetro formal de una rutina. Más tarde se ligará a un objeto que se ajuste al rol de clasificador. Varios objetos pueden jugar un rol dado en tiempo de ejecución y cada uno de ellos debe ajustarse al rol de clasificador. Los roles de clasificador están conectados por roles de asociación, que son marcadores de posición para asociaciones entre objetos.

La manera en que se define como un objeto se ajusta a un rol específico es particularmente interesante, y aquí es donde la noción de la base de un rol interviene. Un rol de clasificador no especifica exhaustivamente todas las operaciones y atributos que un objeto que se ajuste a este rol debería tener. Únicamente las características estrictamente necesarias para la realización del rol se especifican como características disponibles en el rol de clasificador. Por lo tanto, el rol de clasificador se puede ver como una restricción del ajuste del clasificador “completo” del objeto al subconjunto de características necesitadas. Realmente, el UML impone que los roles se definan sólo como una restricción de los clasificadores existentes y hay reglas OCL en el

metamodelo que hacen que se cumpla este criterio. El clasificador(es) del cual un rol de clasificador es una restricción se llama la base(s) del rol.

Se dice que un objeto se ajusta a un rol particular si provee todas las características necesitadas para jugar este rol, que son todas las características declaradas en el rol del clasificador. Aunque no se requiere estrictamente, cualquier objeto que sea una instancia de un clasificador base del rol se ajustará por definición a este rol.

La reutilización de un patrón de diseño se expresa usando el mecanismo de genericidad: para hacer la descripción de un patrón de diseño independiente del contexto, y por lo tanto reutilizable, los clasificadores base de cada rol se convierten en parámetros de plantilla de la colaboración. Poner un patrón en un contexto consiste simplemente en enlazar la colaboración de la plantilla al modelo del usuario proveyendo argumentos actuales para los parámetros de la plantilla.

Sin embargo, las colaboraciones parametrizadas no se pueden enlazar con cualquier clase actual. Las clases participantes deben respetar las restricciones fundamentales del patrón. La colaboración puede imponer distintos tipos de restricciones usan un conjunto de elementos de restricción:

- Una generalización entre dos clasificadores base (formales y genéricos) significa que una relación de herencia similar debe existir también entre los dos correspondientes clasificadores actuales en cada enlace de la colaboración. Esta es sólo la versión de UML de genericidad restringida. Sin embargo, la relación de herencia entre dos clasificadores actuales impuesta por dicha restricción no necesita ser directa.

- Una asociación entre dos clasificadores base (formales y genéricos) significa que una asociación similar debe existir también entre los dos correspondientes clasificadores actuales usados en cualquier enlace de la colaboración. Es muy probable, que estas asociaciones entre clasificadores base actúen como bases para roles de asociación dentro de la colaboración.

Limitaciones de las colaboraciones parametrizadas

Sin embargo, existen serias limitaciones al poder expresivo de elementos restrictivos asociados a la colaboración.

Restricciones sobre las generalizaciones: La representación gráfica de una colaboración superpone los roles de clasificador con sus correspondientes bases, los cuales hacen ambiguo el uso de flechas de generalización. En contra de las relaciones de generalización entre sus respectivas bases, las relaciones de generalización entre los roles de clasificador en sí mismas no traen restricciones suplementarias: simplemente significan (como de costumbre) que el rol de hijo hereda todas las características de sus padre(s).

Restricciones sobre las asociaciones. La documentación UML no considera de utilidad hacer las bases de los roles de asociación parámetros de plantilla de la colaboración. Asume que las bases de los roles de asociación se pueden deducir automáticamente de las asociaciones existentes entre los clasificadores base dados cuando la plantilla de colaboración está ligada. Sin embargo, hay casos en los que este supuesto no se mantiene: cuando hay varias asociaciones candidatas, o cuando no hay asociaciones (directas). El primer caso fuerza una elección arbitraria, mientras que último caso requiere la creación de una asociación <<derivada>> para proveer el acceso directo necesario a través de asociaciones (indirectas).

Restricciones sobre las características disponibles. Para tomar parte en una colaboración, un rol debe disponer de un conjunto de características disponibles, que toma de sus clasificadores base. Pero si la propia base es un parámetro genérico, ¿de dónde provienen las características disponibles? Se pueden plantear diversas posibilidades:

1. Definir un clasificador de propósito especial asociado con la plantilla de colaboración. Este clasificador ofrecería las características necesitadas por un rol dado, y una relación de generalización de restricción aseguraría que el clasificador actual usado como base para este rol es una subclase de esta clase especial.

2. Definir parámetros de plantilla compuesto y reglas asociadas de conformidad para los correspondientes argumentos actuales en un enlace. Las características disponibles provendrán entonces de dentro del parámetro de plantilla.

3. Tornando todas las características necesitadas a parámetros de plantilla (el UML no impone ninguna restricción en el tipo de entidades permitidas como parámetros de plantilla). Se necesitaría entonces un conjunto de limitaciones suplementarias para garantizar que los argumentos actuales para las características son realmente propiedad del argumento actual apropiado para los clasificadores base. Se añadirían restricciones en escritas en OCL en el conjunto de los elementos restrictivos de la colaboración.

Restricciones relativas a cualquier número de clasificadores. De alguna manera, se puede tener la necesidad de enlazar varias clases del modelo del usuario a un único rol de un patón de diseño. Habrá que expresar restricciones sobre el patrón que necesitan capacidad reflexiva completa. Dar expresiones OCL que accedan al metanivel (que normalmente no es accesible para los modelos del usuario) es una manera prometedora de resolver este asunto. La metaclase estándar y los estereotipos UML podrían también resultar útiles en el metanivel.

Restricciones temporales o de comportamiento. La mera existencia de operaciones o atributos necesarios para jugar los roles no es suficiente. Los patrones también establecen como se organizan las llamadas a operaciones, actualizaciones de atributos u otras acciones para lograr una tarea particular. Las interacciones UML se pueden ligar a una colaboración especificando sus aspectos de comportamiento. Una interacción es una petición parcial sobre mensajes y acciones relacionada con el objetivo, y se puede representar gráficamente como un diagrama de secuencia.

Al ser parte de la colaboración parametrizada, las interacciones se ven involucradas en los procesos de enlazado, aunque el UML no define cómo. Hay dos maneras en que las interacciones pueden afectar potencialmente al enlazado:

1. La interacción en la plantilla se podría incorporar tal y como está en el modelo resultante, con los parámetros de plantilla substituidos con argumentos actuales del enlace.

2. Un planteamiento más razonable es considerar las interacciones como un nuevo tipo de restricción que debe ser respetada por los argumentos actuales. Por supuesto, los participantes actuales en el enlace pueden satisfacer las restricciones de forma más o menos directa como consecuencia de su propia estructura de comportamiento.

Está claro pues que las interacciones ligadas a colaboraciones se deben interpretar como restricciones de comportamiento sobre los participantes, no muy diferentes de fórmulas lógicas temporales.

Por un lado, las diferentes limitaciones presentadas hace imposible para las colaboraciones parametrizadas de UML especificar con más precisión algunas de las restricciones más interesantes de los patrones de diseño. Su capacidad expresiva está limitada a la descripción de vínculos de asociación y generalización, y en cierta medida, a la disponibilidad de características. Sin lugar a dudas, restricciones más sofisticadas necesitan acceder al metamodelo UML

Por otro lado, la estructura estática de las colaboraciones implica muchas opciones que no son fundamentales para el propia patrón, pero son específicas de algunas de sus soluciones reificadas. Esto previene a las colaboraciones UML de representar sólo la esencia de un patrón, libre de cualquier opción prematura. Todos los diagramas de representación de patrones o los sucesos de patrones en la literatura UML no alcanzan este objetivo debido a los efectos secundarios de la sobre especificación de las colaboraciones.

2.6.2. Transformaciones de modelos basadas en patrones usando perfiles de UML 2

Aunque que existen muchos lenguajes de transformaciones disponibles para UML, todos comparten la discrepancia entre la sintaxis del lenguaje de especificación de las transformaciones y la sintaxis visual de UML. Las transformaciones de modelos se definen, o bien de manera textual, o en un lenguaje que usa elementos constructivos del metamodelo subyacente.

La siguiente propuesta que se presenta [15] tiene como objetivo especificar las transformaciones de modelos como patrones en la sintaxis concreta de UML 2. Estos patrones son más fáciles de leer que las especificaciones usuales en las transformaciones y usa sólo elementos conceptuales de UML 2. Esto se logra usando el mecanismo de extensión incorporado de UML 2, el *perfil*. Además de la especificación, estos perfiles ofrecen la aplicación de patrones dentro de cualquier herramienta de modelado compatible.

Perfiles de UML 2.0

El paquete Profiles (perfiles) de UML 2.0 define una serie de mecanismos para extender y adaptar las metaclases de un metamodelo cualquiera a las necesidades concretas de una plataforma o de un dominio de aplicación.

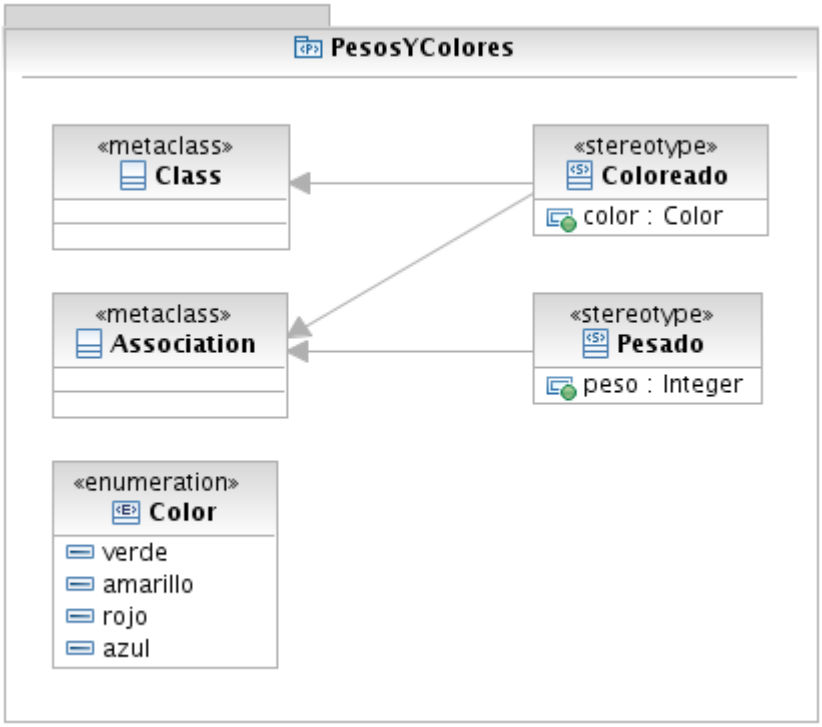


Figura 2.12 Ejemplo de paquete de un perfil

Un perfil se define en un paquete UML, estereotipado «profile», que extiende a un metamodelo o a otro perfil. Hay tres mecanismos que se utilizan para definir perfiles: estereotipos, restricciones y valores etiquetados. Sólo nos interesa para este apartado lo referente a estereotipos.

Un estereotipo viene definido por un nombre, y por una serie de elementos del modelo sobre los que puede asociarse. En el ejemplo de perfil PesosYColores representado en la Figura 2.12, se definen dos estereotipos, Coloreado y Pesado, que proporcionan color y eso a un elemento UML. El perfil especifica los elementos del metamodelo de UML sobre los que se pueden asociar los estereotipos. Sólo las clases y las asociaciones pueden colorearse, y sólo las asociaciones pueden tener un peso.

Uso de perfiles para la especificación de transformaciones

Para explicar la idea principal de esta propuesta, se presenta en primer lugar, la especificación de un patrón de diseño sencillo, el *singleton* (instancia única).

La Figura 2.13 muestra la clase Demo de dos maneras distintas. En el lado izquierdo anotada con el estereotipo Singleton que indica que esa clase es, o debería ser, una clase de instancia única. La clase en el lado derecho posee todos los elementos (atributo, constructor, método) que hacen a una clase una clase de instancia única.

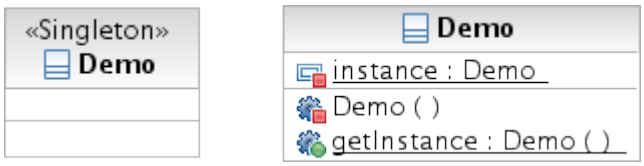


Figura 2.13 Clase con el estereotipo singleton y clase con los elementos de instancia única

El lado izquierdo muestra la aplicación, a través del estereotipado, del patrón singleton y el derecho el patrón expandido. Así, las clases se pueden ver como la fuente y el objetivo de este patrón. Lo que falta es la forma de describir la transición desde la fuente al modelo objetivo con los conceptos estándar del UML 2.0.

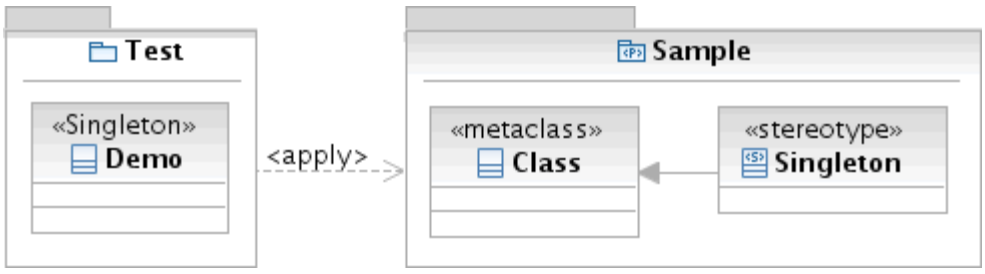


Figura 2.14 Perfil para Singleton y la aplicación de este perfil

El primer paso es definir un perfil que permita la anotación de un estereotipo a una clase. La Figura 2.14 muestra este perfil y la aplicación de este estereotipo. Ahora el

modelo está bien formado al tener un perfil que define el estereotipo usado y que este estereotipo extiende a una clase UML. Pero este modelo no da pistas sobre las transformaciones necesarias para expandir el patrón singleton.

La idea de especificar estas transformaciones es usar la clase expandida singleton de la Figura 2.13 y anotar todos los elementos que deben ser añadidos. Este tipo de anotaciones se puede realizar con estereotipos, mientras que estos estereotipos representan modificaciones del modelo.

La Figura 2.15 muestra un modelo UML 2.0 que usa un cierto perfil en el que todos los estereotipos extienden el elemento más alto del metamodelo, *Element.* Por lo tanto, cualquier elemento UML 2.0 del modelo puede ser anotado con estos estereotipos. El estereotipo New representa la adición de un nuevo elemento en el modelo y This especifica el elemento de referencia del patrón. El paquete Singleton contiene la clase expandida con los estereotipos anotados para describir el rol de los elementos del modelo en el patrón. El elemento de referencia es la propia clase y los otros tres elementos se deben añadir si el patrón está expandido.

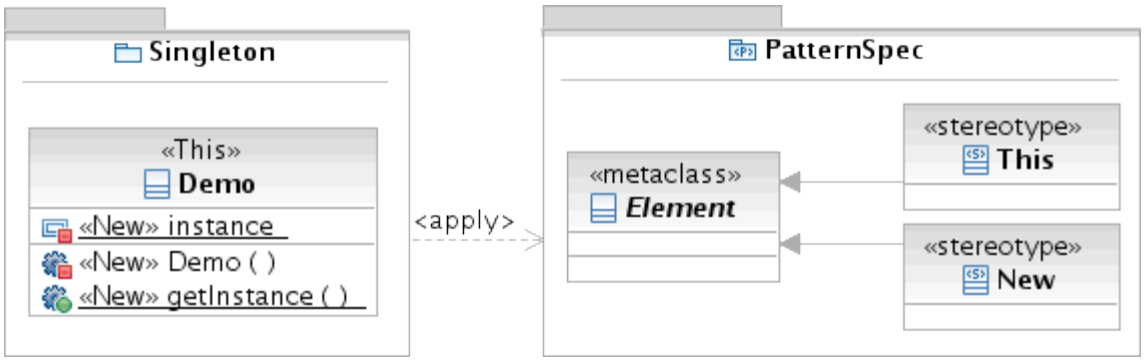


Figura 2.15 Perfil para especificación del patrón y aplicación como una definición de instancia única

El paquete Singleton se puede ver como una especificación del patrón singleton ya que la estructura y el comportamiento del patrón se especifican. El perfil de la Figura 2.14 se puede extraer de este paquete. Ya que el elemento de referencia es una clase y el nombre del paquete es Singleton, podemos generar automáticamente el perfil mostrado en la Figura 2.14.

Además, podemos extraer automáticamente las reglas de transformación del paquete. Para cada elemento que es anotado con New creamos una regla de transformación que añade el elemento al modelo. Estos dos pasos, creación del perfil y creación de las reglas de transformación, pueden ser realizadas por un programa. La Figura 2.16 muestra los dos tipos de datos que este tipo de programa produciría.

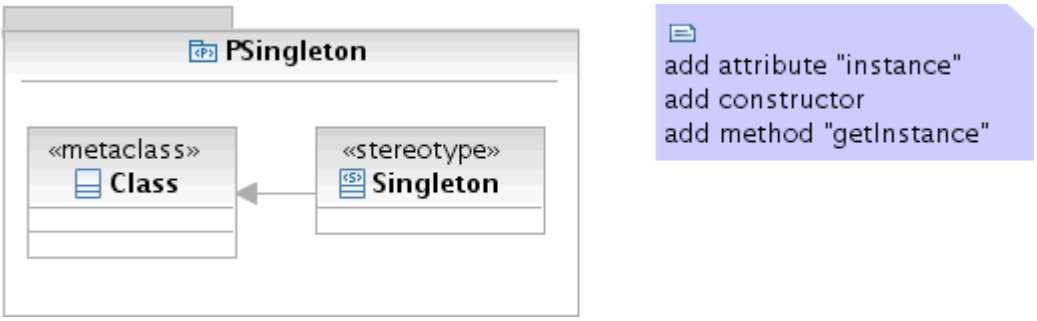


Figura 2.16 Perfil y transformaciones generadas

En el lado izquierdo, el perfil que contiene el estereotipo para la aplicación del patrón. Constatar que la extensión de la metaclase Class restringe la aplicación del estereotipo a clases UML. Así que el perfil ofrece un tipo simple de validación. Las transformaciones se muestran en el lado derecho.

Capítulo 3

Herramientas y entorno de desarrollo

3.1. Rational Software Architect

IBM Rational Software Architect, desarrollado por la división Rational Software de IBM, es un exhaustivo entorno de modelado y desarrollo que usa el UML para el diseño de arquitecturas para C++ y aplicaciones y servicios webs con Java 2 EE. La aplicación está basada en el framework software de código abierto de Eclipse e incluye capacidades enfocadas en el análisis de código de arquitecturas, C++, y Model-Driven Development con el UML para crear aplicaciones resilientes y servicios webs.

Figura 3.1 Logo de Rational Software

Durante la realización del proyecto se ha usado la versión 7.5 siendo vinculante con el correcto funcionamiento de la herramienta desarrollada. La justificación de la elección de RSA para la realización del proyecto, además de ser el entorno de desarrollo basado en modelos elegido para ASys, es que:

- Es una herramienta basada en Eclipse: permite extenderla, y la propia herramienta ofrece utilidades y el entorno de desarrollo para hacerlo
- Es uno de los entornos más avanzados para MDD, con herramientas para el modelado que soportan UML versión 2.1, y transformaciones modelo a código y código a modelo.

UML to Java	UML to C#
UML to C++	UML to EJB
UML to WSDL	UML to XSD
UML to CORBA	UML to SQL (IDL)
Java to UML	C++ to UML

- Incluye todas las capacidades de IBM Rational Application Developer
- Permite la gestión de modelos para desarrollo paralelo y refactorización de arquitecturas, por ejemplo, combinar, comparar y unir modelos y fragmentos de modelos.
- Provee de herramientas visuales de construcción para facilitar el diseño y el desarrollo software.

3.2. Entorno de modelado de RSA

Un modelo describe aspectos de un sistema mediante el uso de diferentes diagramas, elementos o modelos. El propósito del producto determina los tipos de diagramas, modelos y elementos a usar.

En el entorno de modelado de RSA, se maneja y se trabaja con todos los modelos de la misma manera. Por ejemplo, puedes establecer la configuración predeterminada para modelos UML especificándola en las preferencias, o crear y poblar modelos con diferentes diagramas para especificar aspectos o comportamientos del sistema.

Modelos

En la ventana de trabajo se crean proyectos de modelado, y en un mismo proyecto se puede añadir distintos modelos y elementos para describir un sistema o varios. Los modelos usados en el presente proyecto son modelos UML que al crearse en el entorno de trabajo de RSA se guardan como archivos EMX con la extensión .emx.

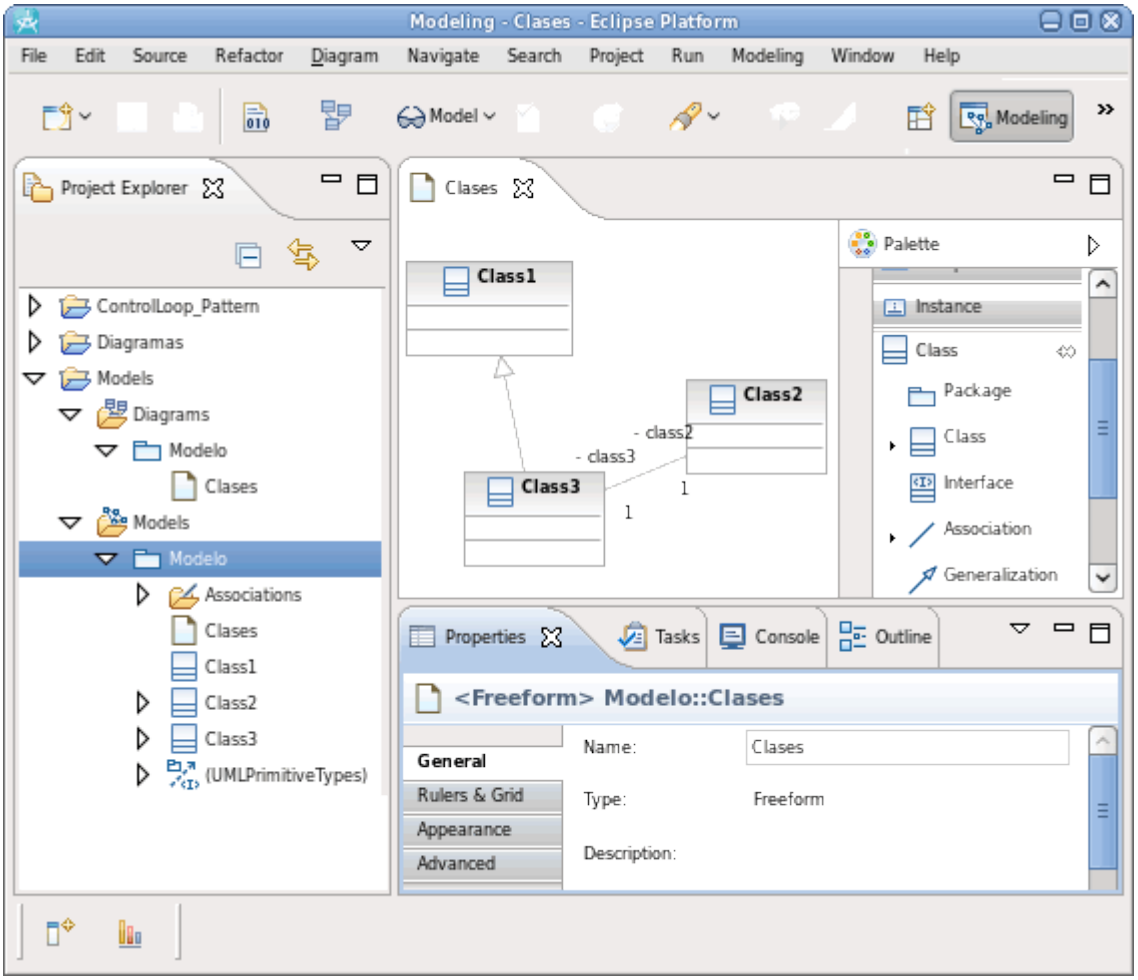


Figura 3.2 Perspectiva de modelado de RSA

RSA permite compartir modelos UML y otros recursos dentro o a través de equipos de trabajo importando y exportando estos modelos. Puedes importar modelos UML 2.1 (.uml) y modelos de intercambio XMI (.xmi) como modelos EMX, o exportar modelos EMX como modelos UML 2.1 o modelos de intercambio XMI.

Los modelos UML pueden ser extensos y complejos, particularmente en entornos de equipo donde los modelos son compartidos y editados por diferentes personas. Para facilitar el trabajo con modelos grandes, se pueden guardar porciones de los modelos como fragmentos, que son subunidades del modelo guardadas en archivos separados. Cada fragmento se guarda en un archivo EFX (.efx). Los fragmentos pueden contener referencias cruzadas a otros fragmentos, y si ya no se requieren, se pueden retornar al modelo padre.

Diagramas

Los diagramas UML ilustran los aspectos cuantificables de un sistema que puede ser descrito visualmente, tales como relaciones, comportamiento, estructura o funcionalidad. En la perspectiva de modelado de RSA, a cada modelo se le puede añadir

diferentes diagramas para describir estos aspectos. Los diagramas de clases, diagrama de casos de uso, diagrama de estados, diagrama de secuencias, diagrama de componentes,… son algunos ejemplos de los diagramas soportados.

Para crear los diagramas se dispone del editor de diagramas; y para añadir elementos y relaciones entre ellos, se usa la pestaña de creación de la paleta (Figura 3.2). Para explorar un modelo desde un diagrama, se puede unas la pestaña de exploración de la paleta.

Los contenidos de los modelos con semántica UML están sincronizados con los correspondientes diagramas por defecto. Este comportamiento canónico supone que cualquier cambio que se realiza en el modelo se refleja en los diagramas, y que cualquier cambio realizado en los diagramas queda reflejado en el modelo.

3.2.1. Perfiles UML 2.0

Los productos de modelado Rational de IBM soportan diferentes mecanismos que pueden ser usados para extender la funcionalidad del entorno de modelado. Por ejemplo, usando los puntos de ampliación y API disponibles, se pueden crear perfiles personalizados para extender el metamodelo UML, crear pluglets para hacer extensiones menores al banco de trabajo, extender la funcionalidad de modelado, o añadir funcionalidad a una transformación existente.

Si se está modelando un sistema y se necesita extender el metamodelo UML para un uso particular, en lugar de cambiar el metamodelo, se pueden crear un conjunto de estereotipos y restricciones, y agruparlos en un perfil personalizado. Después, ese perfil puede ser aplicado a distintos modelos, y usar los estereotipos y restricciones para marcar el modelo para una plataforma o dominio específico.

Creación de perfiles en RSA

Cuando se crea un perfil UML en RSA, éste puede ser añadido a un proyecto de modelado o de perfiles existente. También se puede crear un proyecto de perfiles nuevo si se quiere manejar el perfil por separado.

Se pueden crear y diseñar perfiles UML mediante diagramas de clases, los cuales proporcionan una representación visual, una aproximación de modelado para crear perfiles que es particularmente útil cuando se crear relaciones. Aunque los perfiles se pueden crear y modificar en la vista del explorador de proyectos, el modelado de perfiles es un método alternativo que puede resultar más rápido y sencillo.

Después de crear un proyecto de perfil UML personalizado o añadir un perfil a un proyecto existente, deberán crearse los estereotipos y las restricciones que se quieran incluir en el perfil (Figura 3.3).

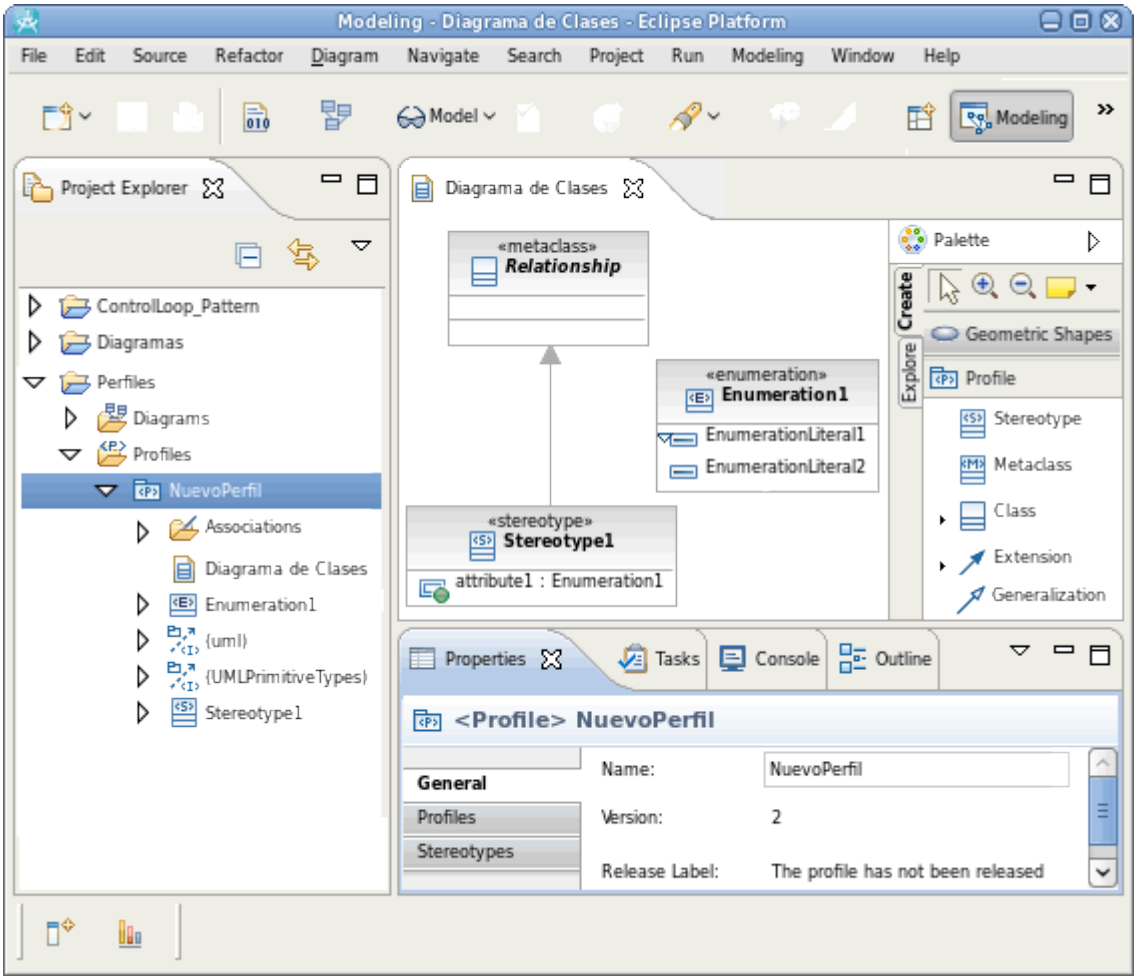


Figura 3.3 Ejemplo de modelado de un perfil en RSA

3.3. Entorno Java de RSA

El proyecto Herramientas de desarrollo Java (JDT) proporciona los plugins de herramientas que implementan un IDE de Java que soporta el desarrollo de cualquier aplicación Java, incluyendo plugins para el propio Eclipse. Añade una naturaleza de proyecto Java y una perspectiva Java al entorno de trabajo de Eclipse, en el que se basa RSA, así como varias vistas, editores, asistentes, constructores y herramientas de fusión de código y refactorización.

El proyecto JDT permite que Eclipse sea un entorno de desarrollo por sí mismo. El editor visual Java es una herramienta que permite construir visualmente interfaces gráficas de usuario basadas en Swing, SWT o AWT. La característica de herramientas Protocolo de inicio de sesión (SIP) proporciona un entorno de desarrollo para la creación de servicios nuevos basados en SIP. La característica mejora la perspectiva de desarrollo de Edición de empresa de Plataforma Java EE existente para permitir la creación de aplicaciones SIP y HTTP/SIP convergentes. La perspectiva Java EE mejorada incluye también un soporte para el formato de archivado (SAR) e incluye un

asistente para editar descriptores de despliegue SIP. Puede empaquetar archivos de archivado SAR en un archivo de archivado de aplicación Java EE, igual que otros componentes de Java EE.

JDT ayuda en la construcción y ejecución optimizada de programas Java. Se pueden usar análisis automáticos de descubrimiento de arquitecturas y de códigos estructurales para encontrar patrones en las aplicaciones desarrolladas, o para asegurar que se siguen reglas estructurales definidas. También, se puede hacer uso del editor de diagramas UML para explorar y editar el código en una representación UML de éste.

Perspectivas java

El proyecto JDT contribuye con las siguientes perspectivas al banco de trabajo:

- Java: Diseñada para trabajar con los proyectos Java. Consiste en un editor y las vistas Package Explorer, Hierarchy, Outline, Problems, Javadoc, y Declaration.

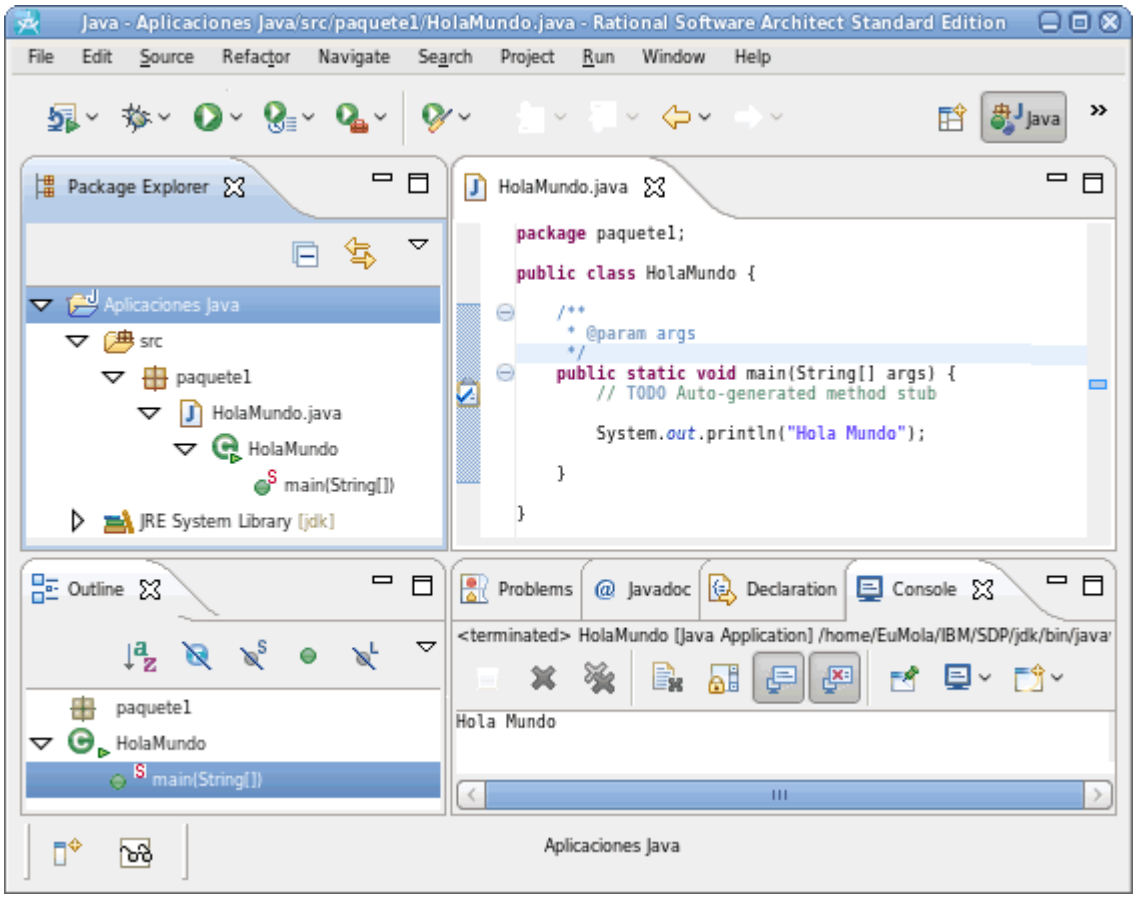


Figura 3.4 Perspectiva Java de RSA

- Java Browsing: Diseñada para navegar la estructura del proyecto Java. Consiste en un editor y las vistas Projects, Packages, Types y Members.

- Type Hierarchy: Diseñada para explorar una jerarquía de tipos. Consiste en la vista Hierarchy y el editor.

- Debug: Diseñada para depurar los programas java. Incluye un editor y las vistas Debug, Breakpoints, Variables, Expressions, Outline, Console, Tasks.

3.3.1. Pluglets RSA

El pluglet es uno de los mecanismos, nombrados en el punto 3.2.1, para ampliar el entorno de modelado de Rational. Son pequeñas aplicaciones Java usadas para hacer extensiones menores al entorno de trabajo de manera simple y sencilla. Los pluglets están escritos en Java y residen en proyectos de pluglet. Los creadores de pluglets pueden usar el entorno de desarrollo Java y acceder a las APIs de los plugins del entorno de trabajo para extender e implementar los pluglets existentes.

Un archivo fuente de un pluglet no es diferente de cualquier otro archivo fuente de Java. Los escritores de pluglets pueden especificar qué plugins requieren sus pluglets, y esto controla el classpath, durante el compilado y la ejecución. La especificación de los plugins se realiza en un manifiesto de plugin en el banco de trabajo, el archivo *plugin.xml*, como se muestra a continuación:

```
<pluglets>
   <require>
      <import plugin="com.ibm.xtools.pluglets"/>
   </require>
</pluglets>
```

Los pluglets permiten la automatización de tareas repetitivas mediante Java. Un beneficio de los pluglets frente a los plugins es que se pueden ejecutar dentro de la misma sesión del banco de trabajo en el que están siendo creados. Una limitación es que los pluglets no ofrecen soporte para la depuración.

3.3.2. Plataforma de modelado de Rational

La plataforma de modelado de Rational es la base sobre la que se basan las soluciones de modelado de Rational. Su característica más visible consiste en un modelador de UML con editores de modelado, vistas, y herramientas que están construidas mediante varios servicios ofrecidos por la plataforma. Esta plataforma está basada en la tecnología Eclipse.

Los componentes de la plataforma de modelado de Rational abarcan tres capas:

- **UML Modeler:** la capa UML Modeler incluye las clases e interfaces requeridas para manejar el entorno de modelado UML de la plataforma. La clase *UML Modeler* es el punto de entrada para controlar el ciclo de vida de los modelos UML y perfiles del modelador.

- **UML Modeling layer:** La capa UML Modeling incluye componentes de ayuda para trabajar con los modelos y diagramas UML. Por ejemplo, para localizar elementos dentro de los modelos.

- **Domain Modeling Layer:** La capa Domain Modeling incluye varios servicios útiles para la producción de arbitraria de editores de modelado basados en el Framework de modelado de Eclipse.

La combinación de los pluglets con las clases facilitadas por la plataforma de modelado de Rational permite el desarrollo de pequeñas aplicaciones para explorar y editar modelos en el entorno de trabajo de RSA. Habrá que especificar en el manifiesto del pluglet el plugin “com.ibm.xtools.modeler.ui”.

3.3.3. Librería Java de modelado UML

El paquete org.eclipse.uml2.uml pertenece al API de eclipse de desarrollo de UML2. Esta API provee una implementación basada en el Framework de modelado de Eclipse del metamodelo de UML 2.2 para la plataforma Eclipse.

En el paquete org.eclipse.uml2.uml define mediante la programación orientada a objetos el metamodelo de UML. Los elementos del paquete contienen los constructores y métodos necesarios para la generación de modelos UML mediante código Java, así como la exploración de sus elementos y las características de estos. De esta manera, por ejemplo, una instancia del elemento UML “propiedad”, tendrá acceso a su “tipo”.

3.4. MDD con Rational Software Architect

Rational Software Architect es una herramienta de diseño y desarrollo integrados que aprovecha el Model-Driven Development con UML para crear aplicaciones y servicios con buenas arquitecturas. RSA tiene las siguientes características que son particularmente relevantes en MDD:

- Editor de UML 2.0 con soporte para la refactorización.

- Soporte para perfiles UML 2.0.

- Infraestructura de patrones con librería de patrones.

- Infraestructura de transformaciones con muestras de transformaciones.

Los patrones, perfiles y transformaciones conjuntamente proveen las capacidades requeridas para personalizar RSA para respaldar la automatización de un proceso. RSA incluye también herramientas de desarrollo J2EE, web, y servicios web.

El entorno de modelado con UML y el soporte para perfiles, así como del entorno de desarrollo J2EE, han sido descritos en los puntos anteriores. Presentamos una descripción de otras herramientas basadas en MDD que han sido estudiadas para la realización del proyecto.

3.4.1. Patrones RSA

RSA dispone de una herramienta de autoría de patrones que permite la creación de patrones software que integran soluciones de diseño software en modelos UML. Los autores de patrones pueden usar las capacidades de patrones para diseñar patrones del más simple al más complejo.

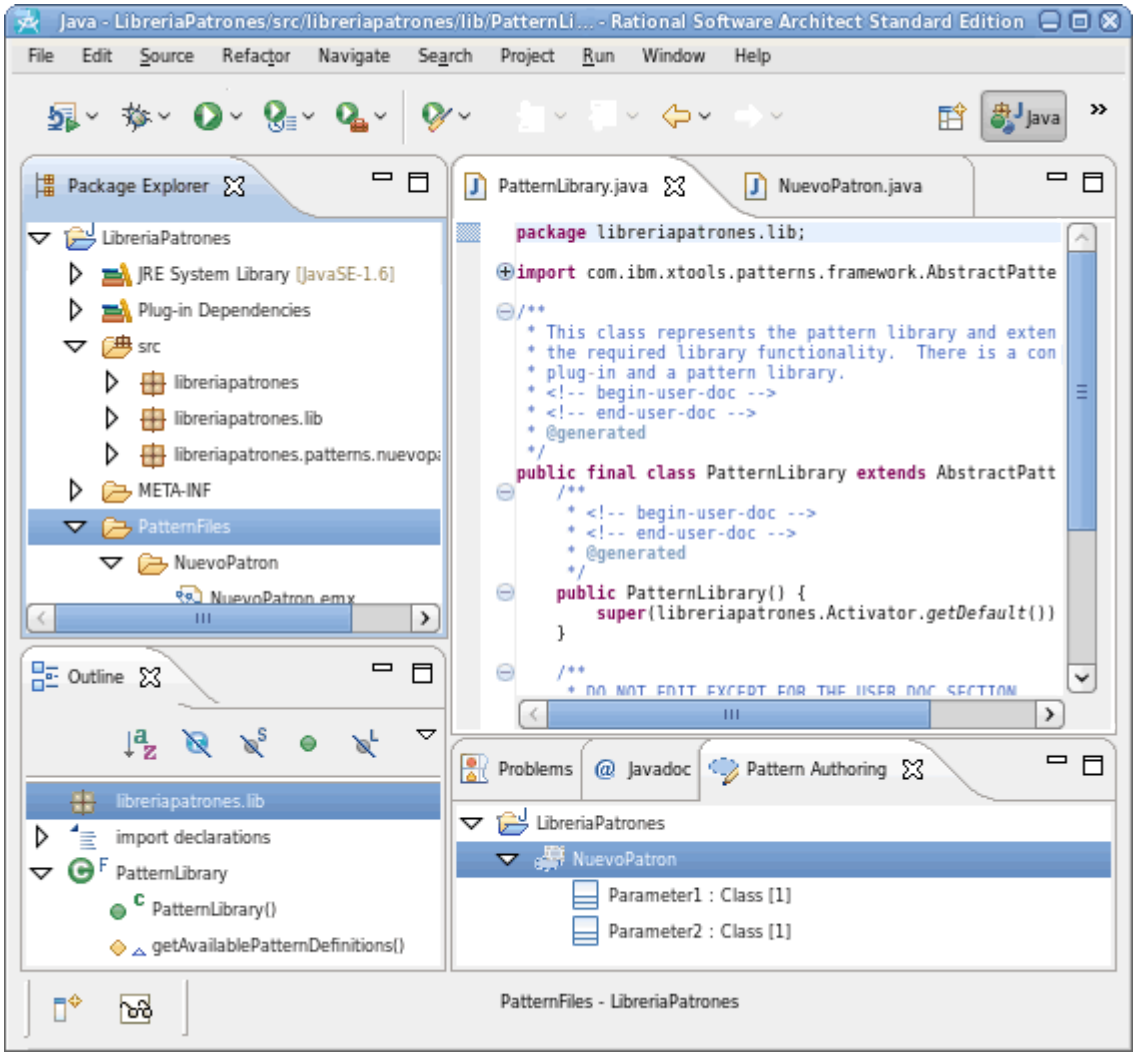


Figura 3.5 Ejemplo de creación de patrón en RSA

El patrón comienza con un modelo UML y termina como un plugin. El modelo de implementación de patrones basado en Java es creado automáticamente mediante la extensión de dos plugins: un servicio de patrones y un framework para patrones que abstrae el uso del servicio de patrones.

Junto con la herramienta de autoría de patrones y la herramienta de navegación de patrones, el servicio de patrones y el framework para patrones proveen las funciones básicas para estructurar, diseñar, codificar, buscar, organizar y aplicar patrones. La herramienta de autoría de patrones es la vista de autoría de patrones (Figura 3.5) y la herramienta de navegación es la vista de explorador de patrones.

Definir patrones de Rational requiere algún conocimiento de elementos UML y conceptos de lenguaje orientado a objetos. El modelo básico del patrón provee código Java por defecto. Las librerías de patrones, los cuerpos de los patrones, los parámetros de los patrones y las dependencias de los parámetros se expresan como clases Java. Se dispone de mecanismos para la documentación y el empaquetado de los patrones.

Creación de patrones

El framework da soporte a un conjunto de mecanismos para el comportamiento por defecto de los patrones, y un conjunto de extensiones para la implementación de patrones específicos.

Este código básico es conocido como el modelo de implementación e incluye una librería de patrones, la definición del patrón, y los parámetros del patrón. El código se añade cuando el autor del patrón crea el proyecto de plugin de la librería de patrones. La librería de patrones, esencial para el diseño del patrón, se crea durante este proceso.

El autor completa el modelo de implementación mediante la adición de uno o más patrones y sus parámetros de plantilla a la librería. Para ello se usa una herramienta de interfaz gráfica. El modelo de implementación es regenerado cada vez que el autor de patrones modifica el modelo de la librería. Los patrones son representados con clases Java y anidadas dentro de estas clases hay una clases por cada parámetro. Cada parámetro tiene métodos de expansión vacíos para tratar la adición, la sustitución o la eliminación de un argumento.

Contenido de un proyecto de patrones

El proyecto de plugin de la librería de patrones se crea en el entorno de desarrollo de plugins de Eclipse mediante el uso de una plantilla para plugins de librerías de patrones que añade los artefactos requeridos por los patrones al proyecto.

Un proyecto de librería de patrones siempre contiene una librería de patrones. Un proyecto de librería de patrones puede poseer uno o más patrones. Un patrón normalmente tiene al menos un parámetro. Aunque múltiples patrones se empaquetan con la misma librería, cada uno es independiente de los otros.

El archivo de manifiesto para el proyecto de librería mantiene una lista de todos los manifiestos en el proyecto. Casa archivo de manifiesto de un patrón mantiene

información sobre el patrón y sus parámetros. Cuando una propiedad del patrón se modifica, los cambios correspondientes se realizan en el manifiesto del patrón.

3.4.2. Transformaciones modelo a modelo

La autoría de transformaciones de modelo a modelo es un proceso basado en modelos que permite al autor especificar modelos o metamodelos de origen y destino, y crear una o más declaraciones de correlación que definen las relaciones entre los elementos en los modelos o metamodelos. Las declaraciones de correlación contienen reglas de correlación que definen relaciones entre características de los elementos; estas reglas de correlación pueden contener información detallada de la implementación. Trabajar a este de nivel de abstracción permite concentrarse en el dominio del problema en lugar de en el dominio de la solución.

Según se definen las declaraciones de correlación y las reglas de correlación, se pueden generar códigos fuente de transformaciones que extienden el framework estándar de transformaciones. Las reglas de correlación se especifican usando un metamodelo de un modelo, incluyendo los metamodelos y perfiles UML del núcleo del framework de modelado de Eclipse (Ecore).

Creación de transformación modelo a modelo

El proceso de creación de transformaciones modelo a modelo consta de los siguientes pasos de alto nivel:

1. Se crea un proyecto de correlación de transformación modelo a modelo que contenga uno o varios modelos de correlación. Cuando se crea un proyecto de correlación, el servicio de transformaciones registra una transformación. Cada transformación tiene un proveedor de transformaciones, un transforme llamado *MainTransform* y un transforme para cada declaración de correlación del proyecto.

2. Se añaden declaraciones de correlación al modelo de correlación. En el modelo de correlación puede haber una o más declaraciones de correlación.

3. Se añaden reglas de correlación a las declaraciones de correlación en un modelo de correlación.

4. Se genera código fuente de transformación a partir del modelo o modelos de correlación del proyecto de correlación. Las herramientas de creación de transformaciones modelo a modelo generan una transformación para cada modelo de correlación del proyecto de correlación. Para cada declaración de correlación, las herramientas de creación generan un archivo fuente Java que implementa un *transform*. Para cada regla de correlación de movimiento o personalizada de una declaración de correlación, se genera una regla en el

código fuente del *transform*. Para cada regla de correlación de subcorrelación, se genera un extractor de contenido en el código fuente del *transform*.

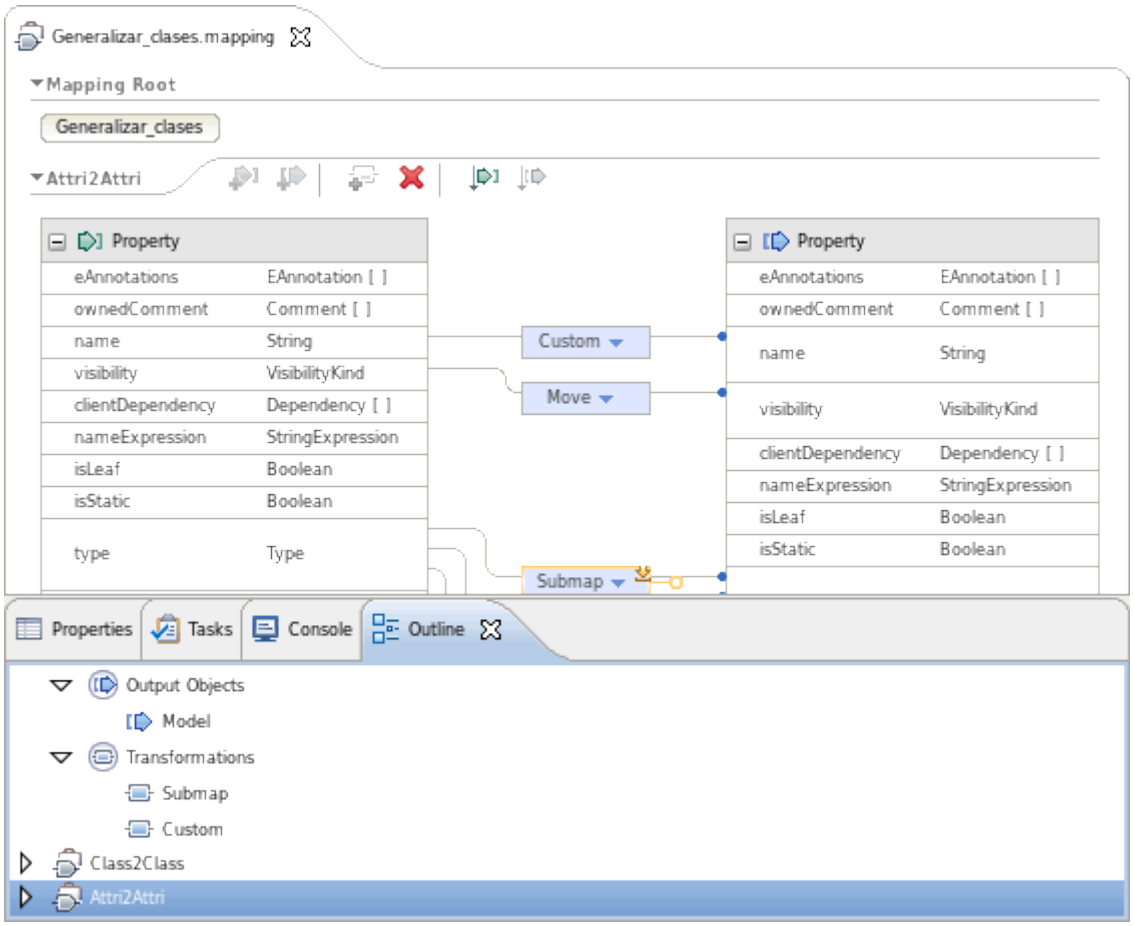


Figura 3.6 Ejemplo de definición de reglas de correlación

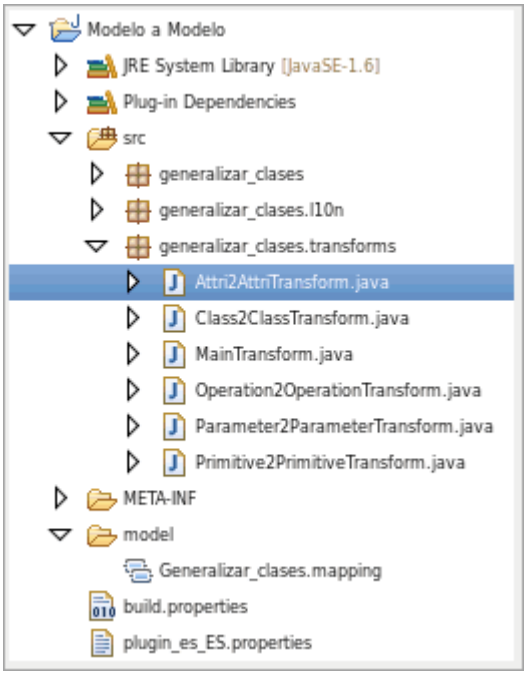


Figura 3.7 Ejemplo de contenido de un proyecto de correlación

3.5. Ontología ASys

Una ontología es una especificación formal de una conceptualización común compartida por las partes interesadas y los expertos en un dominio. Se pueden considerar dos dimensiones en cualquier ontología: la *dimensión pragmática* para expresar su aplicación en un cierto dominio con un uso previsto, desarrollado siguiendo una metodología específica o método de diseño; y la *dimensión semántica* como un mecanismo de representación que estructura, organiza y formaliza un contenido particular, de acuerdo con un nivel de granularidad.

Una ontología contiene clases, relaciones, instancias y axiomas. Las clases corresponden a entidades en el dominio bajo análisis. Las relaciones establecen conexiones entre estas entidades. Las instancias son los objetos reales que están en el dominio. Los axiomas restringen a todos los elementos anteriores.

Enfocado a los sistemas autónomos, las ontologías se usan como mecanismos de representación de conocimientos, en términos de una especificación de una conceptualización. Por consiguiente, la ontología contiene los conceptos para ser manipulados por los actores del sistema autónomo. Los elementos ontológicos se definen en base a un lenguaje computacional, tal como OWL o UML.

El framework de OASys [4], que ha sido mentado en el punto 1.2, captura y aprovecha los conceptos para respaldar la descripción y el proceso de ingeniería de cualquier sistema autónomo. Esto se ha hecho mediante el desarrollo de dos elementos distintos: una ontología en el dominio de los sistemas autónomos (OASys), y una metodología de ingeniería basada en OASys.

La ontología ASys implementada en UML

OASys ha sido desarrollada para describir el dominio de los sistemas autónomos, como soporte software y semántico para el modelado conceptual de la descripción e ingeniería de los sistemas autónomos. La dimensión pragmática de la ontología, es decir, su aplicación en el dominio de los sistemas autónomos, se ha abordado considerando dos ontologías como parte de ello: la ontología ASys para conceptualizar la descripción del sistema autónomo, y la ontología ASys Engineering para hacer lo mismo con los procesos de ingeniería de los sistemas autónomos (Figura 3.8).

Para considerar la dimensión semántica de la ontología, el contenido de las ontologías ASys y ASys Engineering han sido organizadas y estructuradas en sub-ontologías y paquetes.

La Ontología ASys contiene conceptos, relaciones, atributos y axiomas para caracterizar un sistema autónomo, organizados en dos sub-ontologías:

- La sub-ontología System contiene elementos necesarios para definir cualquier estructura de un sistema, comportamiento y función.

- Los elementos de la sub-ontología ASys especializan los conceptos y las relaciones de la sub-ontología System.

La ontología de ingeniería ASys recoge los elementos ontológicos para describir y dar soporte a los procesos de construcción de un sistema autónomo. Comprende dos sub-ontologías para establecer a distintos niveles de abstracción los elementos ontológicos para describir los procesos de desarrollo de la ingeniería de los sistemas autónomos:

- La sub-ontología System Engineering reúne relaciones y conceptos relacionados con los procesos y la ingeniería de los sistemas software.

- La sub-ontología ASys System Engineering contiene la especialización del contenido de la ontología System Engineering, así como elementos ontológicos adicionales para describir un proceso concreto de ingeniería de sistemas autónomos.

Para la realización del proyecto se dispondrá de OASys como ontología de dominio y herramienta de diseño para los modelos y los patrones de los sistemas que capturamos en UML. Ya que los modelos de la ontología han sido realizados en un proyecto de RSA, serán directamente usables en el entorno que se desarrolle.

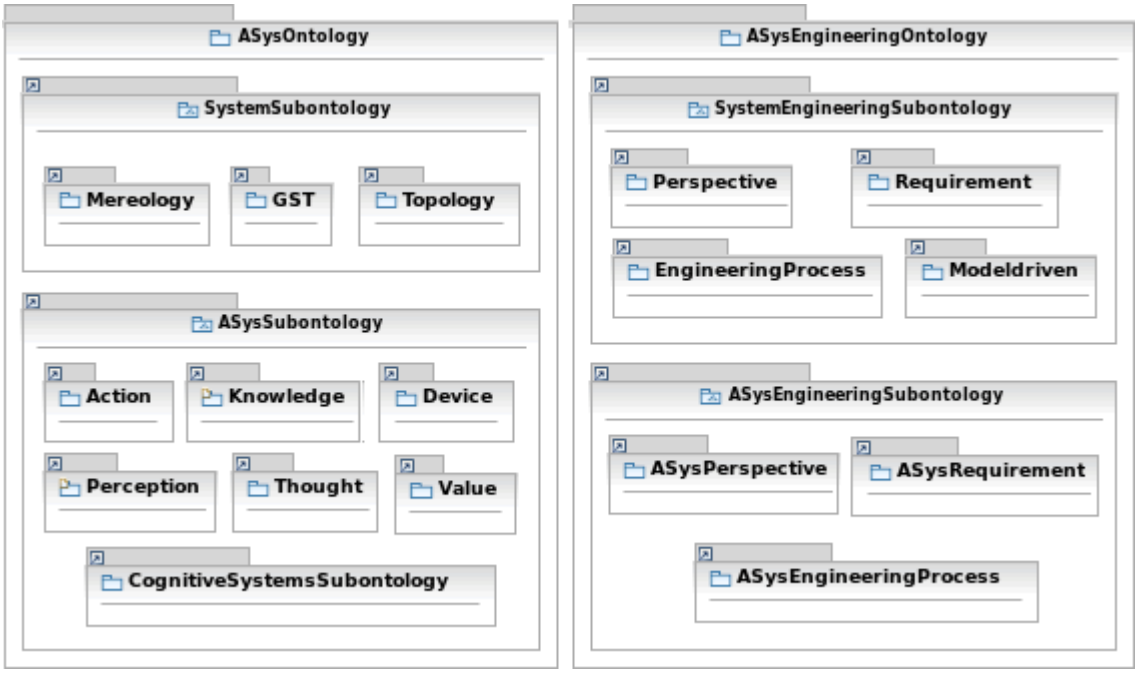


Figura 3.8 Estructura de paquetes de OASys

Capítulo 4

Análisis del problema

4.1. Planteamiento inicial

En el presente apartado se realiza un análisis descriptivo a nivel de usuario del planteamiento de partida para la concepción de la herramienta. Se presenta la funcionalidad objetivo de la herramienta y los requisitos principales que ha de satisfacer para su correcto funcionamiento.

4.1.1. Propósito de la herramienta

Se quiere desarrollar una herramienta de modelado para el desarrollo de sistemas de control en el marco del Desarrollo Basado en Modelos (Model-Driven Development). La finalidad de la herramienta es facilitar el desarrollo por medio de la aplicación automática de patrones a los modelos UML empleados en el desarrollo de sistemas de control intensivos en software.

4.1.2. Alcance de la herramienta

El resultado de aplicar con la herramienta un patrón sobre un modelo será la producción de un nuevo modelo en el que se habrán modificado o generado nuevos elementos de acuerdo con la especificación del patrón.

Se presenta pues dos problemas directamente relacionados: desarrollar una herramienta para aplicar patrones y definir un lenguaje de especificación de los patrones adecuado para la herramienta. Para llevar a cabo estas tareas se dispone de diferentes posibilidades en el entorno de trabajo de Rational Software Architect.

4.1.3. Requisitos de la herramienta

Presentamos una lista de los principales requisitos que se toman como punto de partida para la concepción de la herramienta:

- Es una herramienta de modelado para trabajar con el entorno de desarrollo de sistemas software Rational Software Architect.
- La herramienta debe ser aplicable a cualquier modelo UML 2 desarrollado en Rational Software Architect.
- Los patrones deben ser independientes de los modelos sobre los que se aplican, y por tanto, deben permanecer inalterados tras su aplicación sobre un modelo.
- Los patrones han de ser modelos UML2
- El desarrollo de los patrones ha de limitarse al diseño y especificación de éstos, siendo la herramienta la que interprete los patrones y realice las transformaciones oportunas sobre el modelo de aplicación de forma automática.
- Tras la aplicación del patrón al modelo éste debería reflejar información sobre dicha aplicación, identificando por ejemplo los elementos del modelo que participan en la instancia del patrón y los roles que desempeñan en la misma, por lo que se debe poder realizar algún tipo de enlazado del modelo al patrón.

4.2. Casos de uso

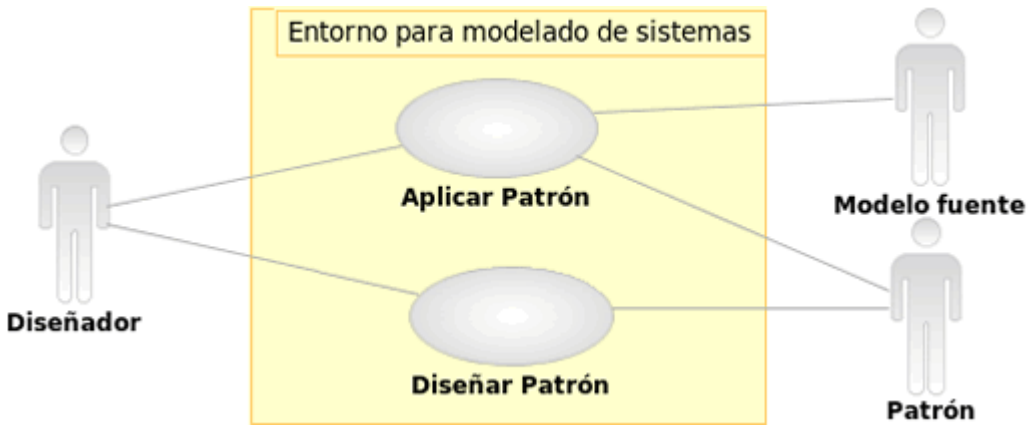


Figura 4.1 Casos de uso de la herramienta PatternApplier

Definición de los Actores		
AC_1	Diseñador	Usuario de la herramienta que diseña modelos en UML
AC_2	Modelo fuente	Modelo sobre el que se aplica un o varios patrones
AC_3	Patrón	Patrones que se diseñan para ser aplicados a otros modelos
Definición de los Casos de Uso		
CU_1	Aplicar patrón	El diseñador selecciona un patrón y un modelo de destino. El modelo es transformado en base al patrón
CU_2	Diseñar patrón	El diseñador crea un patrón que posteriormente podrá ser aplicado a cuantos modelos queramos

Tabla 1. Definición de los actores y los casos de uso

4.3. Análisis de requisitos

Procedemos a enumerar formalmente los requisitos de la herramienta con códigos identificativos y clasificados en grupos. La notación usada será **RX_Y** donde **X** hará referencia al tipo de requisito, e **Y** será el número del requisito.

4.3.1. Requisitos funcionales

RF_1 La herramienta transforma un modelo en base a un patrón que se le aplica.

RF_2 La herramienta permite especificar un modelo ya existente sobre el que se aplica el patrón.

RF_3 La herramienta permite que elementos del modelo sobre el que se aplica el patrón realicen el rol de elementos definidos en el patrón.

RF_4 Se debe poder definir nuevos patrones sin usar la herramienta de aplicación de éstos.

RF_5 Los patrones se especificarán mediante modelos UML2.

4.3.2. Requisitos del sistema

RS_1 Es una herramienta que funciona en el entorno de desarrollo Rational Software Architect 7.5.x.

RS_2 Los patrones podrán diseñarse y especificarse en proyectos de modelado UML2 de RSA.

RS_3 Los modelos sobre los que se pueda aplicar la herramienta serán modelos UML2 de RSA.

RS_4 Los modelos sobre los que se aplican los patrones podrán permanecer abiertos en la vista de modelado de RSA en el momento de aplicación de la herramienta.

RS_5 Los patrones son independientes de la herramienta que realiza las transformaciones sobre los modelos, así como de los modelos sobre los que se apliquen.

4.4. Análisis de las soluciones disponibles

Se procede con el análisis de las posibilidades que ofrecen las distintas soluciones disponibles que han sido descritas en el capítulo 3. Este análisis tendrá como objeto identificar los requisitos que satisfacen cada solución para poder realizar un diseño que cumpla con todos los requerimientos.

4.4.1. Perfiles UML2

Los perfiles UML2 permiten extender la sintaxis y la semántica del UML para expresar conceptos específicos dentro de un determinado dominio de aplicación. Usando estereotipos podemos marcar elementos UML. Estos estereotipos podrán aportar valor de diseño o funcionar como especificaciones de transformaciones.

Será posible pues capturar patrones en modelos UML2 usando los perfiles como la notación necesaria para especificar las transformaciones, satisfaciendo **RF_5.** Para el diseño de estos modelos no se requiere el uso de la herramienta para aplicar los patrones, (**RF_4**)**.**

Dado que RSA 7.5.0 soporta el modelado UML con perfiles, se podrán diseñar los patrones como modelos UML en el entorno de modelado de Rational, (**RS_2**)**.** Además,

los patrones no tienen que estar en el mismo proyecto que la herramienta que los aplica, por lo que también se cumple **RS_5.**

El resto de requisitos no se cumplen pues hacen referencia a la herramienta que realiza la transformación (aplicación del patrón) y a los modelos sobre los que se aplican los patrones.

4.4.2. Patrones RSA

RSA provee de herramientas de autoría de patrones, y permite su posterior aplicación (**RS_1).** Los patrones se podrán aplicar a modelos UML (**RS_3**) con los que se está trabajando en el entorno de modelado de RSA (**RS_4**).

Cuando se crea un patrón RSA se empieza creando un proyecto de librería de patrones al que se añaden patrones. El modelo de implementación se diseña a través de una interfaz gráfica. Después el autor completa con código las clases generadas. Este desarrollo de los patrones hace que no sé cumpla RS_2 ni RF_5 pues la creación de los patrones no se limita al diseño de estos únicamente. Posteriormente, los patrones son exportados como plugins con todo el contenido, listos para aplicarse, por lo que tampoco se cumple RS_5.

Considerando los patrones diseñados como patrón y herramienta de aplicación, todo en uno, cumpliría el **RF_1** ya que realiza las transformaciones oportunas sobre el modelo**.** Además, el patrón presentará parámetros que serán substituidos por elementos del modelo sobre el que se aplica el patrón (**RF_3**). Como los plugins de patrones RSA son patrón y herramienta de aplicación a la vez no tiene sentido valorar el cumplimiento de RF_4.

Para aplicar el patrón a un modelo es necesario crear una instancia del patrón en el propio modelo, por lo que se puede especificar el modelo sobre el que se aplica el patrón y que sea la herramienta quien identifique el modelo (**RF_2**).

4.4.3. Transformaciones modelo a modelo de RSA

Cuando se crea una transformación modelo a modelo, se establecen unas reglas de correlación entre los elementos de un modelo o metamodelo. A partir de las reglas de correlación se genera código de transformación. Al aplicar la transformación se realiza una transformación del modelo en base a estas reglas (**RF_1**). Para aplicar la transformación, se especifica en el modelo que se le quiere aplicar dicha transformación (**RF_2**). De la explicación de este proceso de creación de las transformaciones, se concluye que no se cumple RF_5.

Las transformaciones se aplican sobre los elementos transformándolos en base a las reglas establecidas, pero en ningún momento un elemento del modelo origen puede

jugar ningún tipo de rol ni interactuar de ninguna manera con los modelos de correlación que definen las transformaciones o la propia transformación generada a partir de estos modelos (no cumple RF_3).

Se podrán diseñar distintos modelos de correlaciones con sus distintas reglas; y a partir de ellos los códigos de transformación para implementar las transformaciones de distintos patrones. Esto se realiza independientemente de la herramienta de aplicación. (**RF_4**).

Las transformaciones son una herramienta soportada por RSA (**RS_1**), pero no pueden diseñarse en un proyecto de modelado UML2 (no cumple RS_2). Si se aplicarán sobre modelos UML2 que estén abiertos en el entorno de modelado (**RS_3** y **RS_4**).

Al igual que pasaba con los patrones RSA, no tiene sentido considerar los patrones creados independientemente de una herramienta de aplicación de éstos (no cumple RS_5).

4.4.4. Pluglets

Los pluglets permiten extender el entorno de modelado. Se podrán implementar pequeñas herramientas software, que haciendo uso de distintos paquetes IBM y Eclipse, permitirán manejar o generar modelos UML mediante programación.

4.4.4.1. Plataforma de modelado de Rational

El pluglet que se genera es una herramienta para RSA (**RS_1**) con la que se podrán manejar los modelos abiertos en el momento de ejecución (**RS_4**). Estos modelos podrán ser modelos UML2 (**RS_3**). La herramienta será independiente de los patrones (**RS_5** y **RF_4**) y permitirá manejar los patrones que se hayan definido en proyectos de modelado UML (**RS_2** y **RF_5**).

Para poder transformar los modelos se necesitará hacer uso de componentes para generar UML2 mediante programación (no cumple RF_1). Tampoco se dispondrá de los componentes necesarios para trabajar con elementos que juegan un rol dentro de los patrones (no cumple RF_3). Si se podrá señalar en el entorno de trabajo sobre qué modelos se aplican los patrones (**RF_2**).

4.4.4.2. Org.eclipse.uml2.uml

El paquete org.eclipse.uml2.uml permite generar modelos UML mediante programación. Además, permite navegar por los elementos de los modelos y acceder a todas sus características y subelementos.

Los elementos del paquete se usarán en una aplicación (o pluglet) que podrá ejecutarse en RSA (**RS_1**) y que será independiente de los patrones (**RS_5** y **RF_4**). Se podrán explorar modelos y patrones que hayan sido definidos en UML (**RF_5** y **RF_2**), pero para poder acceder a elementos abiertos en el entorno de modelado de RSA se requiere el uso de los componentes de la plataforma de modelado de Rational, por lo que no se cumplen (RS_2, RS_3 y RS_4).

El paquete contiene todos los componentes necesarios para crear elementos UML por lo que se podrán transformar modelos UML (**RF_1**). Además, con el elemento de UML "TemplateParameter" se podrán manejar elementos que substituyan a otros elementos en los patrones (**RF_3**).

4.4.5. Tabla resumen Soluciones –Requisitos

Solución \ Requisitos funcionales	RF_1	RF_2	RF_3	RF_4	RF_5
Perfiles UML2				X	X
Patrones RSA	X	X	X		
Transformaciones modelo a modelo	X	X		X	
Pluglet / plataforma modelado Rational		X		X	X
Pluglet / org.eclipse.uml2.uml	X	X	X	X	X

Tabla 2. Resumen de requisitos funcionales frente a las soluciones

Solución \ Requisitos de sistema	RS_1	RS_2	RS_3	RS_4	RS_5
Perfiles UML2		X			X
Patrones RSA	X		X	X	
Transformaciones modelo a modelo	X		X	X	
Pluglet / Plataforma modelado Rational	X	X	X	X	X
Pluglet / org.eclipse.uml2.uml	X				X

Tabla 3. Resumen de requisitos funcionales frente a las soluciones

Capítulo 5

Solución adoptada

5.1. Descripción general de la solución

A partir del análisis de los requisitos de la herramienta y del estudio de las soluciones disponibles se ha diseñado una solución que cumple con las necesidades establecidas. La solución que se ha desarrollado es el resultado de la combinación de tres de las soluciones con las que se ha trabajado durante la realización del proyecto, y que han sido explicadas en el capítulo 3. Éstas son el uso de perfiles UML2, la plataforma de modelado de Rational y las librerías Eclipse para generación de UML mediante programación.

Los perfiles han servido para definir una notación para extender UML como lenguaje formal de especificación de patrones, permitiendo especificar las transformaciones que han de realizarse sobre el modelo sobre el que se aplica el patrón. Por otro lado, el uso de elementos de plantilla de UML ha posibilitado que elementos del modelos sobre el que se aplica el patrón puedan jugar roles dentro de los patrones. De esta manera, los patrones son capturados en modelos que la herramienta puede interpretar. Resulta una forma sencilla y clara de diseñar los patrones puesto que se puede hacer uso de los diagramas de clases de UML para modelar los patrones de forma visual.

La herramienta encargada de aplicar los patrones ha sido implementada como un pluglet. Éste puede ser lanzado en el entorno de modelado de RSA durante el desarrollo de sistemas basados en modelos. El pluglet hace uso de la plataforma de modelado

Rational para poder extender el entorno de modelado y manejar los modelos abiertos en el entorno de trabajo. Por otro lado, con las librerías de modelado UML de eclipse, se analizan los elementos de los patrones y se implementan las transformaciones necesarias.

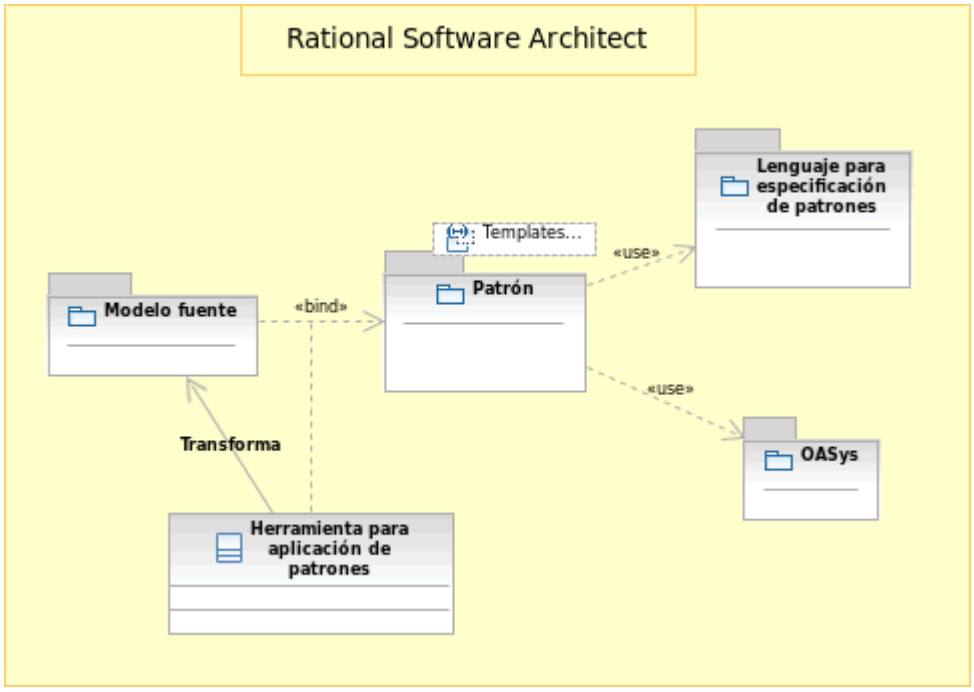


Figura 5.1 Esquema de los elementos implicados en la aplicación de un patrón

5.2. Lenguaje para la especificación de los patrones

Como se ha descrito anteriormente, los patrones se especifican en modelos UML. Esto se realiza usando un lenguaje propio de especificación que hace uso del UML2 y de un determinado perfil diseñado para especificar transformaciones en el patrón.

Esta solución se basa en los enfoques para el diseño de patrones presentados en el capítulo del estado del arte. En una solución de acuerdo entre el enfoque que usa colaboraciones parametrizadas para la definición de la estructura del patrón, y el uso de perfiles como un lenguaje concreto de especificación de transformaciones basadas en patrones.

Estos enfoques se siguen de forma parcial pues para realizar las colaboraciones entre el patrón y el modelo sobre el que se aplica se han utilizado los elementos de UML2 *TemplateParameter* (parámetro de plantilla) y *TemplateBinding*, (vínculo de plantilla) en lugar de usar colaboraciones. Por otro lado, el uso de perfiles se ha limitado a una notación para especificar transformaciones, en lugar de usarse para indicar parámetros dentro del patrón también.

5.2.1. Parámetros de plantilla

Un parámetro de plantilla es un elemento UML que expone un elemento parametrizable como un parámetro formal para la propia plantilla. Esto significa que la plantilla presenta un elemento del tipo de la plantilla que puede ser modelado en base al rol que representa.

En la figura 5.2 se muestra un ejemplo de definición de patrón que usa parámetros de plantilla. En el lado izquierdo de la imagen se muestra la estructura del patrón, mientras que en el lado derecho se muestra el diagrama de clases del patrón. El marco rojo contiene un parámetro de plantilla de una clase *Parameter : Class;* éste contiene un elemento parametrizable *Parameter* del tipo de la plantilla, que es el parámetro formal expuesto por el parámetro de plantilla y que puede ser modelado como se aprecia en el lado de la derecho dentro del marco verde.

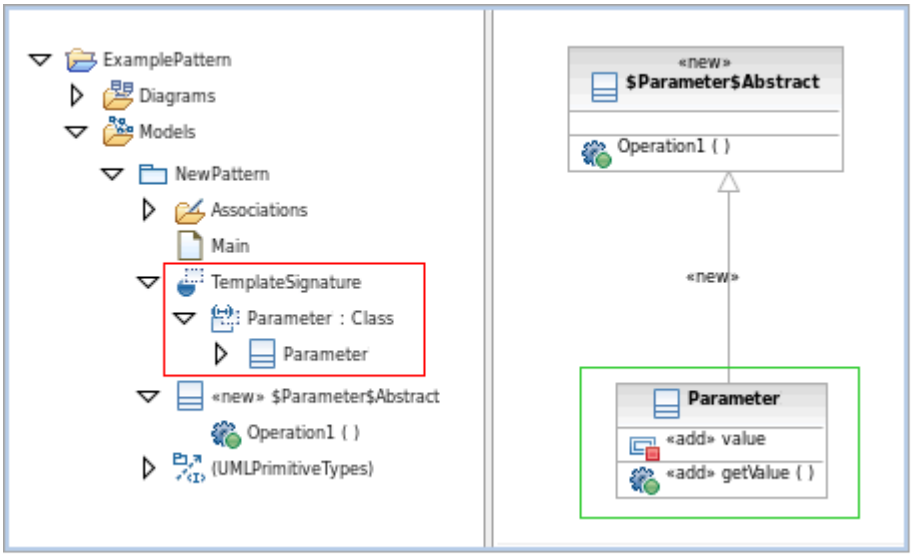


Figura 5.2 Ejemplo de uso de parámetros de plantilla en la definición de un patrón

Los elementos parametrizados en el patrón especifican un rol dentro del patrón que será interpretado por elementos del modelo sobre el que se aplique el patrón. Esta substitución de roles se realiza a través de los vínculos de plantilla. Son elementos UML que representan una relación entre un elemento y una plantilla. Especifican una substitución del parámetro formal de la plantilla por un parámetro real. Estos elementos se añadirán al modelo sobre el que se quiere aplicar el patrón permitiendo la indicación de los elementos que realizan las substituciones[2].

[2] Consultar Manual de usuario en los Anexos para uso de parámetros de plantilla en el diseño de patrones.

5.2.2. Perfil *PatternSpec* para especificación de transformaciones

El perfil *PatternSpec* ha sido diseñado para definir una notación de especificación de transformaciones. Los mecanismos usados son los estereotipos que permiten marcar los elementos del modelo según la transformación que requieran.

En la figura 5.3 se muestra el diagrama de clases del perfil *PatternSpec*. Los estereotipos *add*, *new* y *delete* extienden a la metaclase *Element;* esto significa que en principio, cualquier elemento que en la jerarquía de UML extienda a *Element* puede ser estereotipado con cualquiera de estos tres estereotipos. Sin embargo, los estereotipos *rename* y *optional* extienden la metaclase *ParameterableElement*, por lo que sólo los elementos parametrizables podrían ser estereotipados con estos dos estereotipos. La distinción se debe a que los estereotipos *rename* y *optional* están diseñados para estereotipar elementos formales dentro de los parámetros de plantilla.

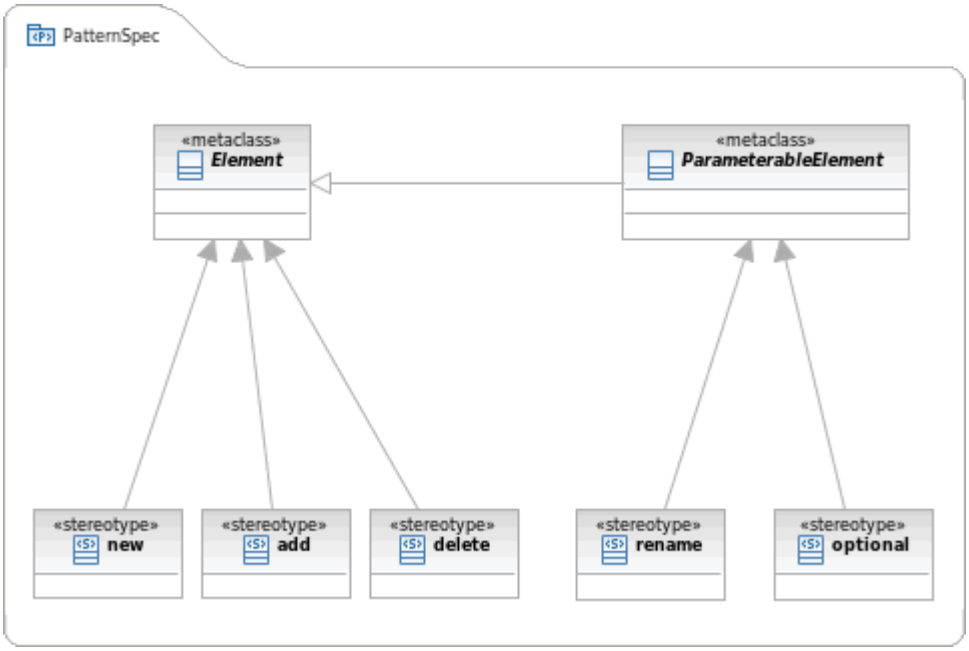


Figura 5.3 Diagrama de clases del perfil *PatternSpec*

La herramienta que aplica los patrones interpretará los estereotipos de los elementos para saber qué transformación debe realizar. A continuación se describe el significado de que un elemento cualquiera esté marcado con cada uno de los estereotipos del perfil:

New. El elemento del patrón será copiado en el modelo sobre el que se está aplicando el patrón.

Add. Se comprobará si existe un elemento similar a éste en el modelo sobre el que se aplica el patrón, y en caso de no existir, se copiará como pasaba con *new.*

Delete. Se comprobará si existe un elemento similar a éste en el modelo sobre el que se aplica el patrón, y en caso de existir, éste elemento del modelo será eliminado.

Rename. El estereotipo se aplicará a un elemento parametrizado dentro de una plantilla en el patrón. El elemento del modelo sobre el que se aplica el patrón que substituya a este elemento estereotipado será renombrado con el nombre del propio elemento estereotipado.

Optional. El estereotipo se aplicará a un elemento parametrizado dentro de una plantilla en el patrón. La substitución del elemento parametrizado por un elemento del modelo sobre el que se aplica el patrón será opcional.

En la figura 5.4 se muestra el ejemplo de patrón usado anteriormente pero remarcando los estereotipos usados.

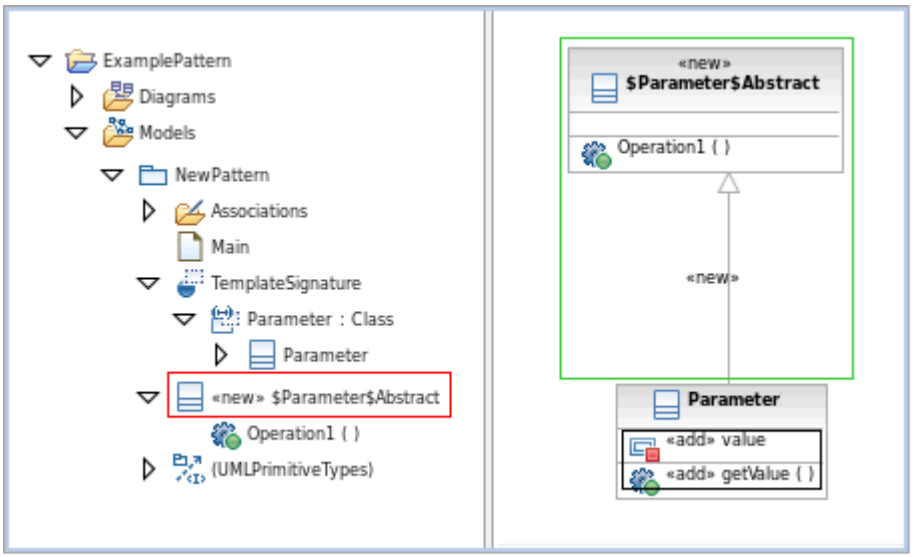


Figura 5.4 Ejemplo de uso de estereotipos en la definición de un patrón

El lado derecho que muestra la estructura del patrón, hay enmarcada un clase en rojo *$Parameter$Abstract* con el estereotipo *new.* Esa misma clase se observa en el diagrama enmarcada en verde. Cuando el patrón se aplique a un modelo, esa clase se creará en el modelo. También se observa una generalización de *Parameter* a *$Parameter$Abstract,* que se creará en el modelo, e irá de la clase del modelo que substituya a *Parameter* a la nueva clase creada en el modelo *$Parameter$Abstract.* Finalmente, dentro del marco negro, se aprecia que los atributos y operaciones del elemento parametrizado *Parameter* están estereotipados con *add.* Esto quiere decir, que aunque una clase substituya a *Parameter* y adquiera sus características, los atributos y operaciones se copiarán sólo si la clase no posee ya unos similares.

5.2.3. Parámetros para renombrar elementos

Además de los mecanismos para el diseño de los patrones que se han descrito en los dos puntos anteriores, se ha diseñado una nomenclatura para renombrar elementos que

se crean en el modelo a partir del patrón. Los elementos se renombran insertando, en el propio nombre del elemento, los nombres de elementos del modelos sobre el que se aplica el patrón. Para indicar que el elemento será renombrado se usa una cadena de caracteres que será sustituida por el nombre que se le pasa como parámetro.

La cadena tiene la forma *$reference_element$*. Los símbolos **$** indican que se inserta un parámetro. *Reference_element* es el elemento del patrón que sirve de referencia para tomar el nombre que se inserta en el nombre del elemento a renombrar. El elemento de referencia tiene que ser un elemento parametrizado del patrón; y el nombre se toma del elemento del modelo que juega el rol de dicho elemento.

En la figura 5.5 se muestra enmarcado en rojo el nombre del elemento del patrón que será renombrado tras copiarse al modelo. En el diagrama se enmarca en verde. La cadena *$Parameter$* indica que la propia cadena será sustituida por otra cadena. La referencia es *Parameter*. Esto quiere decir que la cadena que se tomará como parámetro será el nombre del elemento del modelo que substituya al elemento de referencia *Parameter*. Por ejemplo, si la clase que juega el rol de *Parameter* se llama *Client*, la clase *$Parameter$Abstract* que será copiada al modelo sobre el que se aplica el patrón tendrá como nombre final *ClientAbstract*.

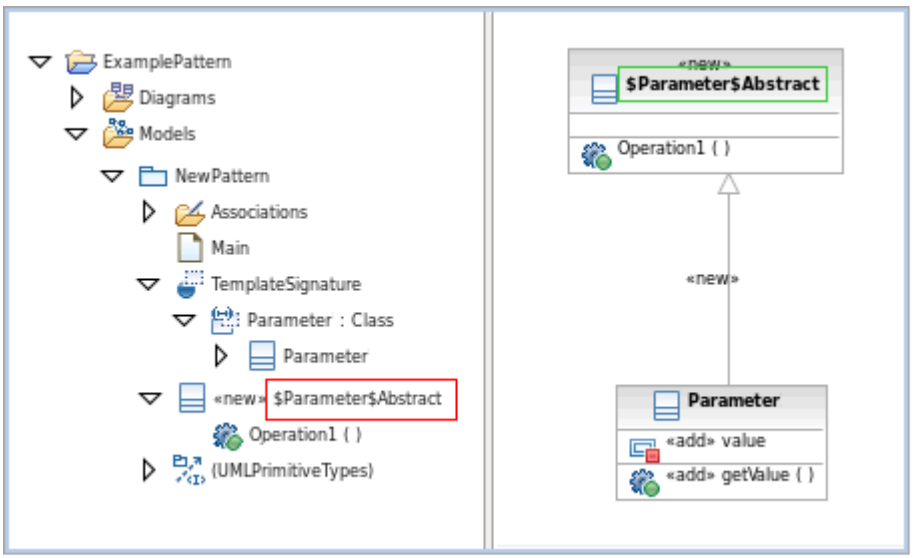


Figura 5.5 Ejemplo de renombre de elementos en la definición de un patrón

5.3. Herramienta para aplicar patrones *PatternApplier*

La solución que se ha adoptado para la herramienta que aplica los patrones ha sido el desarrollo de una aplicación Java, a la que se ha denominado *PatternApplier*, que interactúa entre el modelo y los patrones implicados, realizando las transformaciones.

A parte de cumplir con todos los requisitos exigidos, se ha considerado la mejor opción por diversas razones triviales para el buen funcionamiento del conjunto de la

herramienta. Primero, la facilidad a la hora indicar el modelo y los patrones como parámetros de trabajo para la aplicación. Segundo, la posibilidad leer los estereotipos usados en el diseño de los patrones y poder actuar en base a su interpretación. Finalmente, la libertad para modelar las características UML de los elementos en los modelos hasta el cierto nivel de detalle de acuerdo a las necesidades requeridas.

5.3.1. Descripción del funcionamiento de la herramienta

Como se ha descrito anteriormente, la herramienta está implementada como un pluglet que aporta la ventaja de funcionar ampliando el entorno de modelado de Rational. Así, la aplicación se puede usar repetidas veces durante le etapa de diseño de los distintos sistemas que se estén modelando.

A continuación, se procede a la descripción de los distintos pasos que tienen lugar durante el proceso de aplicación de patrones desde que el usuario decide aplicar un patrón a un modelo. La figura 5.6 ilustra este proceso. Existen pasos previos al lanzamiento del pluglet que debe realizar el diseñador. Estos son importar los proyectos que contienen los patrones al entorno de trabajo y realizar el enlazado del modelo a los patrones que se van a aplicar sobre el modelo[3].

Tras estos pasos previos, se puede proceder a la ejecución del pluglet pasando el modelo sobre el que se aplican los patrones como parámetro[4]. La información referente a los patrones que se aplican sobre el modelo va implícita en el propio modelo pues se ha enlazado previamente el modelo con los patrones mediante vínculos de plantilla.

Para la lectura del modelo se hace uso de la clase *UMLModeler* de la plataforma de modelado de Rational que permite manejar el dominio de edición de los modelos, y obtener elementos del entorno de modelado que estuviesen seleccionados en el momento del lanzamiento de la herramienta. Se analiza el elemento seleccionado, y si todo es correcto, es decir, es un modelo con patrones aplicados, se genera una estructura de datos que contiene las referencias del modelo sobre el que se aplica el patrón, los patrones, y los enlaces del modelo al patrón. Por consola se listará el modelo sobre el que se aplica el patrón, y los patrones que se le aplican. A partir de este momento comienza la aplicación de los patrones al modelos, es decir, la transformación del modelo. Tendrá lugar un proceso de aplicación de un patrón al modelo, que se realizará una vez por cada patrón enlazado al modelo.

El proceso de aplicación del patrón consiste en tres etapas que serán descritas en más detalle posteriormente. En la primera se resuelven las transformaciones de los elementos que juegan algún rol dentro del patrón.

[3] Consultar Manual de usuario en los Anexos para el enlazado del modelos con los patrones.

[4] Consultar Manual de usuario en los Anexos para la ejecución de la herramienta pasando un elemento como parámetro.

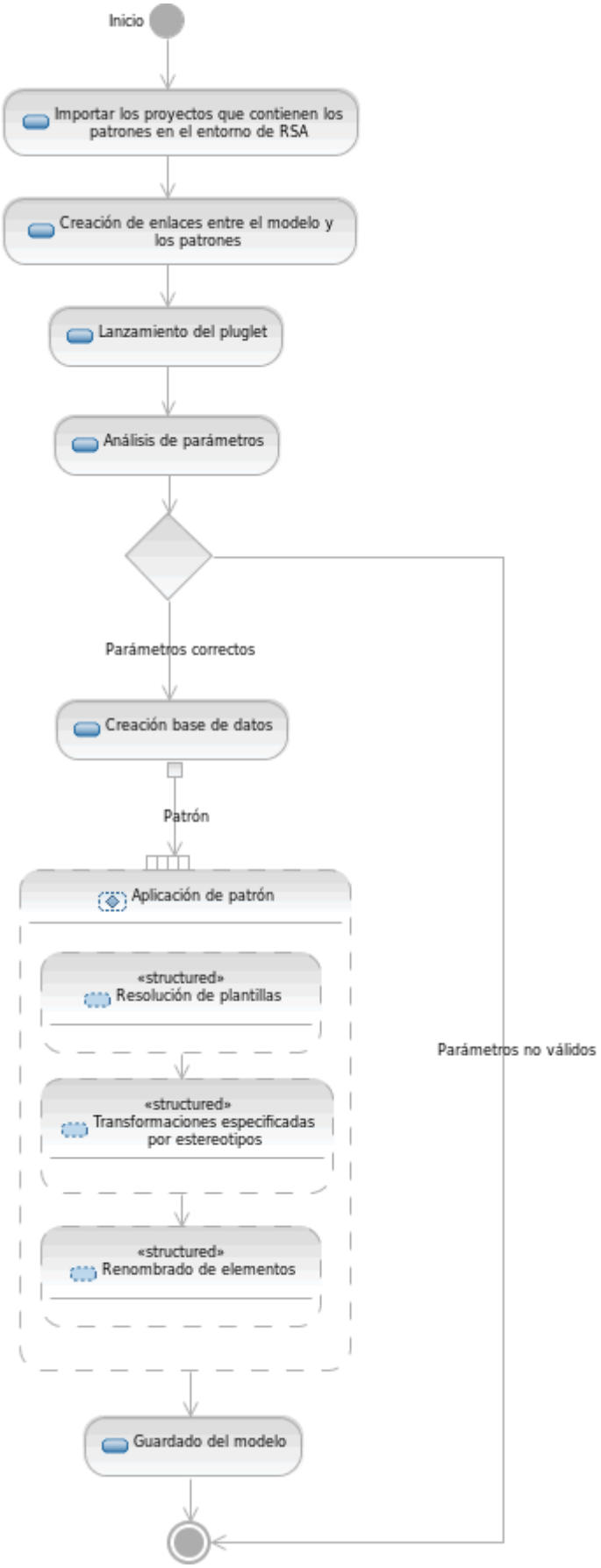


Figura 5.6 Diagrama de flujo general de ejecución de la herramienta *PatternApplier*

Durante esta etapa, se listarán por consola los elementos que juegan roles junto con el rol que desempeñan. En la segunda etapa se analizan los elementos del patrón en busca de estereotipos del perfil *PatternSpec* para realizar las transformaciones oportunas sobre el modelo. En la última etapa, se renombran los elementos del modelo que hayan sido creados a partir del patrón y requieran ser renombrados.

Tras finalizar la aplicación del patrón, el bucle se repite con el siguiente patrón que hubiese sido enlazado con el modelo para aplicarlo. Cuando todos los patrones han sido aplicados, se guardan el modelo que ha sido transformado y termina la ejecución del programa.

Etapas de resolución de plantillas

Esta etapa comienza, como se aprecia en la figura 5.7, con el análisis del vínculo de plantilla *(TemplateBinding)* del modelo al patrón. En él se encuentra la lista de substituciones que se han realizado previamente a la ejecución de la herramienta. Para cada substitución se obtiene una lista de los parámetros reales que substituyen al parámetro formal del patrón. Se considera pues, que un rol puede ser jugado por varios elementos a la vez. Cada parámetro real del modelo es transformado en base al tipo de elemento que es, y siempre tomando como referencia el parámetro formal al que substituyen. Las transformaciones, que serán explicadas con más detalle posteriormente, consisten en copiar las características del parámetro formal al parámetro que lo substituye. También se manejarán elementos propios del parámetro formal que estarán estereotipados, y por tanto, su transformación dependerá de estos estereotipos.

Etapa de transformaciones por estereotipos

Esta etapa está representada en el diagrama central de la figura 5.7. En ella se recorre dos veces la estructura del patrón analizando cada elemento. Se realiza un doble bucle debido a que es necesario que ciertos elementos se creen en el modelo antes que otros. Por ejemplo, las clases que están unidas por una asociación deben estar creadas en el modelo cuando se va a crear la asociación. Por ello, existe la variable *loop* que indica en qué bucle se encuentra la aplicación, y filtra los elementos.

Para cada elemento, se comprueba si tiene especificada alguna transformación a través de un estereotipo del perfil *PatternSpec*. De ser así, se procede a la transformación del modelo. Las transformaciones dependen del estereotipo, y del tipo de elemento estereotipado.

Etapa de renombrado de elementos

En la etapa de renombrado, mostrada a la derecha de la figura 5.7, ya se han realizado todas las transformaciones. Todos los elementos han sido creados en el modelo y sólo queda comprobar si es necesario renombrar alguno.

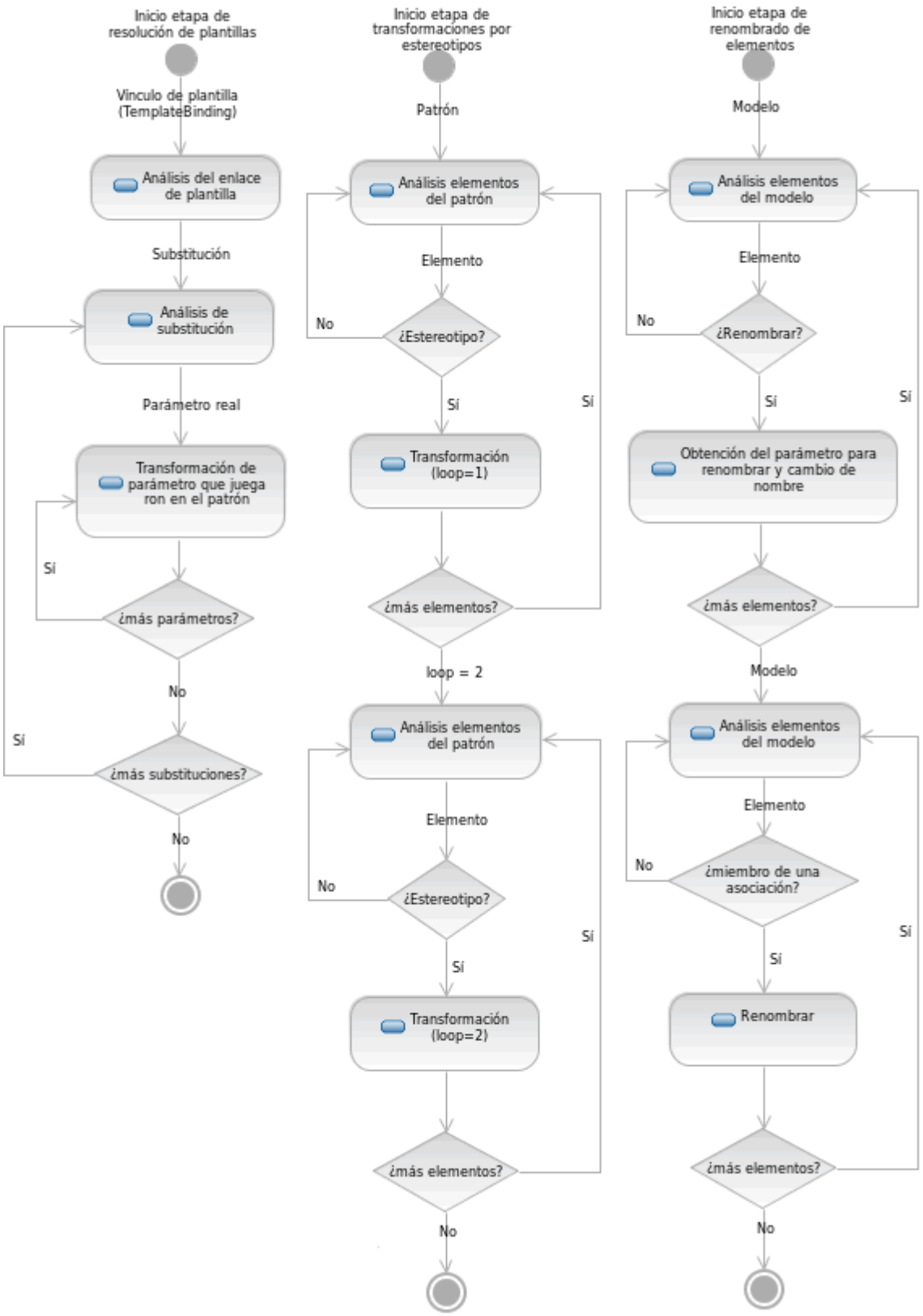


Figura 5.7 Diagramas de de las etapas del proceso de aplicación de los patrones

Para ello se recorre todos los elementos del modelo y se busca si en su nombre se está la cadena para introducir parámetros en el nombre (punto 5.2.3). De ser así, se procede a renombrar el elemento. Esto es así para todos los elementos, excepto para los miembros de las asociaciones, que son renombrados en un segundo bucle a partir de los tipos.

El método que realiza el renombrado será explicado más adelante, en el apartado de implementación, pero su funcionamiento básico es el siguiente. Se toma el nombre del elemento de referencia a partir de la cadena que indica el renombrado. Se localiza este elemento en el patrón. Después se busca en el vínculo de plantilla el elemento que substituye al elemento de referencia, y se toma su nombre para usarlo en el nombre del elemento a renombrar.

5.3.2. Estructura de la herramienta

El código de la herramienta está estructurado en 5 clases, *PatternApplier, PatternParameters, PatternTools, UMLTools* y *CheckTools.* En la figura 5.8 se muestra el diagrama de clases en el que se pueden apreciar los atributos y operaciones de las distintas clases, así como las generalizaciones y usos entre las clases. Además, existen distintas clases anidadas en la clase PatternTools usadas como estructuras de datos.

La programación de la aplicación es estructurada. Durante la ejecución no se crean instancias de las clases anteriores, sino que se usan como librerías que aportan distintos métodos que realizan las transformaciones manejando los modelos, los elementos de los modelos y las estructuras de datos.

Class PatternApplier

Es la clase que contiene el *plugtletmain( )* que controla las distintas fases de aplicación de los patrones. Hereda de la clase *PatternTools* pudiendo usar los métodos y atributos necesarios para la evolución del proceso.

Class PatternTools

Esta clase contiene principalmente, los métodos necesarios para manejar las transformaciones especificadas por los estereotipos. *handleElement ( ), createElement ( )* y *createRelations( ).* También hay métodos para preparar el modelo antes de las transformaciones, y guardarlo al final. *refactorTarget( ), organizePackageImports( )* y *save( ).*

En esta clase están declaradas distintas clases anidadas de las cuales se crean instancias para manejar datos durante la ejecución. *PatternBinding* y *PlugletParameter* se usarán para agrupar el modelo, los patrones y los vínculos de plantilla en una sola

variable. *ClassifiersLists* y *TypesLists* manejarán listas de elementos que participen en relaciones de algún tipo.

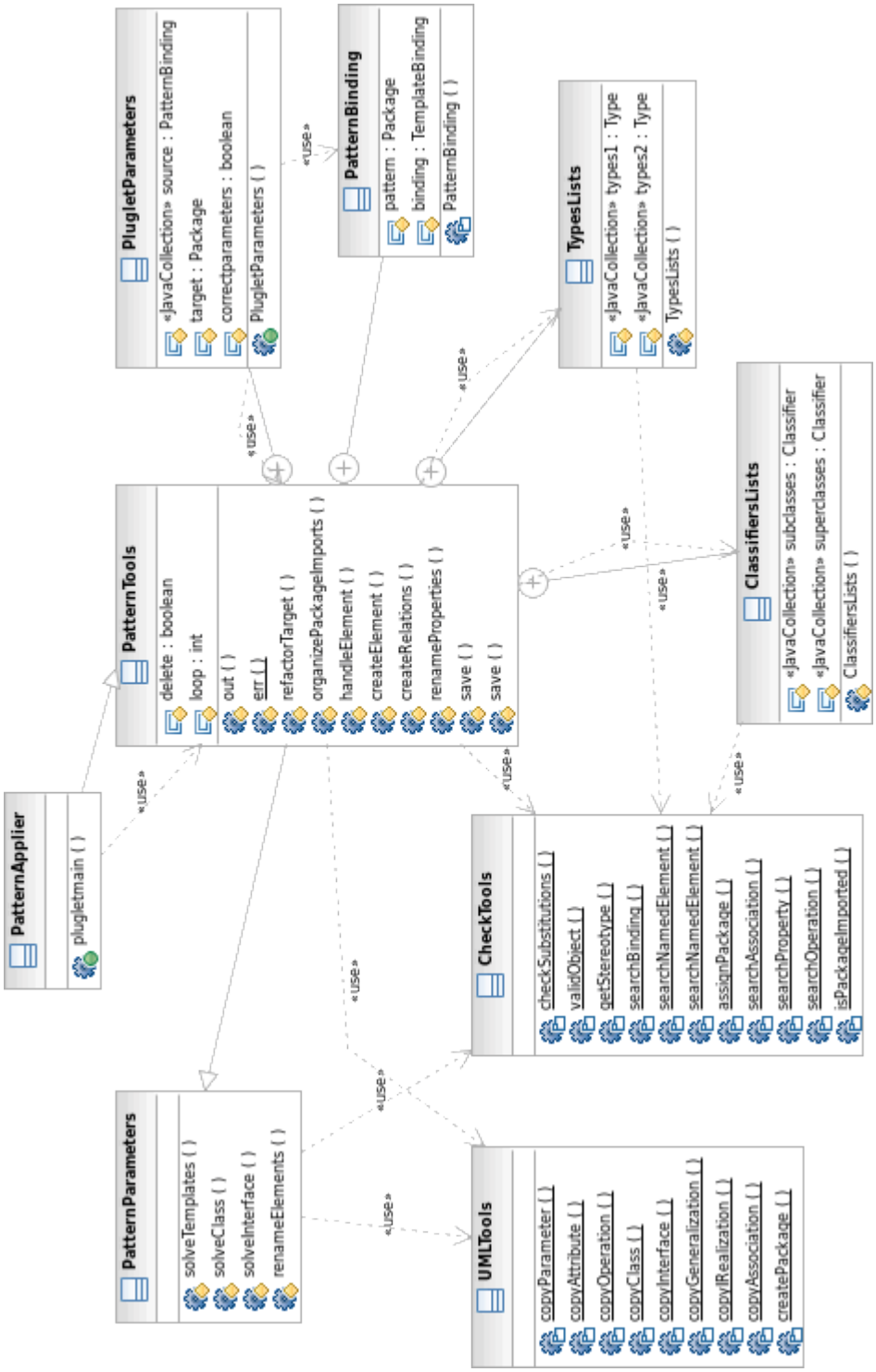


Figura 5.8 Diagrama de clases de la herramienta *PatternApplier*

Class PatternParameters

Los métodos de esta clase se encargan de resolver las transformaciones de las plantillas. También son heredados por la clase *PatternApplier* que es quien llama al método encargado de gestionar las substituciones, *solveTemplates( ).* Este método recibe como parámetro el vínculo de plantilla que contiene las substituciones. Gestiona los elementos que realizan las substituciones y los transforma usando métodos adecuados para el tipo de elemento, *solveClass( )* y *solveInterface( ).*

Esta clase contiene también, el método *renameElements( )* que realiza el renombre de los elementos a los que se le pase algún parámetro en su nombre.

Class UMLTools

Esta clase está implementada como una librería con métodos estáticos. Contiene métodos que pueden ser usados independientemente del resto de la aplicación. Los métodos sirven para copiar[5] elementos UML. En algunos casos, el método devuelve un elemento copia del elemento que se quiere copiar. En otros casos, es necesario pasar como parámetros dos elementos del mismo tipo, y el método modifica el segundo para que sea idéntico al primero.

Class CheckTools

Al igual que en la clase *UMLTools*, los métodos de esta clase con estáticos. Los distintos métodos de la clase permiten realizar comprobaciones o búsquedas de elementos dentro del modelo o de los patrones.

Por ejemplo, los métodos *searchBinding( ), searchNamedElement( ), searchAssociation( ), searchProperty( )* y *searchOperation( )* buscan elementos de distintos tipos en base a unos parámetros que pueden ser el mismo nombre o un elemento de referencia.

Métodos como *checkSubstitutions( ), validObject( ), isPackageImported( )* hacen comprobaciones para el correcto funcionamiento de aplicación. Y *getStereotype( )* y *assignPackage( )* proporcionan variables necesarias para las transformaciones.

5.3.3. Implementación de la herramienta

En este punto se presenta una descripción detallada de la implementación del código. Se tratará cada clase por separado, explicando cada método y resaltando los

[5] Véase punto de contratos de transformaciones

aspectos más significativos para el funcionamiento de la estos. Los códigos pueden seguirse en el anexo Código fuente.

5.3.3.1. Clase PatternApplier

Es la clase que contiene el *plugletmain( )* y controla el proceso de aplicación de los patrones. Al ejecutarse, se obtiene el dominio de edición *(EditingDomain)* de los modelos. Seguidamente se obtiene la pila de comandos *(CommandStack)* asociada al dominio para poder definir comandos que manipulen los modelos. Para crear un comando se extiende la clase *RecordingCommand* y se implementa el método abstracto *doExecute( )* que realiza los cambios requeridos en el modelo.

```
String undoLabel = "PatternApplier";

TransactionalEditingDomain editDomain =
                                    UMLModeler.getEditingDomain();

editDomain.getCommandStack().execute(new
                  RecordingCommand(editDomain,undoLabel) {

      protected void doExecute() {
```

Código 1. Obtención del dominio de edición y método *doExecture( )*

Seguidamente se analiza los elementos seleccionados en el momento de la ejecución, usando la clase *UMLModeler*. Se cran una instancia de la clase *PlugletParameter* que contendrá el modelo, los patrones, los vínculos de plantilla y la confirmación de si los elementos son correctos para el funcionamiento de la herramienta. De ser así, se procede a la aplicación del patrón; pero primero se llama el método *refactorTarget( )* para renombrar el modelo sobre el que se aplica el patrón, en caso de ser necesario.

```
List<EObject> selectedElements =
            UMLModeler.getUMLUIHelper().getSelectedElements();

PlugletParameters parameters = new
            PlugletParameters(selectedElements);

if (parameters.correctparameters) {

      refactorTarget(parameters.target);
```

Código 2. Parámetros de entrada de la herramienta

A continuación hay un bucle que recorre los elementos del tipo *PatternBinding* de la variable *parameters*. Estos contienen un patrón y un enlace de plantilla del modelo

hacia ellos. Los procesos que siguen se realizarán para cada patrón que se aplica sobre el modelo.

```
for (Iterator<PatternBinding> sourceiter =
            parameters.source.iterator();sourceiter.hasNext();){

      PatternBinding pattbind = sourceiter.next();
```

Código 3. Bucle sobre los patrones que se aplican modelo

El primer paso al aplicar un patrón es importar en el modelo los paquetes importados en el patrón que no estén importados ya en el modelo. Esto se hace con el método *organizePackageImports( ).* Después se resuelven las transformaciones relativas a las plantillas del patrón llamando al método *solveTemplates( ),* al que se le pasa como parámetro el vínculo de plantilla al patrón que se está aplicando en ese momento.

Para llevar a cabo las transformaciones especificadas por los estereotipos, se analizan todos los elementos del patrón, que son pasados como parámetro del método *handleElement( )* para que inicie las transformaciones en caso de ser necesario. Este bucle sobre los elementos del patrón se realiza dos veces para diferenciar los elementos individuales de las relaciones. Para ello se usa la variable *loop* que es incrementada tras el primer bucle.

```
TreeIterator<EObject> tree = pattbind.pattern.eAllContents();

for (Iterator<EObject> iter = tree;iter.hasNext();) {

      handleElement(iter.next(), parameters.target,
                                                pattbind.pattern);
}

tree = pattbind.pattern.eAllContents();
loop++;
```

Código 4. Bucle sobre elementos del patrón para analizar estereotipos

La última etapa de aplicación del patrón es la de renombrado de elementos. Para ello se recorren todos los elementos del modelo y se toman como parámetro del método *renameElements( )* que los renombrará si es necesario. Otro bucle sobre el modelo se realiza seguidamente para renombrar las propiedades que sean miembros de relaciones. Se usa el método *renameProperties( ).* Estos bucles tienen la misma forma del mostrado en el código 4; sólo cambia el método al que se llama, y que se realizan sobre los elementos del modelo en lugar de los del patrón. Tras esto termina la aplicación del patrón. Cuando todos han sido aplicados, se guarda el modelos con el método *save( ).*

5.3.3.2. Clase PatternTools

Variables

En esta clase están declaradas la variable entera *loop* para distinguir los elementos en los dos bucles de transformaciones a partir de estereotipo, y la variable booleana *delete* que indicará si un elemento tiene que ser eliminado.

Clase PatternBinding

Consta de dos variables: *pattern* del tipo *Package,* y *binding* del tipo *TemplateBinding.* Está diseñada para que cada instancia tenga como valores un patrón y el respectivo vínculo de plantilla que está enlazado del modelo al patrón.

Clase PlugletParameters

Se crea una instancia al lanzar el programa para agrupar los modelos que entran en juego y los vínculos de plantilla. Las variables declaradas son: *target* del tipo *package, source* que es una lista del tipo *PatternBinding,* y *correctparameters* que es booleana.

El constructor recibe como entrada una lista de objetos, que serán los elementos seleccionados en el momento de la ejecución, y tomará el primero como modelo sobre el que se aplica el patrón. Si esto es correcto, se asigna el modelo a la variable target. Después, recorre el modelo buscando vínculos de plantilla. Se crea una lista *source* para agrupar todos los vínculos con sus respectivos patrones. Si se ha encontrado algún vínculo de plantilla se asigna *true* a la variable *correctparameters.* Los paquetes de los patrones se obtienen a partir de las *Signatures* a las que enlazan los vínculos de plantilla.

```
TemplateBinding bind = (TemplateBinding) element;
Package patt = (Package) (bind.getSignature()).getOwner();
PatternBinding pattbind = new PatternBinding(patt, bind);
source.add(pattbind);
```

Código 5. Obtención de los paquetes de los patrones

Método refactorTarget

Es llamado tras la verificación de que los parámetros del pluglet son correctos. Identifica la ruta del modelo y ofrece la posibilidad de crear una copia nueva a la que aplicar el modelo en lugar de alterar el original. La petición se hace por pantalla usando el método *prompt( )* de la clase *Pluglet*.

Método organizePackageImports

Recorre los paquetes importados en el patrón. Para cada uno, comprueba con el método *isPackageImported( )* de la clase *CheckTools* si está ya importado en el modelo. Si no es así, lo importa.

Método handleElement

Comprueba si el objeto que se le pasa es válido para estar estereotipado con el método *validObject( ).* De ser así, busca los estereotipos aplicados, y si contiene alguno que especifique una transformación, llama al método *createElement( )* para que aplique la transformación.

Al llamar este método, hay que pasarle el paquete donde se ha de realizar la transformación. Normalmente será el modelo sobre el que se aplica el patrón; pero puede ser que se hayan creado paquetes dentro modelo por lo que se llamará al método *assignPackage* que devuelve el paquete exacto donde ha de crearse.

Método createRelations

Para las relaciones de tipo *Generalization* e *InterfaceRealization*, la creación mencionada en el método anterior, se realiza a parte, fuera del método *createElement.*

```
for (Iterator<Classifier> iter1 =
            classifiers.subclasses.iterator();iter1.hasNext();){

      Classifier subclass = iter1.next();

      for (Iterator<Classifier> iter2 =
      classifiers.superclasses.iterator();iter2.hasNext();){


      Classifier superclass = iter2.next();

      if (object instanceof InterfaceRealization) {

      UMLTools.copyIRealization((InterfaceRealization) object,
      (Class) subclass, (Interface) superclass, target);   }

      if (object instanceof Generalization) {

      UMLTools.copyGeneralization((Generalization) object,
      (Classifier)subclass, (Classifier) superclass, target); }
```

Código 6. Doble bucle de creación de relaciones entre clasificadores

Método createElement

En función del valor de la variable *loop* se filtran los elementos *(switch)* por si aún no es el momento de crearlos o eliminarlos. De tener que crear un elemento, se llama al método de la clase *UMLTools* que corresponda con el tipo del elemento. Si hay que eliminarlo, se comprueba con el correspondiente método de *CheckTools* si existe, y de ser así se elimina con el método *destroy( )* de la clase *Object.*

Para las relaciones, se usan las clases *TypesLists* y *ClassifiersLists* que agrupan en listas los extremos de las relaciones. Se crearán relaciones entre todos los elementos de las listas, realizando un doble bucle que las recorra para abarcar todas las posibilidades.

Método save

Los dos métodos *save( )* sirven para guardar las modificaciones realizadas sobre el modelo al que se le aplican los patrones. Uno guarda el modelo que se le pasa como parámetro. El otro lo guarda, pero como un recurso nuevo, a partir de una ruta que se le pasa como parámetro. Ambos usan la clase *UMLModeler.*

Clase TypesLists y Clase ClassifiersLists

Están compuestas por un par de listas de elementos *Type* la primera, y *Classifier* la segunda. Se crean instancias de estas clases cuando es necesario crear relaciones. Estas listas almacenas los extremos de las relaciones. Las listas *Type* se usan para asociaciones, mientras que las listas *Classifier* se usan para generalizaciones y realizaciones de interfaz.

```
List<Element> classifiers = relation.getRelatedElements();

if (((Classifier) classifiers.get(0)).isTemplateParameter()){
subclasses = CheckTools.searchBinding(classifiers.get(0),
                                                    package_);}
else{
      Classifier classifier1 = (Classifier)
      CheckTools.searchNamedElement(((Classifier)
                              classifiers.get(0)), package_);
      subclasses.add(classifier1);
```

Código 7. Obtención de extremos para las relaciones

En el código 7 se muestra cómo se obtiene los extremos de las relaciones. A partir de la relación se obtiene los elementos relacionados en el patrón con esa relación (*getRelatedElements( )).* Estos elementos pueden ser elementos parametrizados o no. Si son elementos parametrizados, se tomará como extremos los elementos del modelo que juegan los roles de dichos elementos parametrizados. Esto se hace con el método

searchBinding de la clase *CheckTools.* Si los elementos relacionados no son elementos parametrizados, se buscará en el modelo elementos con el mismo nombre que éstos. Estos elementos que buscamos, tienen que haber sido creados antes en el modelo. La búsqueda se realiza con el método *searchNamedElement.*

5.3.3.3. Clase PatternParameters

Método solveTemplate

Recibe como parámetro un vínculo de plantilla *(TemplateBinding).* Se obtienen las substituciones en una lista. A partir de las substituciones obtenemos el elemento parametrizado en el patrón y una lista con los elementos del modelo que juegan ese rol.

```
List<TemplateParameterSubstitution> list =
                              binding.getParameterSubstitutions();

for (Iterator<TemplateParameterSubstitution> iter =
                              list.iterator(); iter.hasNext();) {

TemplateParameterSubstitution subs = iter.next();
ParameterableElement formal =
                         subs.getFormal().getParameteredElement();
List<ParameterableElement> actuals = subs.getActuals();
```

Código 8. Obtención de parámetros formales y reales de las plantillas

Para cada elemento del modelo que juega un rol, se comprueba si el elemento del patrón que define ese rol está estereotipado con *rename.* De ser así, es renombrado. Posteriormente, según el tipo del elemento, se llama a un método que realiza las transformaciones del elemento del modelo. Estos métodos reciben como parámetros, el elemento formal (elemento del patrón que define el rol), y el elemento real (elemento que juega el rol).

Método solveClass

Recibe como parámetros una clase parametrizada del patrón que define un rol, y una clase del modelo que juega ese rol. Las características de la clase del patrón son copiadas a la clase de la substituye. Los métodos y atributos de la clase que define el rol son chequeados para valorar según el estereotipo si tienen que ser copiados o eliminados en la clase que la substituye.

Para los chequeos se usa el método *getStereotype( )* que proporciona un valor entero en función de estereotipo. Haciendo un *switch* de este valor, se manejan distintas opciones de transformación. Para realizar las copias de los atributos y las operaciones se usan los métodos *copyAttribute( )* y *copyOperation( )* de la clase *UMLTools.* Las

comprobaciones de existencia de estos atributos u operaciones en la clase del modelo se realizan con los métodos *searchProperty( )* y *searchOperation( ).*

Método solveInterface

Funciona igual que el método *solveClass( )*, pero para interfaces. Este método no contempla la copia de atributos de un elemento a otro.

Método renameElements

Recibe como parámetro un elemento, el modelo al que pertenece, y el patrón que se está aplicando al modelo. Se comprueba que el elemento tenga nombre y que no sea el extremo de una relación. De ser así, se analiza el nombre del elemento buscando el símbolo $, que indica la presencia de la cadena para introducir parámetros en el nombre de los elementos.

Cuando un elemento tiene que ser renombrado, se toma el nombre del elemento referencia y se busca en el patrón usando el método *searchNamedElement( )* de la clase *CheckTools.* Seguidamente se intenta buscar el elemento del modelo que sustituye al elemento de referencia en el patrón, pues este debía ser un elemento parametrizado. La búsqueda de los elementos en el modelo se realiza con el método *searchBinding( ).* Del elemento que localizado en el modelo se toma su nombre para insertarlo en elemento que se quería renombrar.

```
patternreference = CheckTools.searchNamedElement(referencestring,
                                                        pattern);

try {
      List <NamedElement> referenceactuals =
      CheckTools.searchBinding((Element) patternreference,
                               target);

      if (referenceactuals!=null) {
            main = referenceactuals.get(0).getName();
      }
} catch (NullPointerException e) {

      out.println("\n ERROR: Imposible renombrar elemento.");
      out.println("        Referecia no valida\n");
}
```

Código 9. Búsqueda de elemento de referencia y el elemento que lo substituye

5.3.3.4. Clase UMLTools

Método copyParameter

Recibe como parámetros dos elementos del tipo *Parameter.* El parámetro *parameter* es el elemento que hay que copiar. El parámetro *newparameter* se modifica para ser una copia de *parameter.* Usando métodos *get( )* y *set( )* de UML2 obtenemos las características de *paremeter,* y modificamos *newparameter.* No devuelve ningún parámetro pues se modifica directamente uno de los parámetros de entrada.

Método copyAttribute

Funciona como el método *copyParameter( ),* pero los elementos que se copian son del tipo *Property.*

Método copyOperation

Funciona como el método *copyParameter( ),* pero los elementos que copian son el tipo *Operation.* Para copiar los parámetros del elemento se obtienen la lista de parámetros del elemento a copiar . Por cada uno se crea un parámetro en elemento *newoperation*, y usando el método *copyParameter( )* se hacen idénticos a los parámetros del elemento *operation.*

```
for (Iterator<Parameter> iter =
      operation.getOwnedParameters().iterator(); iter.hasNext();)
{
      Parameter newparameter =
                  newoperation.createOwnedParameter(null, null);
      copyParameter(iter.next(), newparameter);
}
```

Código 10. Copiado de parámetros de un método a otro

Método copyClass

Recibe como parámetros una clase *class_* y un modelo *target*. Devuelve una copia de la clase que se le pasa como parámetro, y que ha sido creada en el modelo pasado como parámetro.

Se crea una clase *copy* en el modelo mediante *target.createOnedClass( ).* Usando métodos *get( )* y *set( )* se copian las características de la clase *class_* en la nueva clase *copy.* Los atributos y métodos de la clase *class_* se copian usando la misma mecánica usada en el código 10, pero creando operaciones y atributos, y usando *copyAttribute( )* y *copyOperation( ).*

Método copyInterface

Funciona igual que el método *copyClass( )* pero para elementos del tipo *Interface*. La interfaz se crea mediante *target.createOwnedInterface( ).* Además de las características, se copian los métodos.

Método copyGeneralization

Se copia la generalización *general*. También recibe como parámetros, los clasificadores relacionados con la generalización, *subclass* y *superclass*. Se crea una generalización *copy* mediante *subclass.createGeneralization(superclass).* Se copian las características de *general* a *copy*, y se devuelve *copy*.

Método copyInterfaceRealization

Funciona igual que el método *copyGeneralization( ),* pero con elementos del tipo *InterfaceRealization.* Se usa *subclass.createInterfaceRealization(irealiza.getName( ), superclass)* para crear la realización.

Método copyAssociation

Se crea una asociación igual a la asociación *association* entre los elementos *copytype1* y *copytype2.* Se usa *copytype2.createAssociation( )* para crear la asociación copy. Las características se copian con métodos *get( )* y *set( )* de UML2.

Los miembros de la nueva asociación se modifican para que tengan las mismas características que los de la asociación *association* usando el método *copyAttribute( ).* Para ello, necesitamos los miembros de *association* que se obtienen mediante *List<Property> members = association.getMemberEnds( ).* De igual manera se obtiene los miembros de la nueva asociación *copy.* Es necesario volver a definir los tipos y los nombres de los miembros de la nueva asociación tras *copyAttribute( ),* porque habrán adoptado los de los elementos que están relacionados con la asociación *association.*

Método createPackage

Se crea en el paquete *target*, un paquete con las mismas características que *package_*. Se usa t*arget.createNestedPackage(package_.getName( ))* para crear el paquete.

5.3.3.5. Clase CheckTools

Método checkSubstitutions

Recibe como parámetro un vínculo de plantilla *binding (TemplateBinding).* A partir de *binding* se obtienen, por un lado los parámetros de plantilla del patrón mediante *getSignature( )* y *getOwnedParameters( )*; por otro lado, se obtienen las substituciones en el modelo usando *getParameterSubstitutions( ).*

Un bucle recorre los parámetros de plantilla y comprueba si el elemento parametrizado está estereotipado con *optional.* Si lo está, se pasa a comprobar el siguiente parámetro de plantilla. Si no lo estaba, se recorren las substituciones. Para cada una se comprueba si el elemento parametrizado al que substituyen coincide con el elemento parametrizado del parámetro de plantilla del primer bucle. Estas comprobaciones de coincidencia de elementos se realizan comprobando la igualdad de las identidades de los recursos, *eResource().getURIFragment( ).* Si no se encuentra ninguna coincidencia entre los parámetros formales de las substituciones, significará que ningún elemento del modelo está substituyendo un rol del patrón, que al no estar estereotipado con *optional*, debería ser substituido. Cuando esto ocurre se devuelve directamente *false* como resultado del método.

```
for(Iterator<TemplateParameterSubstitution> iter2 =
                  substitutions.iterator();iter2.hasNext();) {

subs = iter2.next();
formalsubs = subs.getFormal().getParameteredElement();
formalsubsID = formalsubs.eResource().getURIFragment(formalsubs);

if (formalID.equals(formalsubsID)) {
      check = true;
}
```

Código 11. Comprobación de igualdad de identidades entre parámetros formales de un parámetro de plantilla y una substitución

Método validObject

Comprueba si el elemento es válido para estar estereotipado. Esto es que sea instancia de *Element*, y que no sea un elemento parametrizado, pues las transformaciones de estos se tratan de manera distinta. Esta comprobación se realiza mediante *(ParameterableElement) object).isTemplateParameter()).*

Método getStereotype

Se obtienen los estereotipos aplicados al elemento que se le pasa como parámetro mediante *getAppliedStereotypes( ).* Se comprueban sus nombres, y si coinciden con alguno del perfil *PatternSpec* se devuelve un valor entero según el estereotipo.

Método searchBinding

El método busca en el modelo *target* un elemento o varios que substituyan al elemento *element* en el patrón. *element* deberá ser un elemento parametrizado. Se recorren los elementos de *target* en busca de vínculos de plantilla *(TemlateBinding).*

Para todos las substituciones del vínculo de plantilla se comprueba si la identidad del parámetro de plantilla al que están enlazadas coincide con el del parámetro de plantilla al que pertenece *element*. Si esto es así, significará que se ha encontrado una substitución para el elemento parametrizado *element*, y se devolverán los valores reales que lo substituyen. La mecánica de estas comprobaciones es como en el método *checkSubstitutions( )*, con la diferencia de que en este se comprueban las identidades de los parámetros de plantilla en lugar de los elementos parametrizados.

Métodos searchNamedElement

Hay dos métodos *searchNamedElement( ).* Se diferencian en los parámetros de entrada, pero el resultado que se busca es el mismo. Localiza un elemento concreto en el modelo *target.* El elemento se busca por su nombre que según el parámetro de entrada deberá coincidir con otra cadena, o con el nombre de otro elemento.

```
for (Iterator<EObject> iter = modelElements;iter.hasNext();) {

      EObject eobject = (EObject) iter.next();

      if (eobject instanceof NamedElement) {

            String name = ((NamedElement) eobject).getName();

            try{
                  if (name.equals(element.getName())){
                        search = (NamedElement) eobject;
                  }
            } catch (NullPointerException e) { }
      }
```

Código 12. Búsqueda de un elemento a partir de una cadena

Cuando no se encuentra ningún elemento se devuelven distintos resultados. Si la búsqueda se realizó a partir de una cadena, se devolverá *null.* Mientras que si la búsqueda se hizo a partir de un elemento, se devolverá el propio elemento.

Método assignPackage

El método asigna el qué paquete perteneciente al modelo target se tiene que crear un elemento estereotipado *object* del patrón *pattern.*

Primero se obtiene el paquete *owner* que contiene al elemento *object* por encima en la estructura del patrón; *getNearestPackage()* para elementos y *getNestingPackage( )* si *object* es una instancia de paquete. Después, se comprobará si la identidad de este paquete obtenido es la misma que la de *pattern*. De ser así, significará que *object* pertenece al paquete más arriba en la estructura del patrón, es decir, el propio modelo del patrón *pattern.* Por lo tanto, el elemento que se vaya a crear en el modelo sobre el que se aplica el patrón deberá ser el paquete más alto en la estructura del modelo, es decir *target.*

Si no coincidieron las identidades. El paquete que se devolverá será el encontrado en el modelo *target* mediante el método *searchNamedElement( )* y usando como parámetro de búsqueda el paquete *owner.*

Método searchAssociation

El método buscará una asociación igual a *association* en el modelo *target* que relacione los tipos *type1* y *type2.* Se recorre todo el modelo *target* buscando asociaciones. Para cada asociación *testasso* se comprueba si los extremos tienen las mismas identidades que *type1* y *type2.* Se obtienen mediante *testasso.getEndTypes( ),* y las identidades con *eResource().getURIFragment( ).*

Cuando se encuentre una asociación que tenga los mismos extremos que *association,* se procederá a examinar la igualdad de las características de los miembros de ambas asociaciones. Estos se obtienen mediante *testasso.getMemberEnds( ).* Si se cumplen las condiciones exigidas se devuelve como resultado *testasso.* Cuando no se encuentre ninguna asociación se devolverá *null.* Las condiciones para la igualdad serán explicadas en el punto de Contratos de las transformaciones.

Método searchProperty

Se comprueba si en la lista de propiedades *properties* hay alguna que cumpla ciertas condiciones de igualdad con *attribute*. Se recorre la lista haciendo las comprobaciones sobre cada elemento de ésta. Si alguna cumple las condiciones será devuelta como resultado del método. En caso contrario, se devuelve null.

Método searchOperation

El funcionamiento es el mismo que en el método *searchProperty( )* pero para elementos del tipo *Operation.*

Método isPackageImported

Comprueba si el paquete *pack* está importado en el modelo *target.*

Se obtienen las importaciones *(PackageImports)* usando *getPackageImports( ).* De cada una se obtiene el paquete *modelpack* que importa con *getImportedPackage( ).* Se comprobará si las identidades de *pack* y *modelpack* coinciden, es decir, son el mismo paquete. De ser así, se devuelve *true* como resultado del método. De no encontrarse el paquete importado, se devolverá *false.*

5.4. Contratos de las transformaciones

En el presente punto se tratarán los acuerdos y restricciones que se han considerado para las especificaciones de las transformaciones de UML. Se indicarán las características de igualdad a la hora de copiar elementos y chequear la existencia de elementos semejantes.

5.4.1. Copias de elementos UML

Cuando las transformaciones requieren la creación de un elemento nuevo en el modelo sobre el que se aplica el patrón, o un elemento del modelo juega un rol dentro del patrón, tiene lugar un proceso de copiado de características de elementos. Es decir, si hay que realizar una copia de un elemento del patrón, se creará un elemento del mismo tipo en el modelo, y se copiarán las características del elemento del patrón al elemento del modelo. De igual manera ocurrirá con los elementos del modelo que jueguen algún rol. Estos copiarán las características del elemento del patrón al que substituyen.

A continuación se presentan los acuerdos de igualdad para cada tipo de elemento UML que se ha contemplado en la herramienta. En las tablas se presenta en las columnas en blanco la característica que se copia, mientras que en las grises se indica el método del API usado para obtener o establecer el valor de la característica.

Copia de parámetros *(org.eclipse.uml2.uml.Parameter)*

Característica	**Método**	**Característica**	**Método**
Nombre	*setName( )*	Límite inferior	*setLower( )*
Tipo	*setType( )*	Límite superior	*setUpper( )*
Dirección	*setDirection( )*	Es ordenado	*setIsOrdered( )*
Valor por defecto	*setDefaultValue( )*	Es único	*setIsUnique( )*
Visibilidad	*setVisibility( )*	Es excepción	*setIsException( )*
Efecto	*setEffect( )*	Es stream	*setIsStream( )*

Tabla 4. Características en el copiado de parámentros

Copia de atributos *(org.eclipse.uml2.uml.Property)*

Característica	Método	Característica	Método
Nombre	*setName( )*	Es ordenado	*setIsOrdered( )*
Tipo	*setType( )*	Es estático	*setIsStatic( )*
Valor por defecto	*setDefaultValue( )*	Es único	*setIsUnique( )*
Visibilidad	*setVisibility( )*	Es derivado	*setIsDerived*
Límite inferior	*setLower( )*	Es unión de derivados	*setIsDerivedUnion( )*
Límite superior	*setUpper( )*	Es stream	*setIsStream( )*
Es rama	*setIsLeaf( )*		

Tabla 5. Características en el copiado de atributos

Copia de operaciones *(org.eclipse.uml2.uml.Operation)*

Característica	Método	Característica	Método
Nombre	*setName( )*	Es abstracta	*setIsAbstracj( )*
Visibilidad	*setVisibility( )*	Es duda	*setIsQuery( )*
Es rama	*setIsLeaf( )*	Concurrencia	*setConcurrency( )*
Es estática	*setIsStatic( )*	Parámetros	*copyParamete( )*

Tabla 6. Características en el copiado de operaciones

Copia de clases *(org.eclipse.uml2.uml.Class)*

Característica	Método	Característica	Método
Nombre	*setName( )*	Visibilidad	*setVisibility( )*
Es abstracta	*setIsAbstracj( )*	Es activa	*setIsActive( )*
Atributos	*copyAttribute( )*	Es rama	*setIsLeaf( )*
Operaciones	*copyOperation( )*		

Tabla 7. Características en el copiado de clases

Los atributos de las clases que son miembros de una asociación no son copiados. Estos se crearán al copiar la asociación entre las clases implicadas. Cuando se copian las características de una clase parametrizada del patrón a una clase del modelo que substituye a la clase del patrón, el nombre de la clase del modelo se mantendrá, siempre que no se estereotipe la clase del patrón con *rename.*

Copia de interfaces *(org.eclipse.uml2.uml.Interface)*

Característica	Método	Característica	Método
Nombre	*setName( )*	Visibilidad	*setVisibility( )*
Operaciones	*copyOperation( )*	Es rama	*setIsLeaf( )*
Es abstracta	*setIsAbstracj( )*	Miembros	*copyAttribute( )*

Tabla 8. Características en el copiado de interfaces

Cuando se copian las características de una interfaz parametrizada del patrón a una interfaz del modelo que substituye a la interfaz del patrón, el nombre de la interfaz del modelo se mantendrá, siempre que no se estereotipe la interfaz del patrón con *rename.*

Copia de generalizaciones *(org.eclipse.uml2.uml.Generalization)*

Se considera si la generalización es sustituible, *(setIsSubstitutable( ))*.

Copia de realizaciones (org.eclipse.uml2.uml.InterfaceRealization)

Además del nombre, se considera la visibilidad *(setVisibility( ))* y el mapeo *(setMapping( ))*.

Copia de asociaciones (org.eclipse.uml2.uml.Association)

Característica	Método	Característica	Método
Nombre	*setName( )*	Visibilidad	*setVisibility( )*
Operaciones	*copyOperation( )*	Es rama	*setIsLeaf( )*
Es abstracta	*setIsAbstracj( )*	Miembros	*copyAttribute( )*

Tabla 9. Características en el copiado de asociaciones

Además de copiar las características de los miembros de la asociación, también se establecen sus nombres en base a su tipo, si son navegables *(isNavigable( ))* y sus agregaciones *(getAggregation( ))*.

Copia de realizaciones (org.eclipse.uml2.uml.InterfaceRealization)

Además del nombre, se considera la visibilidad *(setVisibility( )*.

5.4.2. Igualdad entre elementos

Las comprobaciones de existencia de elementos, en el modelo al que se aplica el patrón, iguales o similares a elementos del patrón, se realizan con la clase *CheckTools.*

A continuación se indican las características que se han de satisfacer para cada tipo de elemento que se ha considerado. Satisfechas estas comprobaciones, los elementos comparados se consideran iguales.

Igualdad entre asociaciones

Característica	Método
Tipos relacionados	*getEndType( )*

Navegabilidad de los miembros	*isNavigable( )*
Agregación de los miembros	*getAggregation( )*
Límites de los miembros	*lowerBound( ), upperBound( )*

Tabla 10. Características de igualdad entre asociaciones

Igualdad entre propiedades

Característica	**Método**	**Característica**	**Método**
Nombre	*setName( )*	Visibilidad	*getVisibility( )*
Tipo	*getType( )*	Es estático	*isStatic( )*

Tabla 11. Características de igualdad entre propiedades

Igualdad entre operaciones

Característica	**Método**	**Característica**	**Método**
Nombre	*setName( )*	Visibilidad	*getVisibility( )*
Tipo	*getType( )*	Es estático	*isStatic( )*
Es abstracta	*isAbstract( )*		

Tabla 12. Características de igualdad entre operaciones

Capítulo 6

Ejemplos de uso

6.1. Patrón *Control Loop*

El siguiente ejemplo mostrará la aplicación del patrón *Control Loop* a un modelo.

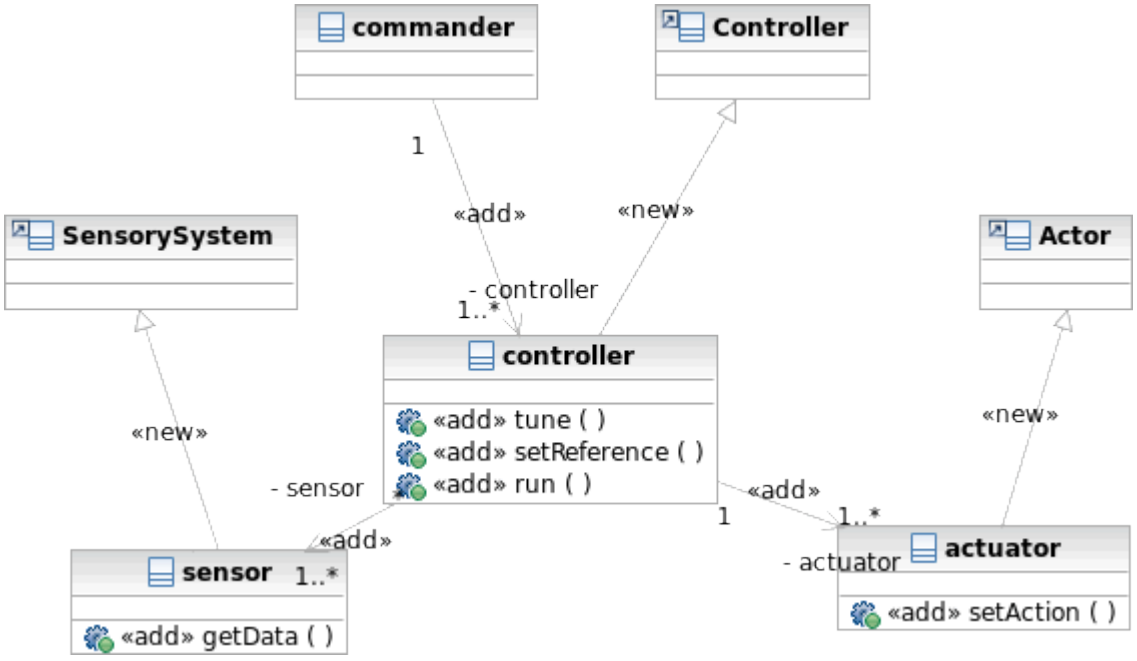


Figura 6.1 Diagrama de clases del patrón *Control Loop*

El patrón representa la estructura de un sistema de control en bucle cerrado. La clase *commander* representa el dispositivo que puede manejar distintos controladores dependiendo de los distintos procesos que se desee controlar. La clase *controller* representa a los controladores encargados de analizar los datos de sensores y eliminar el error entre la referencia y la señal real, mandando al actuador funcionar según las necesidades. La clase *controller* generaliza la clase *Controller* de OASys.

El controlador dispone de distintos sensores, clase *sensor,* para obtener los datos reales de las variables del proceso. La clase *sensor* generaliza la clase *SensorySystem* de OASys. Por otro lado, el controlador manda a los actuadores, clase *actuator* entrar en acción para que realicen la acción correctora en el proceso. La clase *actuator* generaliza a la clase *Actor* de OASys.

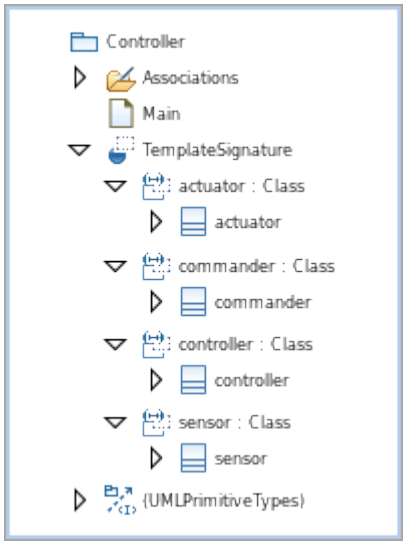


Figura 6.2 Estructura del patrón *Control Loop*

En la figura 6.2 se muestra la estructura del patrón. Se pueden cuatro parámetros de plantilla para definir los distintos roles que jugarán los elementos del sistema al que se le aplique el patrón. Las clases *actuator, commander, controller* y *sensor* son clases parametrizadas.

Para el ejemplo, se usará un sistema para regular la temperatura.

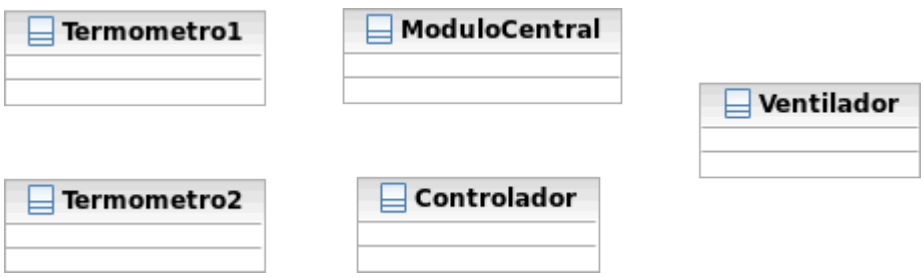


Figura 6.3 Elementos del sistema ejemplo de regulación de temperatura

El sistema consta de dos tipos de termómetros, para realizar las medidas, representados por las clases Termómetro1 y Termómetro2. La clase Ventilador representa los elementos encargados de realizar la acción correctora de temperatura. El elemento Controlador realiza el rol de *Controller,* y ModuloCentral el de *Commander.* Tras añadir al modelo un vínculo de plantilla al patrón, se realizan las substituciones como se muestra en la figura 6.4.

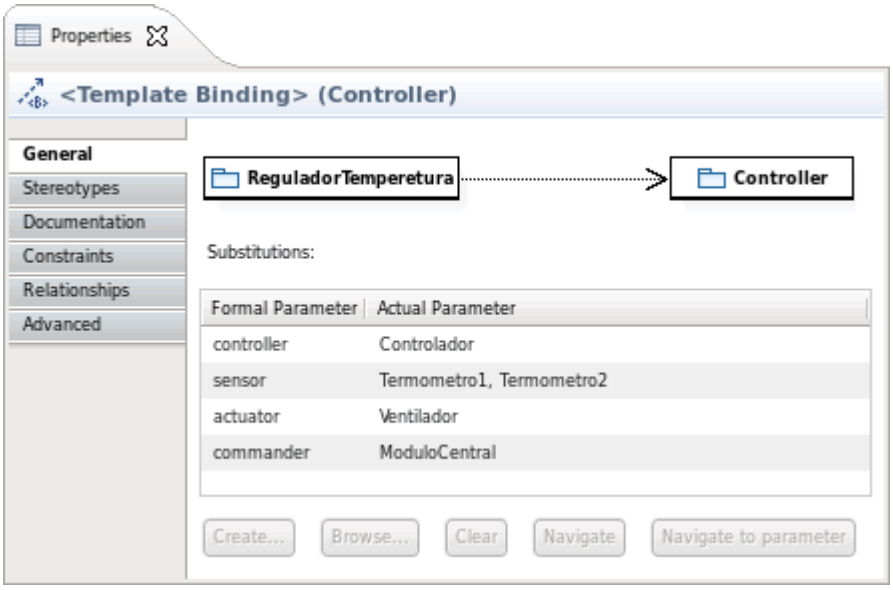


Figura 6.4 Substituciones del Regulador de temperatura en el patrón

Tras ejecutar la herramienta, se obtiene el resultado representado en la Figura 6.5

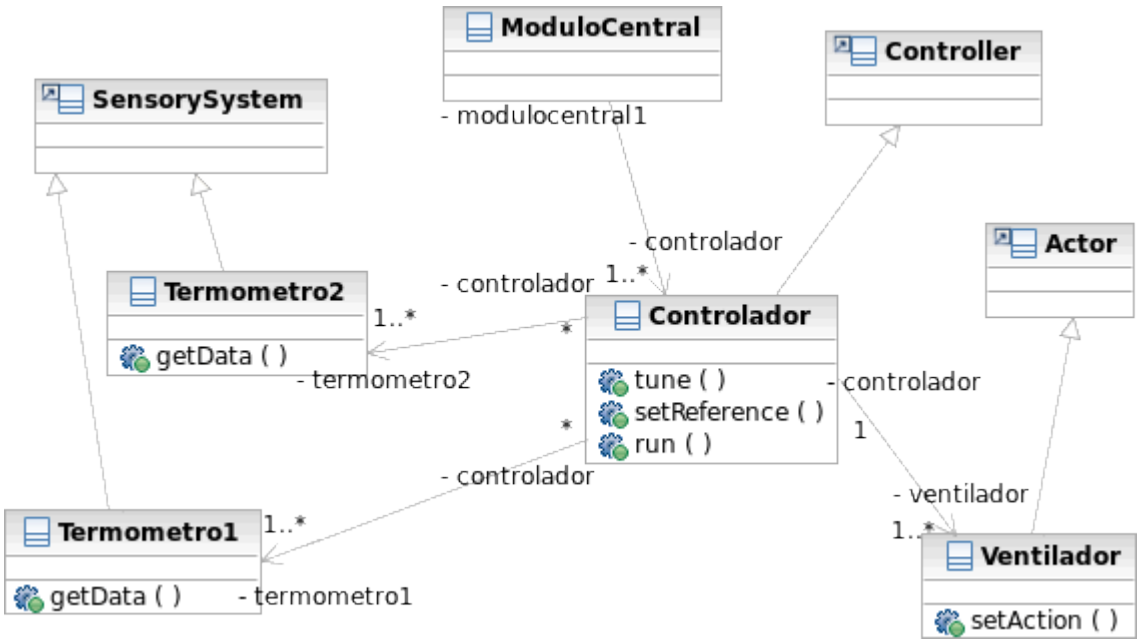


Figura 6.5 Sistema regulador de temperatura tras la aplicación del patrón de control

La salida por consola muestra que la aplicación del patrón se ha realizado correctamente, indicando el modelo del patrón, el modelo sobre el que se aplica el patrón, y las sustituciones de roles que se han realizado.

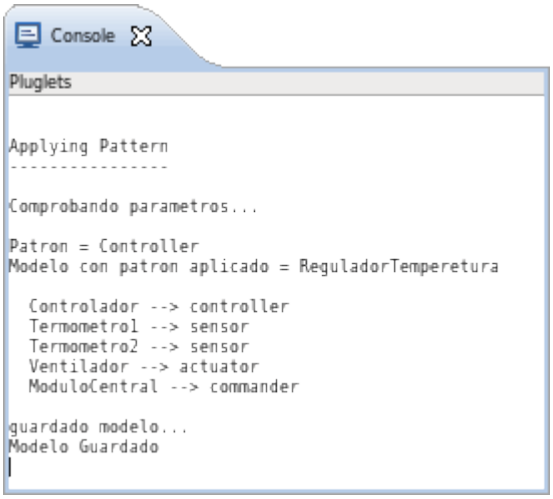


Figura 6.6 Salida por consola tras la aplicación del patrón de control

Los elementos del modelo han substituido a los elementos parametrizados del patrón y han adquirido sus operaciones como propias. Las relaciones entre los elementos se han creado de la misma manera que están especificadas en el patrón para los roles definidos. Por último, se han generalizado los distintos elementos de OASys.

6.2. Patrón *Singleton*

El siguiente ejemplo mostrará la aplicación de un patrón de diseño típico, el patrón *Singleton*, que ha sido capturado en un modelo usando la semántica diseñada para la herramienta *PatternApplier.*

El patrón de diseño singleton (instancia única) está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

El patrón singleton se implementa, como se muestra en la figura 6.7, creando en nuestra clase un método que crea una instancia del. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado). El patrón singleton provee una única instancia global gracias a que:

- La propia clase es responsable de crear la única instancia.
- Permite el acceso global a dicha instancia mediante un método de clase.

- Declara el constructor de clase como privado para que no sea instanciable directamente.

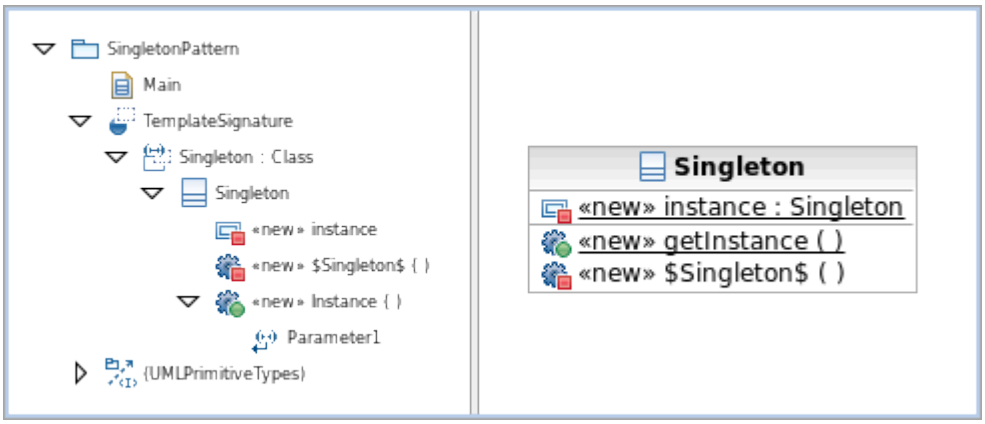


Figura 6.7 Diagrama de clases y estructura del patrón *Singleton*

En la estructura del patrón se observa que la clase Singleton es un elemento parametrizado. Cuando se aplique el patrón, una clase substituirá a la clase *Singleton.* Se usará una clase cualquiera *Demo* para mostrar el resultado de la aplicación del patrón.

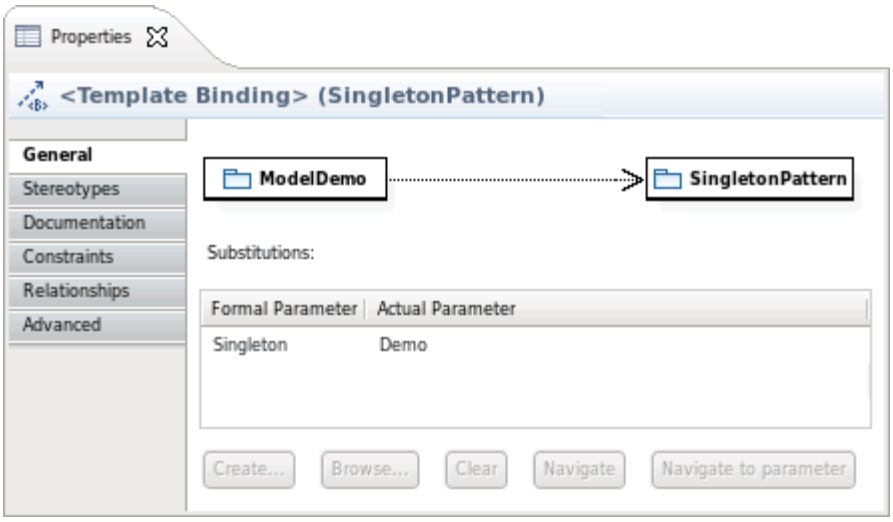


Figura 6.8 Substitución de la clase *Demo* en el patrón *Singleton*

El elemento es transformado en una clase de instancia única.

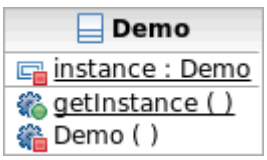


Figura 6.9 Clase *Demo* de instancia única

Por consola, se comprueba que la aplicación de la herramienta se ha realizado correctamente.

Figura 6.10 Clase *Demo* de instancia única

Capítulo 7

Conclusiones y líneas futuras

7.1. Conclusiones

Como resultado del proyecto se ha obtenido un lenguaje para la especificación de patrones en modelos UML2, y una herramienta para la aplicación de estos patrones durante la fase de diseño de sistemas en el entorno de desarrollo RSA.

Lenguaje para la especificación de patrones

El uso de los elementos de UML2 y los estereotipos ha permitido reducir el problema de la creación y desarrollo de patrones al diseño de éstos. En comparación con las otras tecnologías estudiadas en el proyecto, la solución adoptada para capturar los patrones, por un lado resulta más sencilla al limitarse a la concepción del patrón y su captura mediante elementos de modelado de forma gráfica. De esta manera, la solución para la especificación de los patrones está perfectamente orientada al usuario de los patrones, puesto que los patrones serán usados por diseñadores qué podrán capturar los patrones de la misma manera que modelan sistemas.

Por otro lado, debido a que los patrones son modelos, el producto creado con cada patrón no se limita a un útil para realizar una determinada función. Primero, la representación del patrón con su diagrama permite un mayor entendimiento de la

estructura y funcionamiento de los sistemas que se capturan. Y segundo, al ser modelos UML, se dispone de unos activos que pueden ser usados en diferentes procesos de desarrollo basado en modelos. Un ejemplo, es la aplicación de los patrones con la herramienta *PatternApplier.*

Herramienta para aplicación de patrones *PatternApplier*

Con la implementación de la herramienta como un pluglet, y el uso de la plataforma de modelado de Rational, se ha conseguido que el proceso de diseño en el entorno de RSA se realice de manera continua. El trabajo se realiza sobre los modelos abiertos en el entorno de modelado, incluidos los patrones, y la ejecución se realiza en la misma perspectiva.

Desde el punto de vista de la funcionalidad, se ha cumplido con las necesidades que se exigían para la transformación de los modelos. El trabajo directo de la herramienta sobre los elementos los modelos, haciendo uso de las librerías UML2 de eclipse, ha permitido una buena interpretación del patrón; y en consecuencia, se han podido definir las transformaciones hasta cierto nivel del metamodelo de acuerdo a las necesidades.

Conclusiones generales

Se concluye que los objetivos del proyecto se han logrado satisfactoriamente. La herramienta desarrollada además de cumplir con los requisitos, resulta fácil y cómoda para manejar. Su uso puede aportar beneficios en del desarrollo de sistemas basados en modelos y proporciona una manera sencilla diseñar soluciones de diseño reutilizables.

Además la herramienta está abierta a la extensión de las transformaciones que se pueden especificar, los elementos involucrados y las características que adquieren mayor relevancia dentro de los diseños.

Como beneficio colateral, el desarrollo de estas herramientas y el análisis realizado para el mismo han aportado un conocimiento valiosísimo sobre el papel que los modelos, y los metamodelos (como por ejemplo las ontologías de dominio, el propio lenguaje de especificación de modelos) deben jugar en la ingeniería de sistemas basada en modelos.

7.2. Líneas futuras

Se pueden identificar dos líneas de acción futuras sobre el trabajo desarrollado:

La primera es la extensión de la herramienta y el perfil para la especificación de las transformaciones. Los elementos UML contemplados en las transformaciones realizadas por la herramienta son limitados. De igual manera, se han definido un número de

estereotipos de acuerdo a las transformaciones más básica necesarias para la aplicación de los patrones. La ampliación del funcionamiento a otros elementos o la adición de nuevos estereotipos, requerirá las precisas extensiones del código de la herramienta.

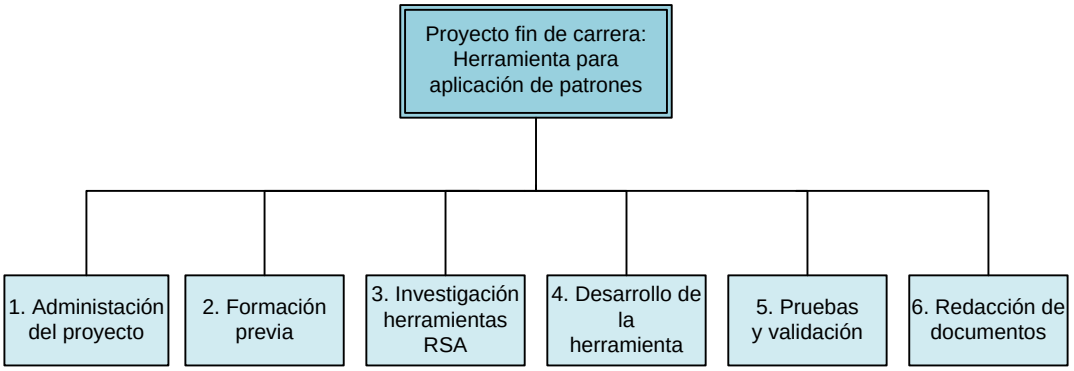
La segunda línea consistiría en independizar la herramienta de aplicación de patrones del lenguaje empleado para especificarlos, usando como vehículo intermedio un lenguaje formal de especificación de transformaciones, como Atlas.

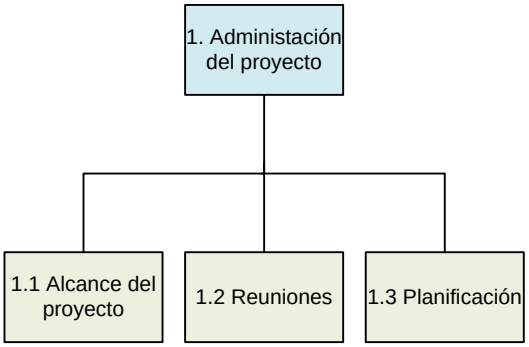
En cuanto a la aplicación del trabajo ya desarrollado, queda pendiente la ampliación de la librería de patrones, incluyendo otras soluciones de control e inteligencia artificial para la construcción sistemas de control inteligentes, tales como bucles feed-forward, control predictivo basado en modelos, arquitecturas tipo pizarra, etc. Así mismo, el lenguaje de especificación de patrones será empleado en los próximo años para la especificación de los bucles de control basados en conocimiento que constituyen uno de los pilares de ASys para la ingeniería de sistemas autónomos.

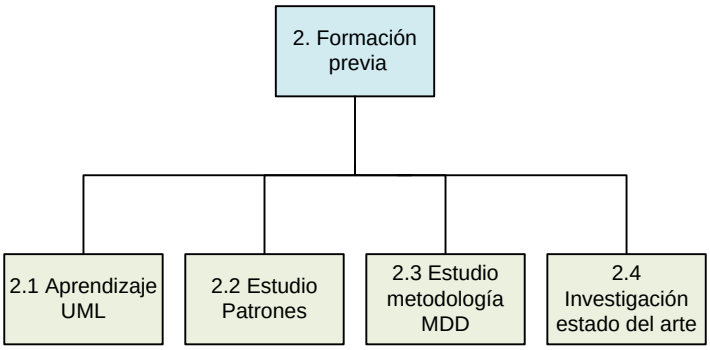
Capítulo 8

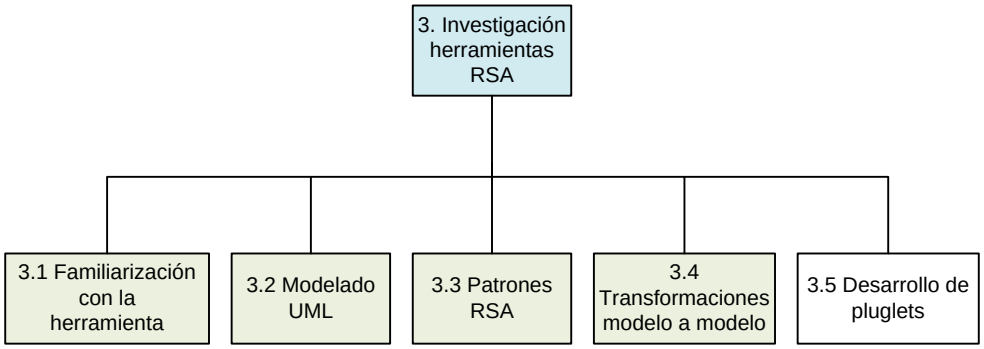
Planificación del proyecto

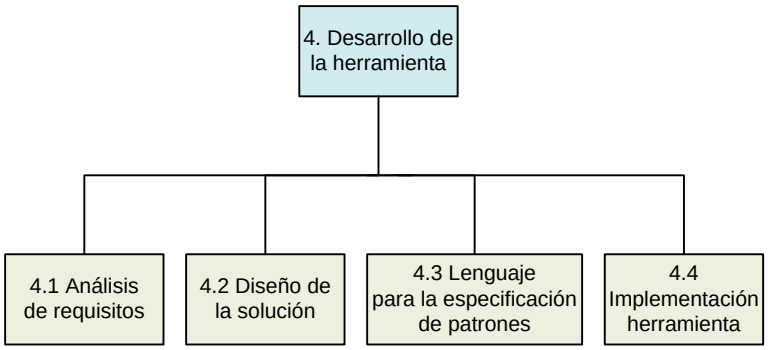
8.1. Estructura de descomposición del trabajo

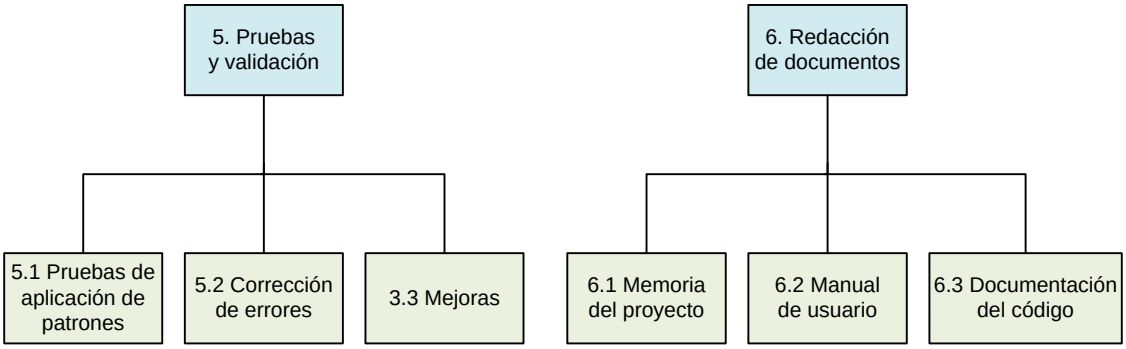





2. Formación previa
2.1 Aprendizaje UML
2.2 Estudio Patrones
2.3 Estudio metodología MDD
2.4 Investigación estado del arte

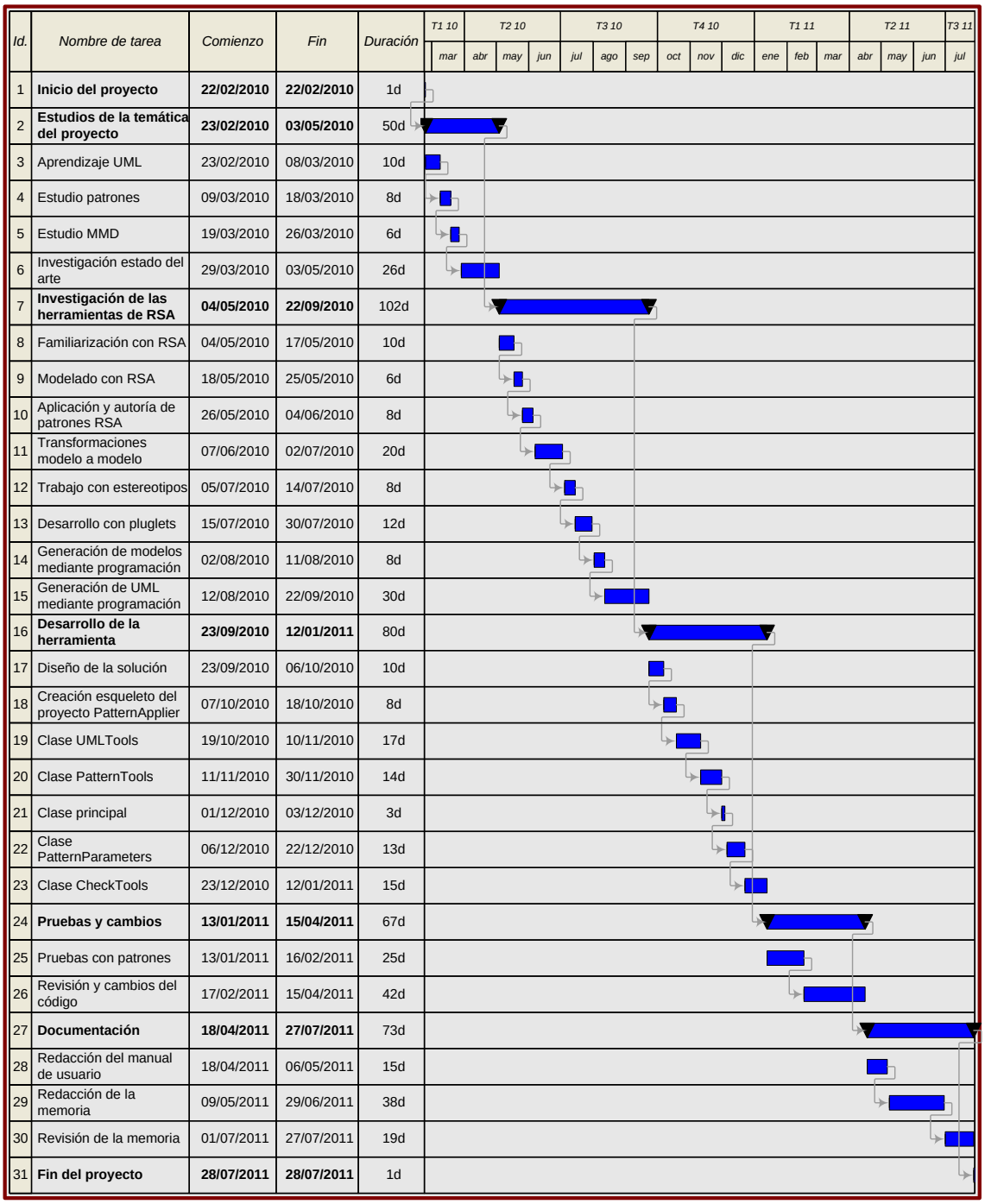

3. Investigación herramientas RSA
3.1 Familiarización con la herramienta
3.2 Modelado UML
3.3 Patrones RSA
3.4 Transformaciones modelo a modelo
3.5 Desarrollo de pluglets


4. Desarrollo de la herramienta
4.1 Análisis de requisitos
4.2 Diseño de la solución
4.3 Lenguaje para la especificación de patrones
4.4 Implementación herramienta


5. Pruebas y validación
6. Redacción de documentos
5.1 Pruebas de aplicación de patrones
5.2 Corrección de errores
3.3 Mejoras
6.1 Memoria del proyecto
6.2 Manual de usuario
6.3 Documentación del código

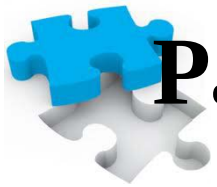
8.2. Diagrama de Gantt

El proyecto se ha realizado en el Laboratorio de Sistemas Autónomos ASLab durante 16 meses, en los que se distinguen 5 fases principales: Estudios previos, Investigación de la herramienta RSA, Desarrollo de la herramienta, Pruebas y Documentación. A continuación se presenta el diagrama de Gantt que muestra el desarrollo de las distintas actividades a lo largo de la realización del proyecto.



Id.	Nombre de tarea	Comienzo	Fin	Duración
1	**Inicio del proyecto**	**22/02/2010**	**22/02/2010**	1d
2	**Estudios de la temática del proyecto**	**23/02/2010**	**03/05/2010**	50d
3	Aprendizaje UML	23/02/2010	08/03/2010	10d
4	Estudio patrones	09/03/2010	18/03/2010	8d
5	Estudio MMD	19/03/2010	26/03/2010	6d
6	Investigación estado del arte	29/03/2010	03/05/2010	26d
7	**Investigación de las herramientas de RSA**	**04/05/2010**	**22/09/2010**	102d
8	Familiarización con RSA	04/05/2010	17/05/2010	10d
9	Modelado con RSA	18/05/2010	25/05/2010	6d
10	Aplicación y autoría de patrones RSA	26/05/2010	04/06/2010	8d
11	Transformaciones modelo a modelo	07/06/2010	02/07/2010	20d
12	Trabajo con estereotipos	05/07/2010	14/07/2010	8d
13	Desarrollo con pluglets	15/07/2010	30/07/2010	12d
14	Generación de modelos mediante programación	02/08/2010	11/08/2010	8d
15	Generación de UML mediante programación	12/08/2010	22/09/2010	30d
16	**Desarrollo de la herramienta**	**23/09/2010**	**12/01/2011**	80d
17	Diseño de la solución	23/09/2010	06/10/2010	10d
18	Creación esqueleto del proyecto PatternApplier	07/10/2010	18/10/2010	8d
19	Clase UMLTools	19/10/2010	10/11/2010	17d
20	Clase PatternTools	11/11/2010	30/11/2010	14d
21	Clase principal	01/12/2010	03/12/2010	3d
22	Clase PatternParameters	06/12/2010	22/12/2010	13d
23	Clase CheckTools	23/12/2010	12/01/2011	15d
24	**Pruebas y cambios**	**13/01/2011**	**15/04/2011**	67d
25	Pruebas con patrones	13/01/2011	16/02/2011	25d
26	Revisión y cambios del código	17/02/2011	15/04/2011	42d
27	**Documentación**	**18/04/2011**	**27/07/2011**	73d
28	Redacción del manual de usuario	18/04/2011	06/05/2011	15d
29	Redacción de la memoria	09/05/2011	29/06/2011	38d
30	Revisión de la memoria	01/07/2011	27/07/2011	19d
31	**Fin del proyecto**	**28/07/2011**	**28/07/2011**	1d

Anexo 1. Manual de Usuario



PatternApplier

Manual de Usuario

Índice:

Índice de ilustraciones:

Índice de códigos:

Introducción

PatternApplier es una herramienta de diseño que permite transformar un modelo UML aplicando uno o varios patrones sobre él. La herramienta está desarrollada como un pluglet para ejecutar en el entorno de modelado de Rational Software Architect de IBM, y permite usar patrones especificados mediante modelos UML en este entorno.

El presente manual muestra al usuario de la herramienta *PatternApplier* cómo usarla en sus modelos utilizando patrones ya definidos, así como la manera de diseñar sus propios patrones o de ampliar la funcionalidad de la herramienta. Se supone que el usuario está familiarizado con RSA y el modelado con UML.

Conceptos fundamentales

- **Modelo:** Es una abstracción de un sistema. El modelo describe los aspectos del sistema que son relevantes para el propósito del modelo, a un nivel de detalle apropiado. En este documento nos referiremos con el término "modelo" a modelos de sistemas técnicos capturados mediante el lenguaje de modelado UML.

- **Proyecto:** Los proyectos a los que se hace referencia en el documento son proyectos de modelado de RSA. Un proyecto de modelado en RSA es un proyecto en el espacio de trabajo de RSA que contiene uno o varios modelos UML y, opcionalmente, uno o varios perfiles UML, así como cualquier otro tipo de recursos auxiliares: ficheros con imágenes, de texto, etc.

- **Patrón:** Es una solución a un problema de diseño. Los patrones mencionados en el documento son soluciones de diseño que capturamos en modelos UML. Así pues, tendremos un modelo de definición del patrón.

- **Rol:** Es un papel o función que cumple algo o alguien en un contexto. En la definición de un patrón, varios elementos (parámetros) realizan distintos roles en el contexto del patrón para resolver el problema. Al aplicar un patrón a un modelo, se especifican qué elementos del patrón dan valor a sus parámetros, es decir, realizan los roles definidos en el patrón.

- **TemplateParameter:** Es un elemento UML que permite definir una plantilla con parámetros (véase [UML2-2009] pág. 628). Se usarán en los modelos de definición de los patrones para definir roles que tendrán que ser jugados por elementos externos al patrón.

- **TemplateBinding:** Es un elemento UML que permite enlazar un modelo con otro modelo que contenga plantillas con parámetros (véase [UML2-2009] pág. 627). A través de este enlace se puede saber elementos que juegan los roles.

1. Instalación de la herramienta

A continuación se explican dos maneras de importar la herramienta *PatternApplier* al explorador de proyectos del RSA para poder trabajar con ella. Podremos descargar las fuentes del repositorio svn de ASLab o importarla desde un RAS Asset.

1.1 Descargar desde el repositorio svn ASLab

En la vista **SVN Repository Exploring**, buscamos el directorio **/root/people/ealarcon/PatternApplier**[1] en el repositorio SVN de ASLab. Pinchamos con el botón secundario del ratón y seleccionamos **Checkout.**

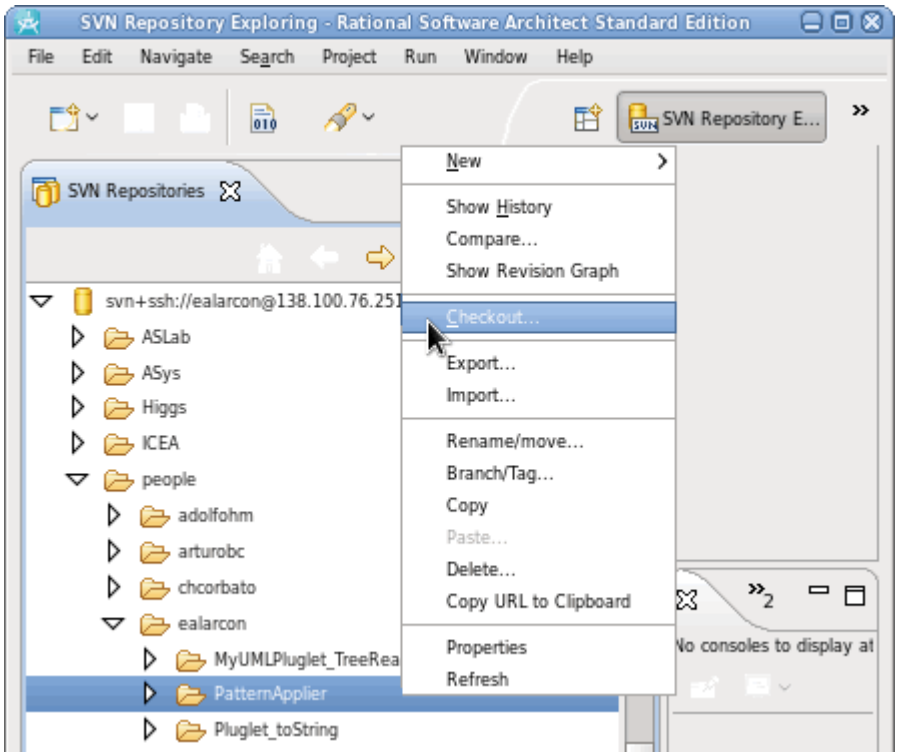


Ilustración 1. Descarga de PatternApplier desde repositorio

En las siguientes ventanas elegimos las opciones que sean de nuestro interés. Será necesario marcar que la salida ha de ser un proyecto en el espacio de trabajo. ***(Check out as a Project in the workspace).***

Al final se realiza la descarga del proyecto al explorador de proyectos.

[1] URL del repositorio: svn+ssh://software.aslab.upm.es/home/svnroot

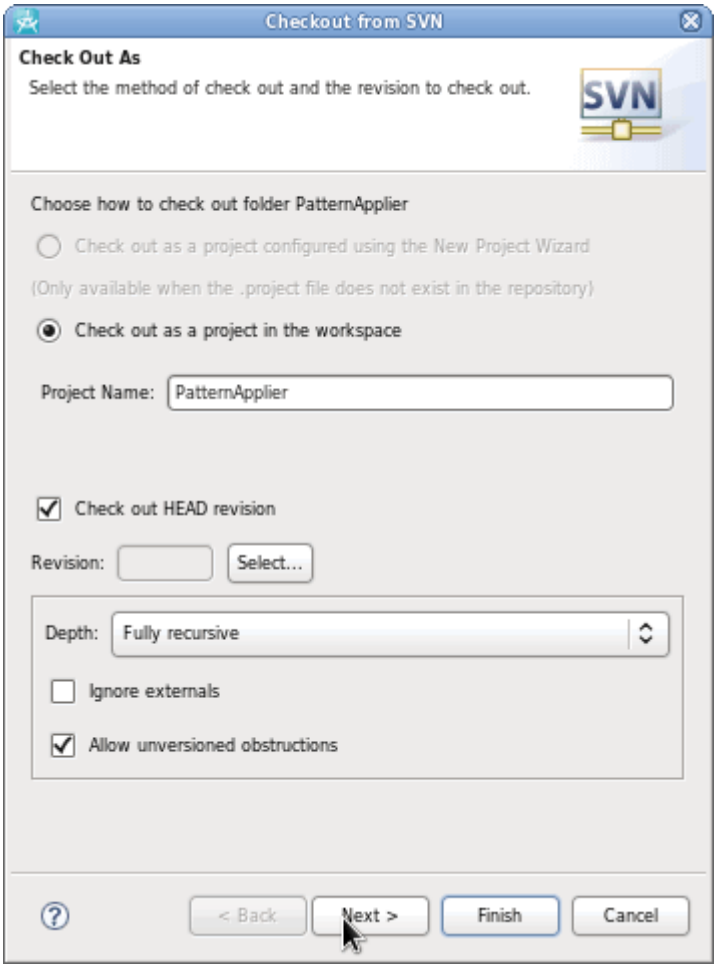


Ilustración 2. Check out (paso 1)

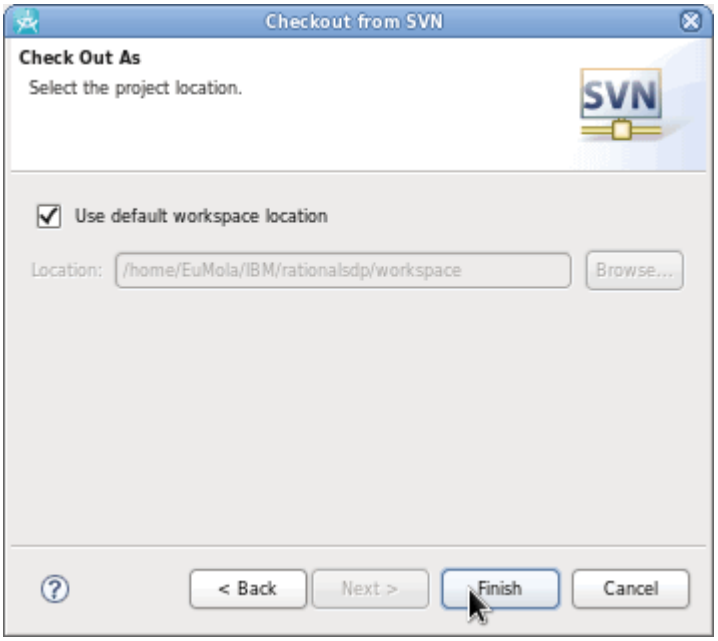


Ilustración 3. Check out (paso 2)

1.1 Importar RAS Asset

Otra opción es importar el RAS Asset en el que está empaquetada la herramienta. En primer lugar debemos descargar en nuestro equipo el fichero que lo contiene de http://aslab.org/software/PatternApplier.ras. Seguidamente tenemos que importar el contenido en nuestro RSA. Para ello, seleccionamos **File/Import…**

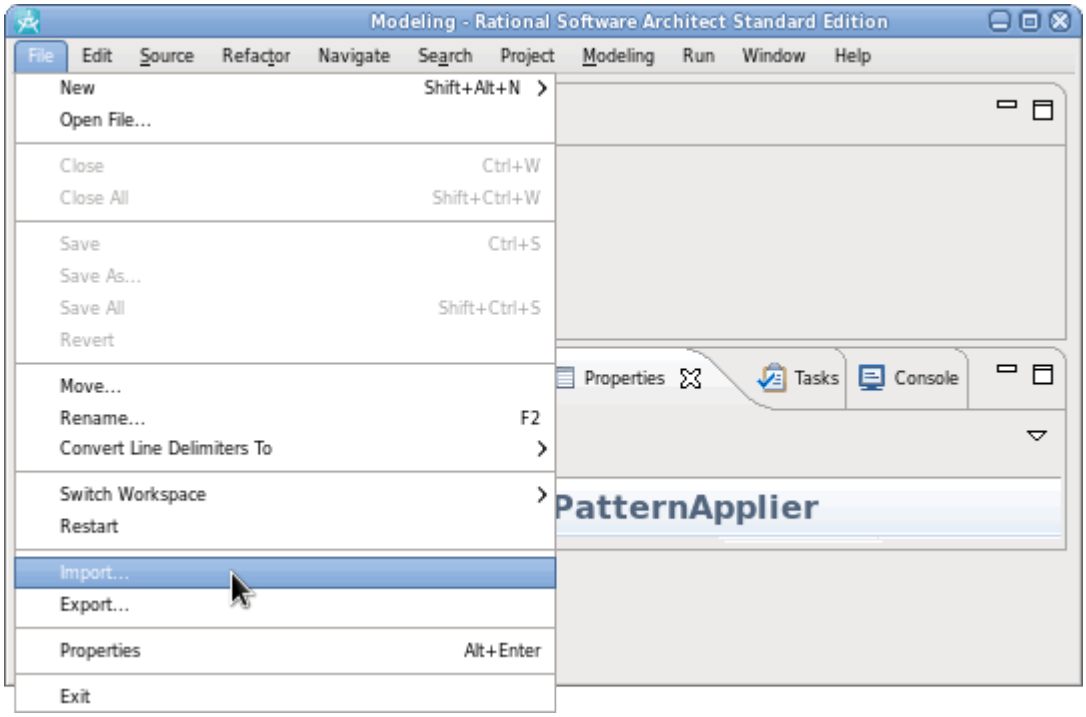


Ilustración 4. Importar

En la ventana que surge, elegimos **RAS Import** y continuamos.

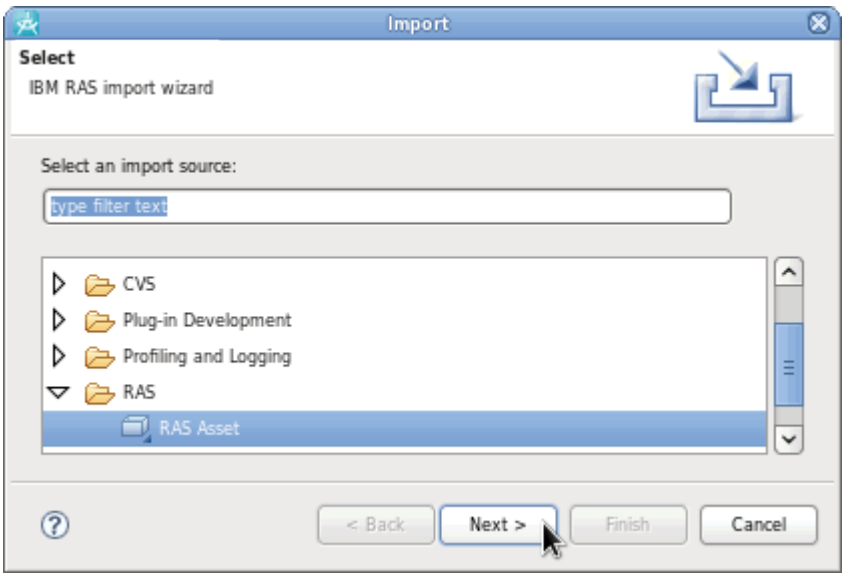


Ilustración 5. Importar RAS Asset

A continuación indicamos la localización del fichero que contiene el RAS Asset, y finalmente aceptamos el destino pinchando **finish**.

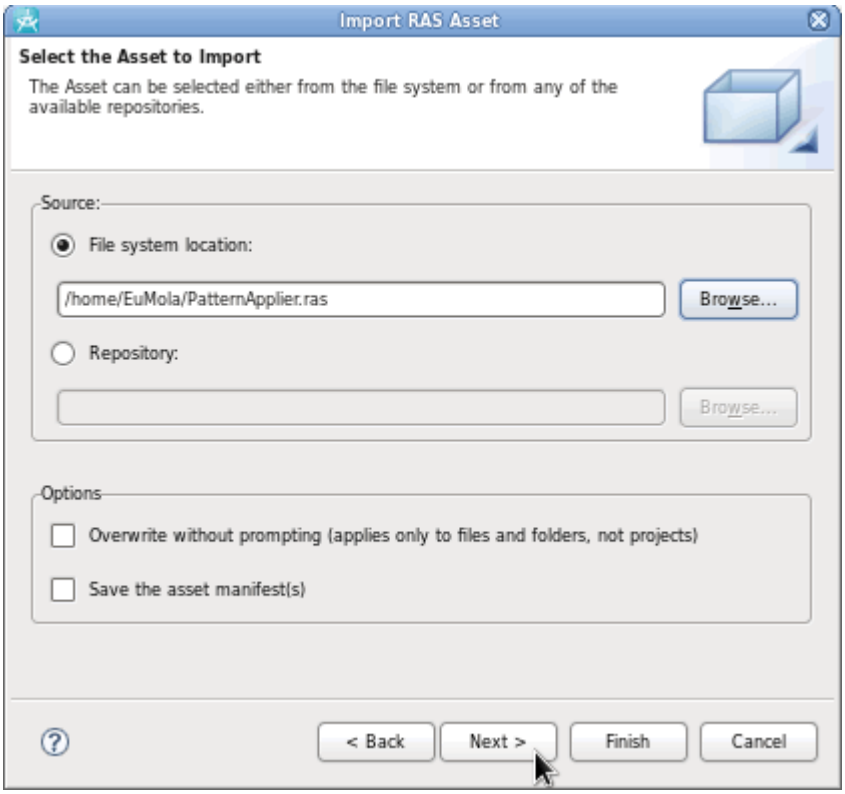


Ilustración 6. Seleccionar localización de PatternApplier.ras

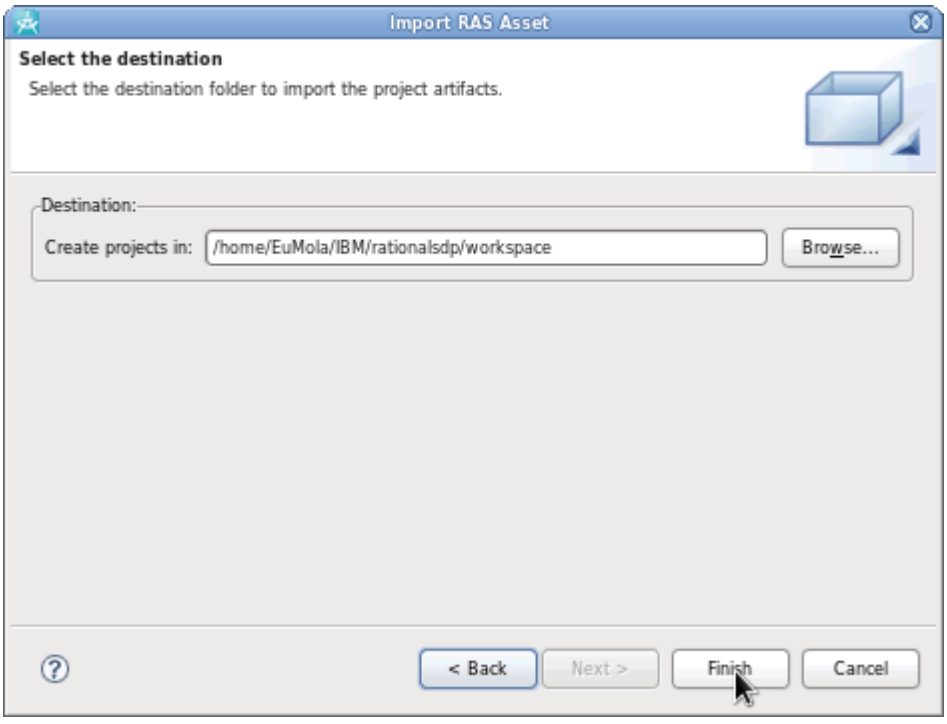


Ilustración 7. Finalizar importación de RAS Asset

2. Aplicación de patrones a modelos

Al usar la herramienta sobre un modelo, se generará automáticamente un nuevo modelo (o transformación del original, dependiendo de la opción elegida) resultante de la aplicación del patrón o patrones sobre éste. A continuación se explican los pasos necesarios para ejecutar el pluglet.

2.1 Apertura del modelo y los patrones

Para aplicar los patrones al modelo es necesario haber abierto tanto el modelo como los patrones (que son modelos UML en RSA también).

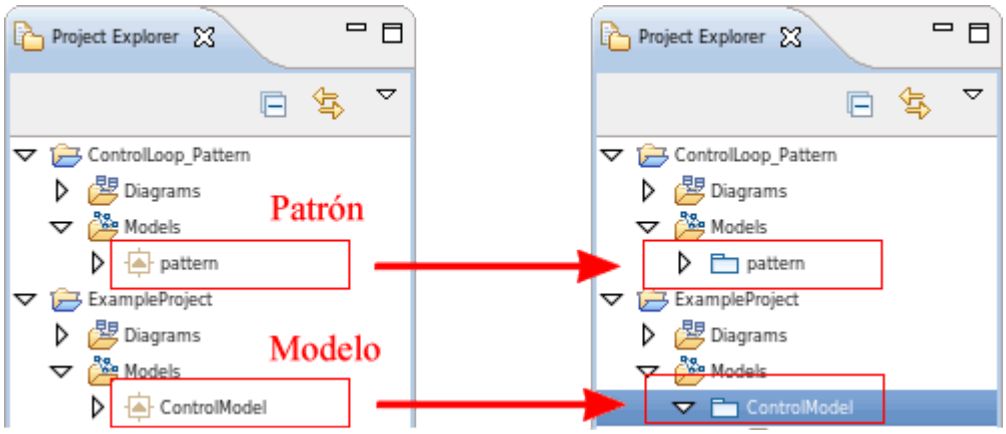


Ilustración 8. Modelos abiertos y cerrados

A la izquierda de la figura se muestran dos proyectos de modelado del espacio de trabajo de RSA: *ControlLoop_Pattern*, que contiene un modelo denominado *pattern*, y *ExampleProject*, que contiene un modelo denominado *ControlModel*.

El modelo del primero es la definición del patrón que se quiere aplicar en el modelo *ControlModel*. A la derecha de la figura se muestra el resultado de abrir dichos modelos (siempre en la perspectiva de modelado de RSA).

2.2 Enlazado del modelo con los patrones

Para enlazar el modelo con un patrón añadimos al modelo un *Template Binding* por cada patrón a aplicar sobre éste. Para ello, en el explorador de proyectos, pinchamos el modelo con el botón secundario y seleccionamos **Add UML/Template Binding.**

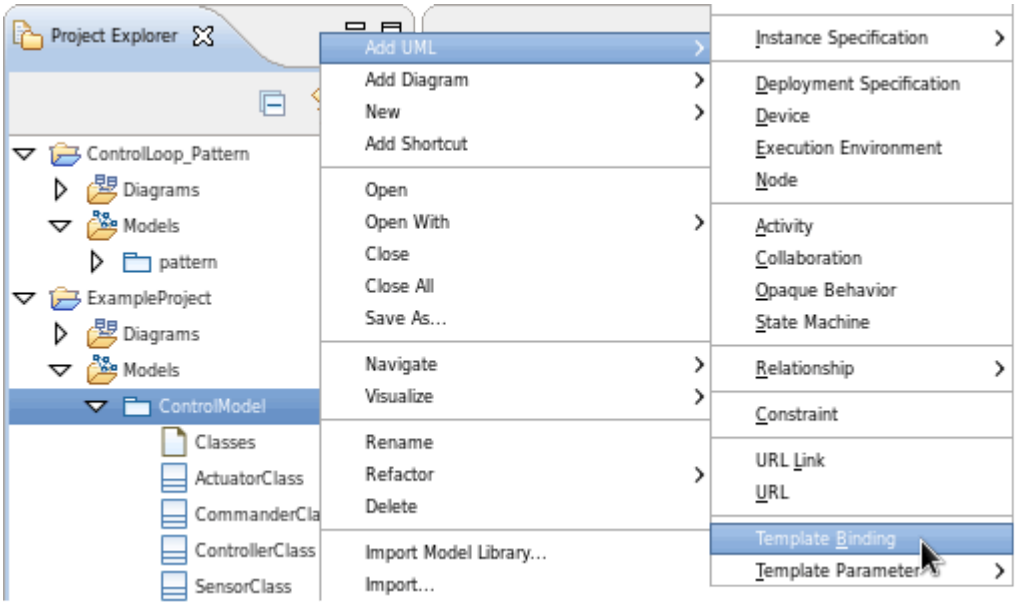


Ilustración 9. Añadir Template Binding

Aparecerá una ventana en la que buscaremos el patrón que queremos aplicar.

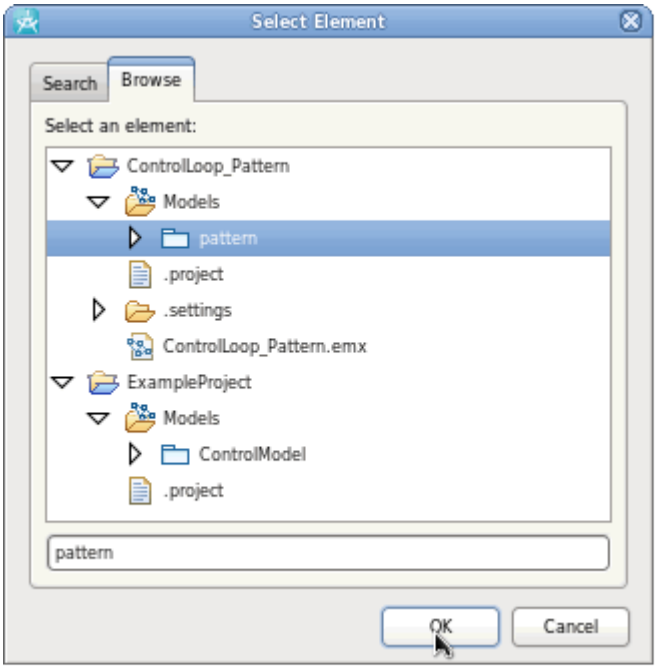


Ilustración 10. Seleccionar patrón con el que enlaza el Template Binding

En las propiedades del *Template Binding* que se ha creado aparecerán los parámetros de la plantilla del patrón (roles). Para ello, pinchamos el elemento *TemplateBinding* que hemos añadido *(Pattern)* y seleccionamos la pestaña de la vista de propiedades. Seleccionamos una de los parámetros ***(controller)*** y pinchamos **Browse.**

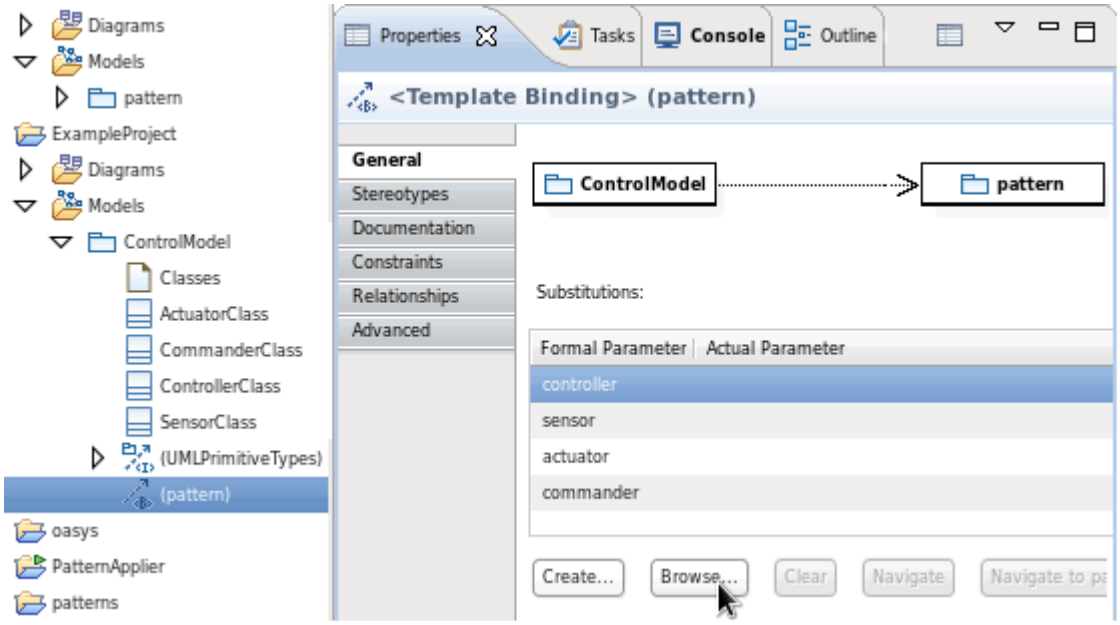


Ilustración 11. Añadir Actual Parameter (substituciones)

Seleccionaremos por cada parámetro tantos elementos del modelo queramos que asuman dicho rol del patrón pudiendo no seleccionar ninguno en el caso de que el patrón contemple dicha posibilidad. Cuando más de un elemento pueda asumir un rol, en la misma ventana, haremos una selección múltiple de los elementos (manteniendo pulsado **Ctrl** mientras seleccionamos los elementos del modelo).

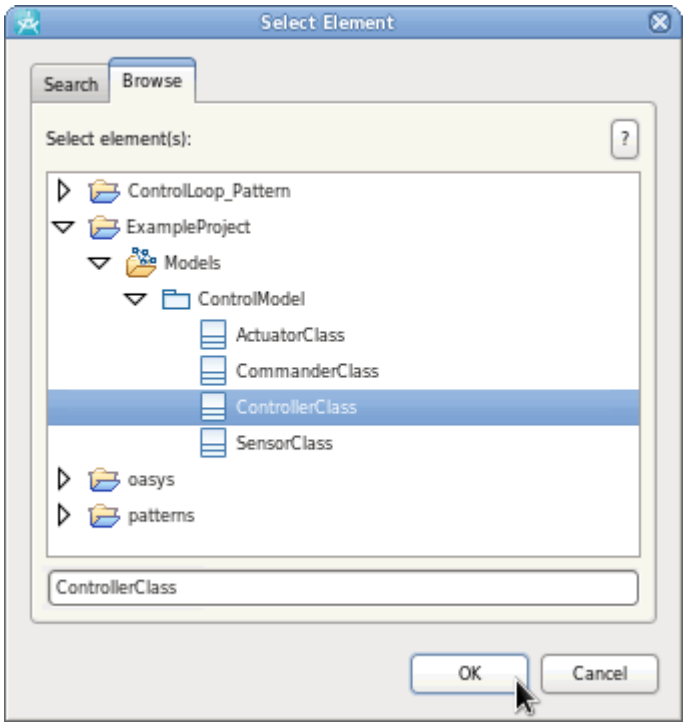


Ilustración 12. Seleccionar Actual Parameter para una plantilla

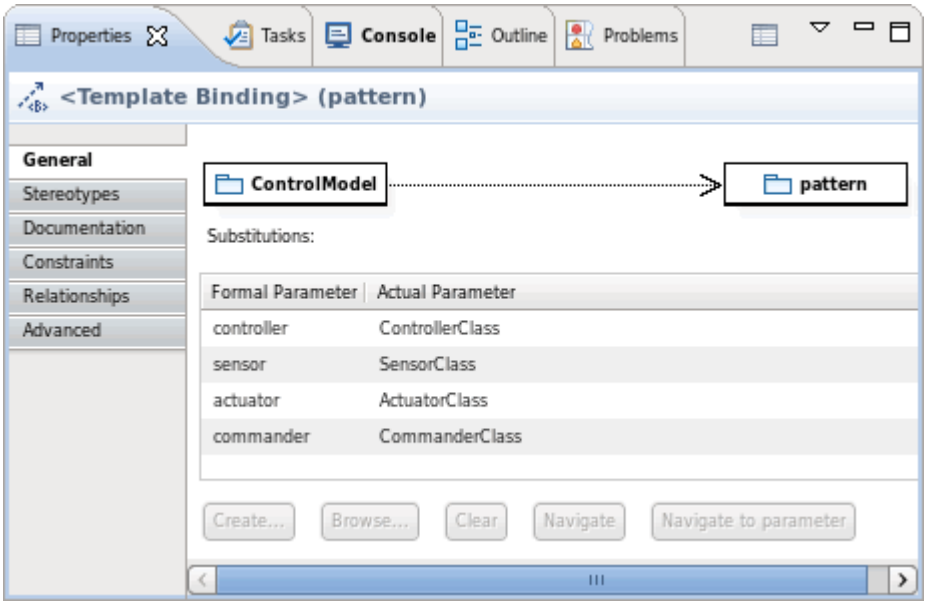


Ilustración 13. Template Binding con todas las substituciones realizadas

2.3 Ejecución del pluglet

Una vez creados todos los enlaces a los distintos patrones y asignados todos los parámetros reales podemos proceder a ejecutar la herramienta. Si es la primera vez que se ejecuta *PatternApplier*, la información relativa a la configuración de su ejecución no estará registrada en nuestro RSA, por lo que habrá que realizar una primera ejecución de la siguiente manera: en el explorador de proyectos desplegamos el proyecto **PatternApplier**, y en el paquete **org.aslab.asys.models.patterns.tools** pinchamos con el secundario sobre **PatternApplier.java** y seleccionamos **Run As/Pluglet**[2]**.**

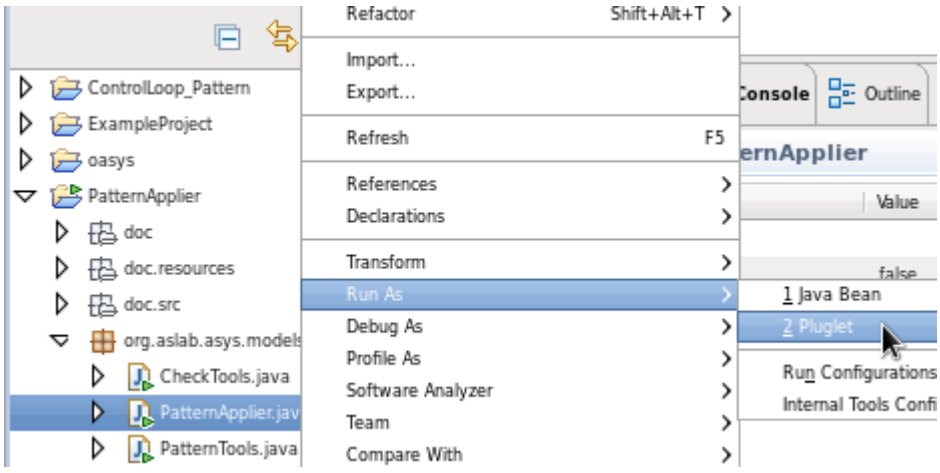


Ilustración 14. Ejecutar PatternApplier como Pluglet por primera vez

[2] Si esta opción no aparece puede que debas habilitar las capacidades de desarrollo de plugins en las preferencias del RSA: Help->Preferences->Capabilities

La configuración de ejecución quedará registrada y la aplicación podrá ser lanzada de manera que se identifique como parámetro de entrada del pluglet el modelo sobre el que se aplica el patrón. Para ello, en el explorador de proyectos, seleccionamos el modelo. A continuación (sin marcar ningún otro elemento en cualquier parte del entorno del RSA) seleccionamos en la barra principal **Run/Internal Tools/PatternApplier.**

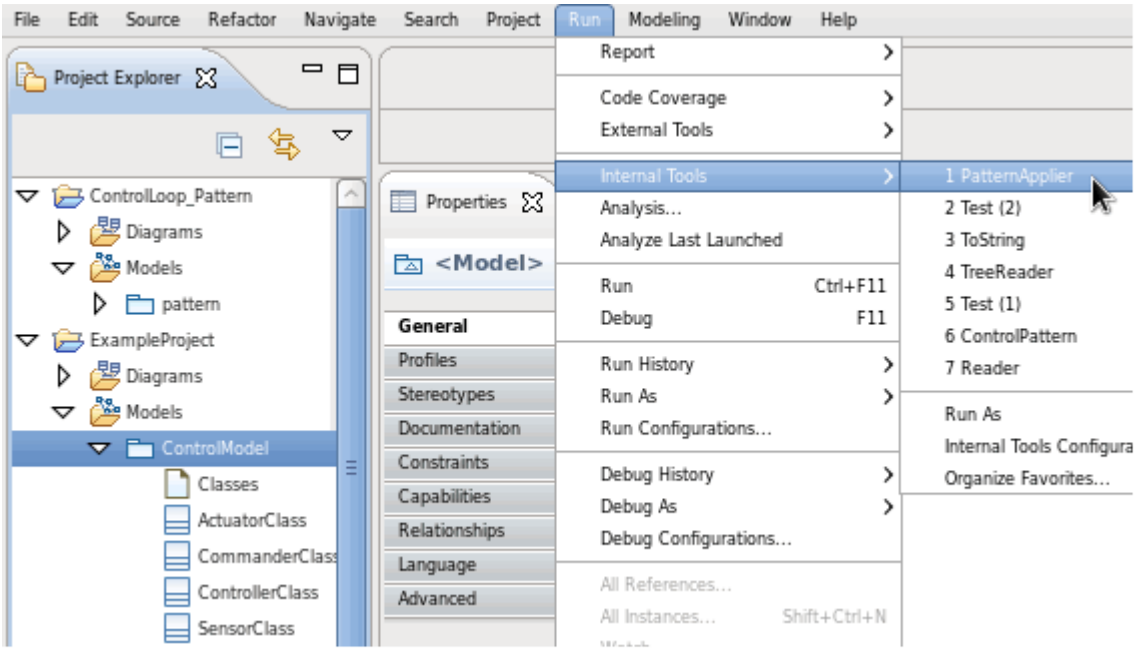


Ilustración 15. Ejecutar PatternApplier

Si el modelo seleccionado es válido aparecerá una ventana para introducir el nombre del nuevo modelo generado (en caso de no querer alterar el original). Finalmente, las transformaciones sobre el modelo se realizarán. Podremos comprobar en la vista de consola si el proceso ha sido correcto.

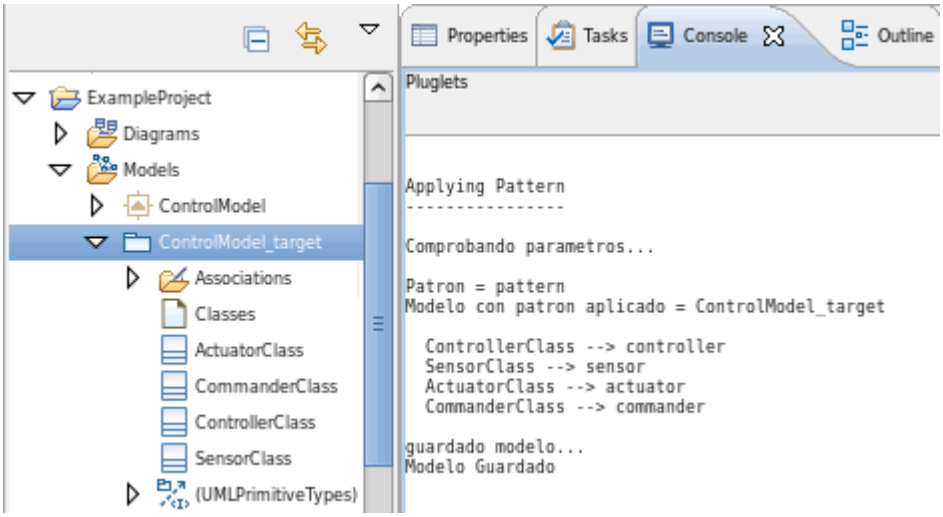


Ilustración 16. Consola y explorador de paquetes tras ejecución de PatternApplier

En el caso de haber elegido un modelo de destino nuevo, en el explorador de proyectos aparecerá un nuevo modelo *(ControlModel_target)* en el proyecto de modelado *(ExampleProyect)* que contenía el modelo de partida *(ControlModel)*. Este nuevo modelo será el resultado de aplicar el patrón al modelo inicial. Si no se eligió un modelo de destino nuevo, el patrón se aplicará sobre el modelo inicial *(ControlModel)*.

3. Diseño de patrones

La herramienta transforma modelos UML de acuerdo con la especificación del patrón, que se realiza siguiendo un lenguaje propio de especificación de patrones que hace uso del UML2 y un perfil del mismo, el *PatternSpec.*

Así, los dos elementos fundamentales del lenguaje de especificación de patrones son dos elementos de UML: los parámetros de plantilla *(template parameters)* y los estereotipos del perfil *PatternSpec*. Estos completan el diseño de los patrones a la par que los dotan de funcionalidad en su interacción con la herramienta.

3.1 Parámetros de plantilla

Al incluir un parámetro en el diseño del patrón estaremos modelando un elemento que posteriormente será substituido por otro, es decir, uno o varios elementos del modelo asumirán el rol del elemento que modela el parámetro. La herramienta sigue así parcialmente la recomendación de la OMG para la especificación de patrones mediante colaboraciones parametrizadas (véase [Booch-2005] Capítulo 29). Con la inclusión de los parámetros necesarios tendremos en el patrón una plantilla con parámetros que se deben asignar a la hora de aplicar el patrón sobre un modelo.

3.1.1 Adición de parámetros a nuestro patrón

Para añadir un parámetro al patrón, pichamos con el secundario el modelo de definición del patrón y seleccionamos **Add UML/Template Parameter/(tipo elemento)[3].**

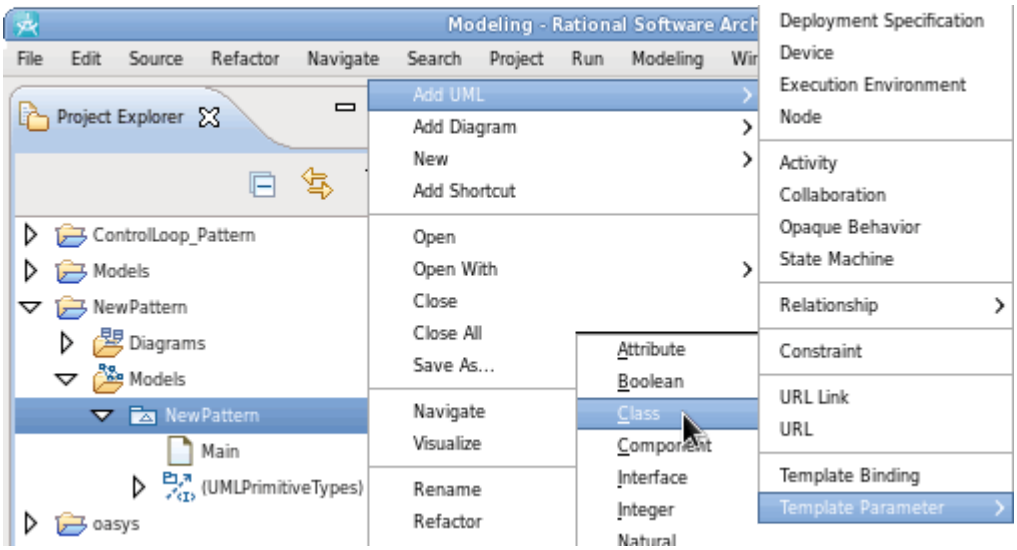


Ilustración 17. Añadir Template Parameter

[3] Ver apartado 3.1.2 Limitaciones

En el modelo de definición del patrón aparecerá un *TemplateSignature* (firma de platillas, visible en el explorador de proyectos. Dentro, aparecerán distintos tipos de plantillas dependiendo del tipo de parámetro que se haya seleccionado.

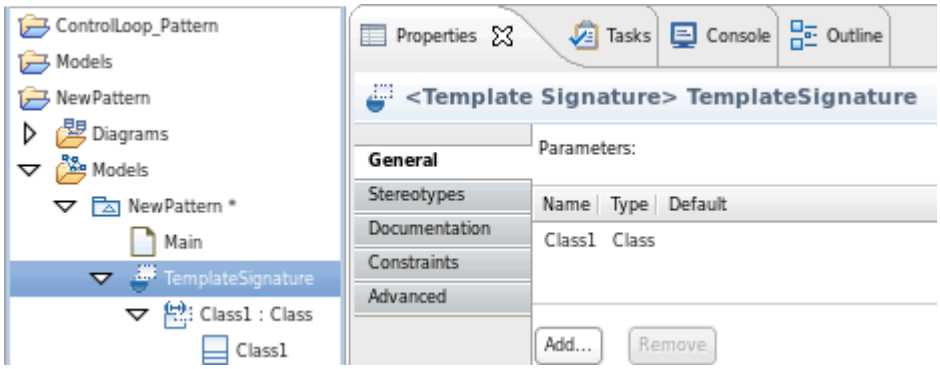


Ilustración 18. Vista de propiedades de un TemplateSignature

Cada plantilla, contiene un elemento (elemento parametrizado) del tipo del *template parameter* que podrá ser modelado en base al rol que representa dentro del patrón. Este rol será asumido por otros elementos al aplicar el patrón sobre un modelo.

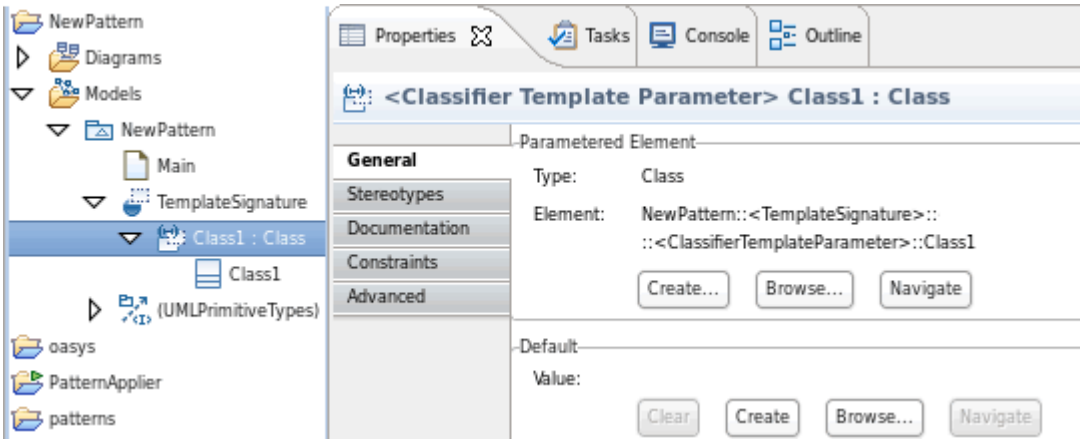


Ilustración 19. Vista de propiedades de un Template Parameter

En el diagrama de clases, tienen la misma apariencia un elemento sin parametrizar (Class2) que uno parametrizado del mismo tipo (Class1).

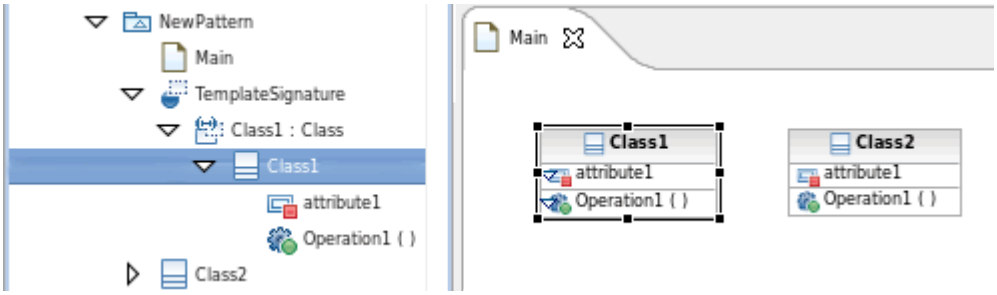


Ilustración 20. Vista de una clase parametrizada frente a otra clase sin parametrizar

3.1.2 Limitaciones de parámetros de plantilla

El uso de parámetros en patrones para ser aplicados con la herramienta *PatternApplier* está limitado a elementos del tipo ***Class*** (clase) e ***Interface*** (interfaz). Los elementos como atributos y operaciones, pertenecientes a dichos elementos deberán ser tratados usando el mecanismo explicado en el siguiente punto. De manera que habrá que especificar si son creados o no en el elemento que juega el rol.

Si se quiere poder utilizar otro tipo de elementos como parámetros será necesario ampliar la funcionalidad de la herramienta.

3.2 Perfil para especificación de transformaciones

El uso de perfiles UML permite la extensión de la sintaxis y la semántica del UML para expresar conceptos específicos de un determinado domino de aplicación. En nuestro caso de diseño de patrones, definir una notación para extender UML como lenguaje formal de especificación de patrones. Es "formal" porque especifica las transformaciones que realizará la herramienta *PatternApplier*.

El perfil *PatternSpec*, contenido en la carpeta de proyecto de la herramienta, ha sido definido para proveer de estas notaciones necesarias para los patrones. Uno de los mecanismos usados en la definición de perfiles son los estereotipos. Estos son unos tipos limitados de metaclases que se usan en combinación con otro tipo de metaclases a las que extienden.

3.2.1 Uso de estereotipos

En el perfil *PatternSpec* hay definidos diferentes estereotipos. En el modelo de definición del patrón, se aplicará un estereotipo a un elemento en función de lo que se quiera que la herramienta haga con ese elemento (por ejemplo, crearlo en el modelo final).

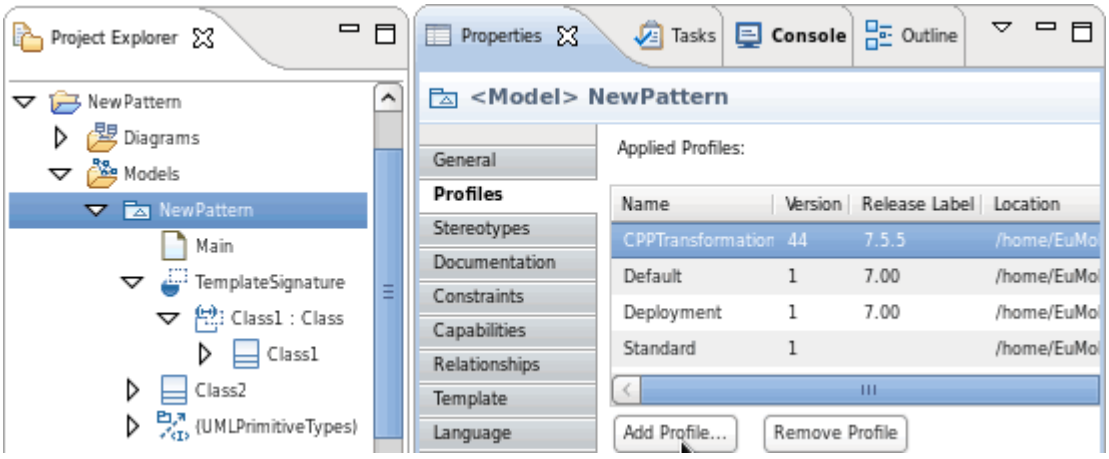


Ilustración 21. Añadir perfil a un modelo

Para poder estereotipar los elementos del patrón es necesario añadir el perfil al modelo de definición del patrón (Ilustración 21). Marcamos el **modelo en el explorador de proyectos**; en la pestaña de **propiedades** seleccionamos **perfiles** y pulsamos **Add Profile…**

Seleccionamos la opción de **perfil en el espacio de trabajo.**

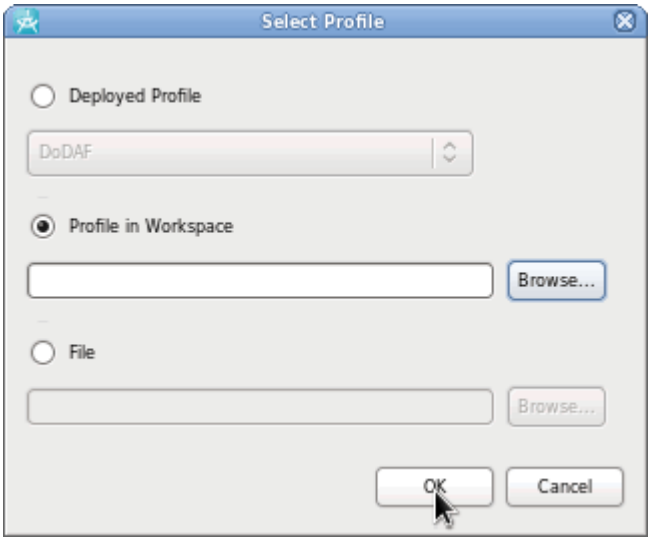


Ilustración 22. Seleccionar origen del perfil a añadir

Elegimos el perfil ***PatternSpec*** en la **carpeta *PatternApplier***.

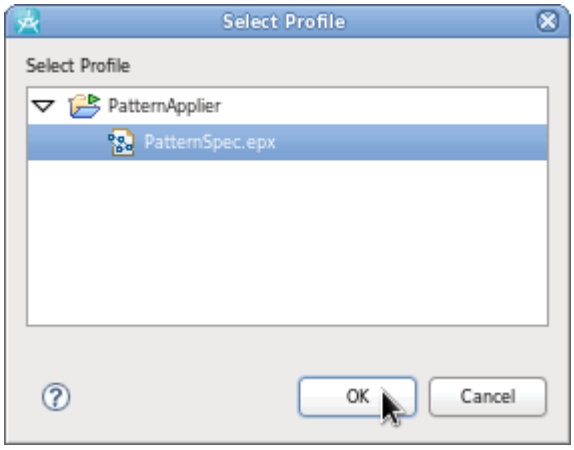


Ilustración 23. Seleccionar el perfil a añadir

Ahora ya podemos estereotipar los elementos que definen el patrón. Para ello, marcamos el **elemento** y en la pestaña de **propiedades**, seleccionamos **estereotipos** y pulsamos **Apply stereotypes…**

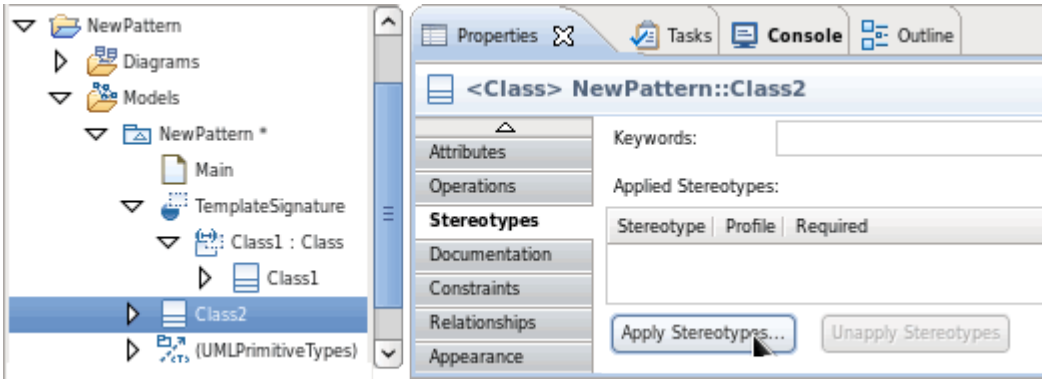


Ilustración 24. Aplicar estereotipos

Aparecerá una ventana con un listado de los estereotipos que se pueden aplicar a ese tipo de elemento. Al lado del nombre del estereotipo aparece el nombre del perfil, por lo que será fácil identificar los estereotipos aportados por el perfil *PatternSpec*.

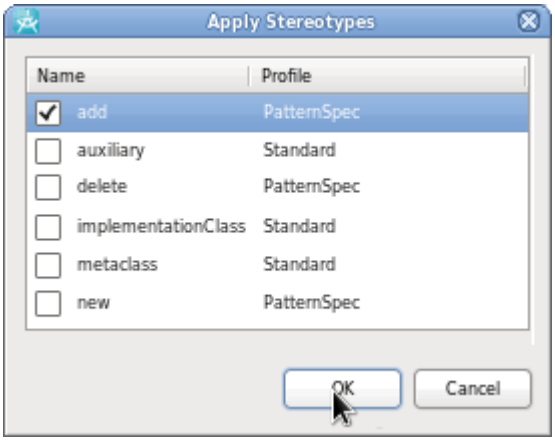


Ilustración 25. Seleccionar estereotipos a aplicar

Tras seleccionar un estereotipo, el elemento se mostrará estereotipado tanto en el explorador de proyectos como en el diagrama. En la siguiente figura se muestran además de una clase parametrizada (*Class1*), una clase y una asociación estereotipadas *(Class2 y Association.*

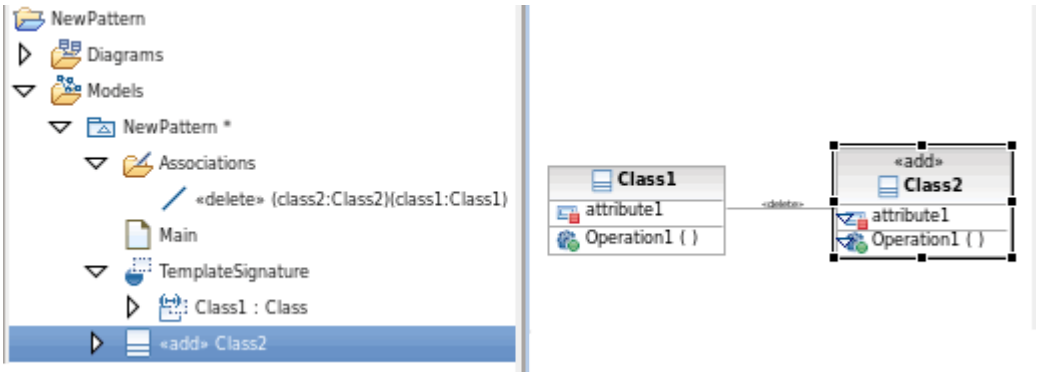


Ilustración 26. Ejemplo de elementos con estereotipos del Perfil PatternSpec aplicados

3.2.2 Estereotipos disponibles

La siguiente tabla muestra los estereotipos definidos en el perfil y la transformación que especifican:

Estereotipo	Especificación
new	El elemento será creado en el modelo de destino sin importar si éste ya existe o no en el modelo fuente.
add	El elemento se añadirá al modelo de destino si éste no existe ya en el modelo fuente.
delete	Si el elemento existe en el modelo fuente, éste será eliminado.
optional	El elemento parametrizado no tiene porqué ser sustituido por un elemento del modelo fuente, cuando es así no se crea en el modelo final.
rename	Un elemento del modelo fuente que substituye a otro en el patrón tomará el nombre de este último.

3.2.3 Limitaciones de los estereotipos

A pesar de que en el diseño del perfil RSA permite aplicar los estereotipos de *PatternSpec* a cualquier elemento, existen limitaciones a los elementos que pueden ser usados en el diseño de los patrones. Para poder usar elementos distintos es necesario ampliar la funcionalidad de la herramienta a través del código.

Las restricciones de cada perfil a los elementos que pueden extenderlos en el diseño de los patrones se muestran en la siguiente tabla:

Estereotipo	Elementos
new	*Class, Interface, Package, Association, Generalization, InterfaceRealization, Attribute (en las clases parametrizadas de las plantillas), Operation (en las clases e interfaces parametrizadas de las plantillas).*
add	*Attribute (en las clases parametrizadas de las plantillas), Operation (en las clases e interfaces parametrizadas de las plantillas).*
delete	*Association, Attribute (en las clases parametrizadas de las plantillas), Operation (en las clases e interfaces parametrizadas de las plantillas).*
optional	*ParameterableElement (elementos parametrizados de las plantillas)*
rename	*ParameterableElement (elementos parametrizados de las plantillas)*

3.3 Renombrar elementos mediante parámetros

En el diseño de los patrones se pueden añadir unas notaciones específicas para indicar que elementos del patrón que serán creados en el modelo final, adoptarán el nombre de elementos del modelo fuente como parámetro en sus propios nombres. La forma de indicar que algún elemento toma el nombre de otro para el suyo propio es insertando una cadena de caracteres que indicará que en su lugar se insertará el nombre de un elemento del modelo fuente.

La cadena tiene la forma **$*reference_element*$.** Los símbolos **$** indican que se inserta un parámetro. ***Reference_element*** es el elemento del patrón que sirve de referencia para tomar el nombre que se inserta en el nombre del elemento a renombrar. El elemento de referencia tiene que ser un elemento parametrizado del patrón; y el nombre se toma del elemento del modelo que juega el rol de dicho elemento.

Ejemplo de caso de uso:

Este caso sería el de un elemento que es creado en el modelo porque está así especificado en el patrón y queremos que el nombre del elemento creado tenga parcial o totalmente el nombre de un elemento de nuestro modelo.

La figura inferior muestra un ejemplo de patrón con un parámetro de plantilla y su correspondiente elemento parametrizado “Interface” y otra clase “$Interface$_wrapper” que será creada y generalizará a la clase que asuma el rol de Interface.

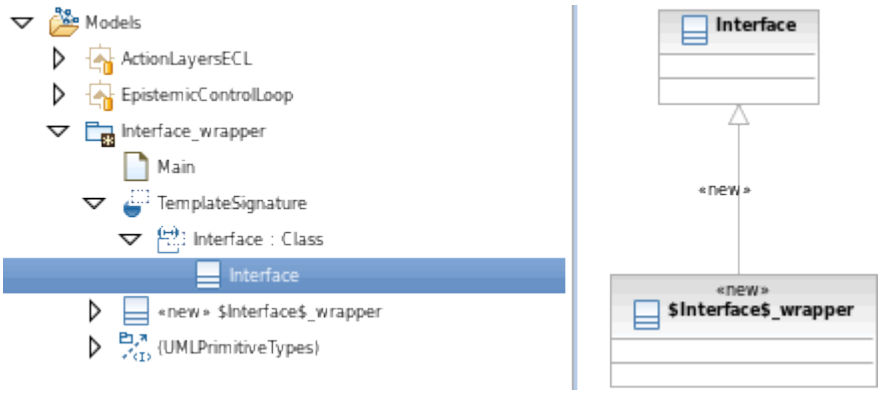


Ilustración 27. Ejemplo de patrón con renombre de elementos a través de parámetros

La subcadena “*_wrapper*” permanecerá intacta (también se podía haber añadido una subcadena al principio), mientras que la parte entre los símbolos “$” (incluidos) será sustituido por el nombre que pasamos como parámetro. La referencia que se da para el nombre es el elemento *Interface*. Este elemento, al ser un elemento parametrizado, aportará como parámetro el nombre del elemento real del modelo que asume el rol de dicho elemento. Es decir, el nombre que se añade a la clase creada será el de la clase que asume el rol de *Interface*. (En el caso de que haya más de un elemento real, se tomará el del primero de ellos).

Aplicando el patrón a un modelo formado por una clase *Class1* que asume el rol de *Interface* el resultado tras la transformación sería el siguiente:

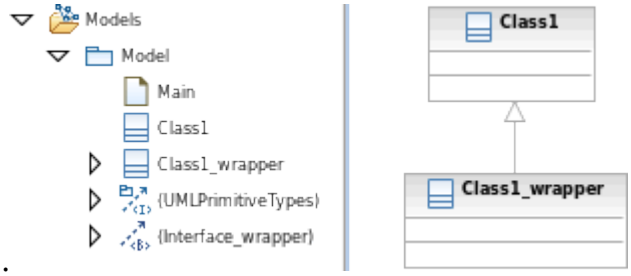


Ilustración 28. Resultado de aplicar patrón con renombre de elementos a través de parámetros

El resultado es que la clase *Class1* asume el rol de *Interface*, y se crea una clase que hereda de ella. Esta clase, toma el nombre de *Class1* para insertarlo en el suyo propio resultando *Class1_wrapper*.

4. Ampliación de la funcionalidad

En los siguientes apartados se explicarán algunos puntos de extensión de la funcionalidad de la herramienta, así como algunas reglas para el correcto funcionamiento de las extensiones que se realicen dentro del código.

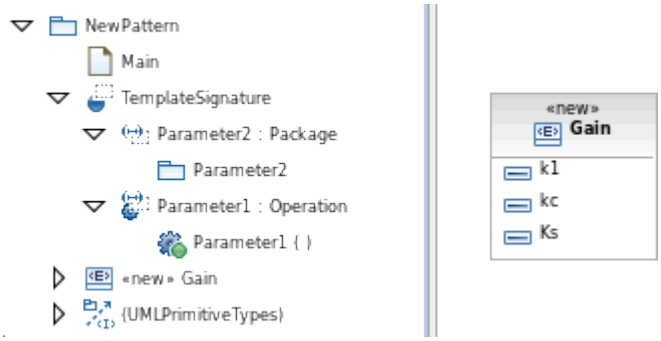


Ilustración 29. Ejemplo de extensión de parámetros de plantilla y elementos estereotipables

La idea es que se puedan realizar pequeñas extensiones en los elementos que pueden ser parametrizados o estereotipados sin tener que conocer en profundidad la estructura del código. En el patrón de la ilustración 29 hay dos elementos de plantilla *Parameter1 (Operation)* y *Parameter2 (Package)*, que en caso de ser substituidos, sólo aportarían un elemento que juega un rol; no serían transformados de modo alguno pues no está contemplado en el código de PatternApplier. También hay un elemento *Gain (Enumeration)* que está estereotipado para ser creado en el modelo sobre el que se aplica el patrón *(New)*. Este tipo de elemento no está soportado por la herramienta, por lo que sería necesario realizar una extensión.

Son casos simples de extensiones que pueden surgir durante el diseño de un patrón. Aún así, se considera que el usuario de este punto posee conocimientos de cierto nivel sobre las librerías que permiten la generación de UML mediante programación y el Eclipse Modeling Framework para manejar los elementos de los modelos UML.

4.1 Extender elementos parametrizables

Como se ha mencionado en el punto 3.1.2, los elementos que se pueden usar como parámetros de plantilla están limitados. Si se quiere poder usar en el diseño alguno distinto a los indicados será necesario añadir la semántica de la transformación que produce ese tipo de elemento en un patrón. Esto se hará en el código de PatternApplier.

Para ello existe la clase *PatternParameters*. Esta clase maneja las substituciones realizadas en el modelo sobre el que se aplican los patrones. Cuando la herramienta es ejecutada, tras reconocer el modelo y los patrones que interactúan, la primera acción de transformación que realiza es resolver los parámetros de plantilla.

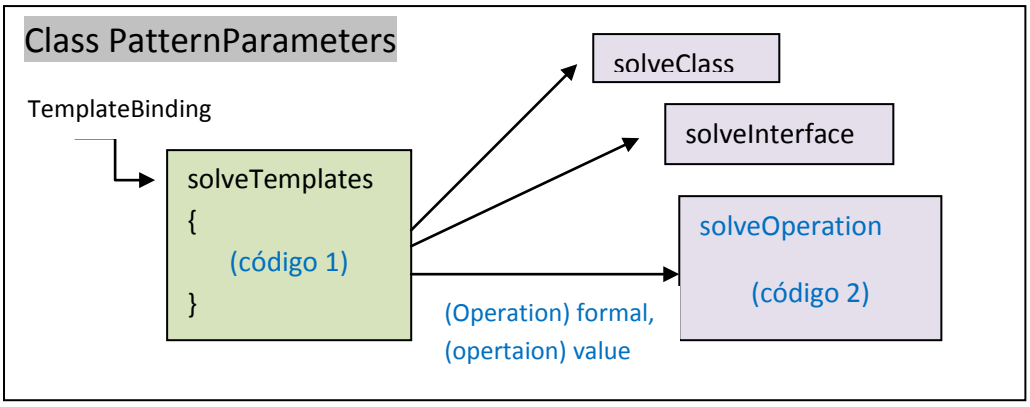


Ilustración 30. Diagrama de funcionamiento de la clase *PatternParameters*

El primer método de la clase es *solveTemplates*. Este método recorre las substituciones realizadas interpretando el tipo de elemento de la plantilla. Si se quiere algún tipo nuevo se tendrá que usar el punto de extensión al final del método (Ilustración 30). Por ejemplo, si tenemos un tipo de elemento *Operation*, la extensión sería:

```
else if (value instanceof Operation) {

      solveOperation((Operation) formal, (Operation) value);

}
```

Código 1. Extensión de tipos de parámentros de plantilla

Hay que indicar el tipo de elemento del que es instancia el elemento de la plantilla, en este caso *Operation*. Así, cuando se dé el caso, los elementos implicados serán tratados con el método especialmente diseñado para operaciones *solveOperation*. A este método se le pasan el elemento *formal*, que es el elemento parametrizado del patrón, y *value*, que es el elemento del modelo que lo substituye. Es importante hacer "casting" del tipo de elemento de la plantilla en el paso de los parámetros *(Operation) formal*, *(Operation) value*.

Después habrá que implementar el método *solveOperation*, que se añadirá en la misma clase *PatternParameters*. El usuario tendrá que completar el método en base a la semántica que se quiera dar a este nuevo tipo en el diseño de patrones. Se requerirá conocimiento del paquete org.eclipse.uml2.uml para manejar los elementos UML del modelo y del patrón. A continuación se muestra la definición del método.

```
protected void solveOperation(Operation formal,Operation actual)
{
      * Código del método
}
```

Código 2. Forma del método solveOperation

Los métodos ya implementados para otros tipos de elementos pueden servir de orientación. Esta es la estructuración básica para los elementos de plantilla añadidos en el diseño del patrón. En los puntos 4.3 y 4.4 se explicarán otras clases de la herramienta para dar soporte durante la transformación de los elementos del modelo.

4.2 Extender elementos estereotipados

Como pasaba con los parámetros de plantilla, los elementos que pueden ser estereotipados durante el diseño de los patrones están limitados también. Por ejemplo, elementos que pueden ser marcados con el estereotipo *new* para que sean creados en el modelo de destino. Será necesario pues ampliar también el código para añadir nuevos elementos a los indicados en el punto 3.2.3.

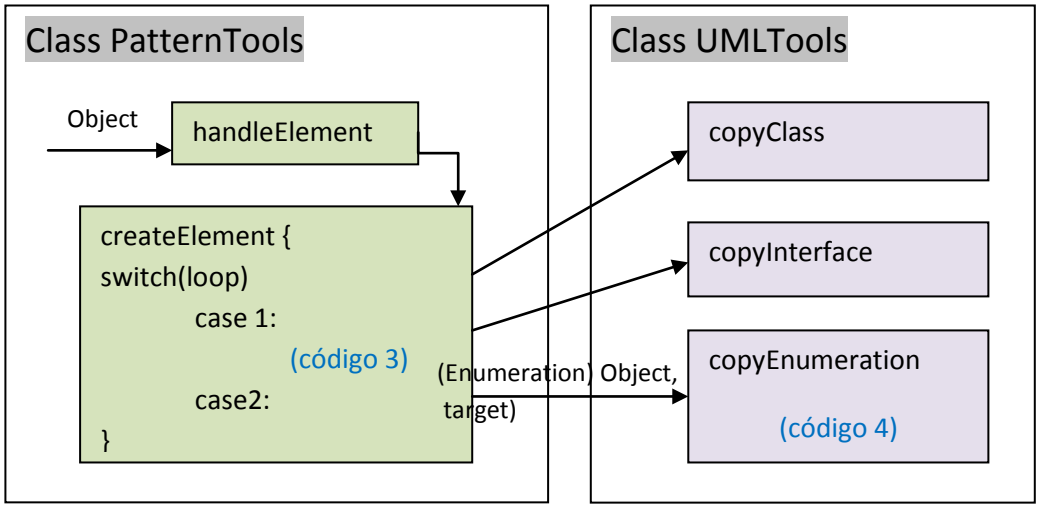


Ilustración 31. Diagrama de creación de elementos nuevos en el modelo de destino

Estas modificaciones tendrán comienzo en la clase *PatternTools*. Tras realizar las tareas oportunas con los parámetros de plantilla, la aplicación pasa a trabajar con los elementos estereotipados del patrón. La herramienta analizará dos veces los elementos estereotipados y realizará las transformaciones necesarias. Se hará dos veces porque se hará distinción entre dos tipos de elementos según sean elementos individuales (clases, interfaces, paquetes,…) o sean elementos de tipo asociación. Esta distinción se realiza ya que en ocasiones, para poder crear elementos del segundo grupo, será necesario haber creado ciertos elementos del primero. Por ejemplo, para crear una generalización de una clase C1 a otra clase C2 que se crea en el modelo porque está así indicado en el patrón, será necesario que se cree antes la superclase C2, para así poder crear la asociación de generalización de C1 a C2.

En el método *createElement* habrá que hacer una extensión para dar soporte a un nuevo tipo de elemento. Suponiendo que queramos poder añadir un tipo de elemento como *Enumeration*, habría que añadir un caso en el primer bucle *loop* (ilustración 31), pues se consideraría un elemento individual.

```
if (object instanceof Enumeration) {
      UMLTools.copyEnumeration((Enumeration) object, target);
      break;
}
```

Código 3. Extensión de tipos de elementos que se pueden crear en el modelo de destino

Con esta extensión se podría crear un elemento *Enumeration* en el modelo sobre el que se aplica el patrón. Si se quisiera que la funcionalidad fuese borrar en lugar de crear, habría que habría que implementar un método para chequear la existencia del tipo (punto 3.4) y en tal caso eliminarlo. Se puede tomar como ejemplo para ello, el caso de *Association* que permite eliminar asociaciones.

En el código superior se llama a un método perteneciente a otra clase de la herramienta, *UMLTools*, que se tendrá que implementar también. Esta clase que proporciona métodos para copiar elementos UML se explicará en el punto siguiente. Cabe destacar, que al método se le han de pasar como parámetros el elemento del patrón que se toma como referencia para copiar *(Enumeration) Object*, y el modelo *(target)* que donde se realiza la copia del primero. La forma en que se crean elementos en el modelo dependerá siempre de el propio elemento que queremos crear por lo que será necesario tener los conocimientos necesarios para ello.

4.3 Clase UMLTools

Esta clase de la herramienta está implementada como una librería estática que proporciona métodos para copiar elementos UML. Los métodos están implementados de forma que al pasarles como parámetros un elemento y un paquete o modelo UML se crea una copia del primer elemento es este paquete o modelo. Se considerará copia hasta el nivel que el usuario quiera que compartan unas mismas características.

En el ejemplo propuesto en el punto anterior, había que implementar un método llamado *copyEnumeration* dentro de la clase *UMLTools.* La forma que debería tener es la siguiente:

```
static Enumeration copyEnumeration(Enumeration enum,
                                   Package target ) {
      newenum = target.createOwnedEnumeration(null);
                  .
                  .
                  .
      * Más código del método
}
```

Código 4. Forma del método copyEnumeration en la clase UMLTools

El usuario tendrá que completar el método copiando las características que crea oportunas del elemento *enum* al elemento *newenum.* Si fuese necesario, podrían usarse otros métodos de *UMLTools* para completar el método; este sería el caso de que el elemento que se copia contenga otros tipos de elementos UML.

4.4 Clase CheckTools

Esta clase proporciona métodos para realizar comprobaciones o localizar elementos dentro del modelo sobre el que se le aplica el patrón.

Uno de los métodos que puede ser necesario a la hora de ampliar la funcionalidad de la herramienta es el método *searchNamedElement.* Este método localiza en el modelo sobre el que se aplican los patrones un elemento a partir de otro elemento o de un nombre. Por ejemplo, si necesitamos buscar las clases que relaciona en el modelo una asociación, tomaremos las clases del patrón que están relacionadas con la asociación en el patrón y usando *searchNamedElement* se asignará la clase equivalente para el modelo. Esta clase equivalente puede presentar 3 casos: primero, que la clase haya sido copiada del patrón al modelo y entonces existiría una con el mismo nombre en el modelo; segundo, que la clase que se toma como referencia para la búsqueda sea un elemento parametrizado, por lo que el método buscaría en el modelo sobre el que se aplica el patrón el elemento que juega el rol del elemento parametrizado y ese sería el que asignaría; y finalmente, si no se da ninguno de casos anteriores, se devolvería como resultado de la búsqueda el propio elemento leído en el patrón. Esto significaría que es un elemento perteneciente a otro modelo.

En esta clase se pueden crear también métodos para comprobar si ciertos elementos existen dentro del modelo sobre el que se aplica el patrón. Esta igualdad está sujeta siempre a las características que se consideren de interés. Es el caso de los métodos *searchAssociation, searchOperation, y searchProperty.* En el caso de *searchAssociation,* la asociación se buscará dentro un modelo dado. En los otros dos métodos, el elemento se buscará en una lista de elementos del mismo tipo. La forma de búsqueda estará sujeta a la finalidad de uso del método.

Siguiendo con el ejemplo del elemento *Enumeration,* se presenta la forma que tendría el método *searchEnumeration.* Se puede tener un elemento de tipo *Enumeration* en el patrón y querer buscar un elemento del mismo tipo en el modelo, que se considere igual hasta cierto nivel de detalle. Será necesario pasar como parámetros al método, el elemento *Enumeration* del patrón cuya copia se está buscará en el modelo, así como el modelo o el paquete del modelo donde se realizará la búsqueda. Se analizarán todos los elementos del modelo; y si alguno cumple todas las condiciones para considerarlo igual al elemento de referencia *Enumeration,* se devolverá este elemento encontrado. Si tras analizar todos los elementos del modelo no se ha encontrado ningún elemento que satisfaga todas las condiciones, se devolverá *null*.

```
static Enumeration searchEnumeration(Enumeration enum,
                                     Package target) {

      for (Iterator<EObject> iter = target.eAllContents();
            iter.hasNext();) {


            EObject object = iter.next();
            if (* Comprobaciones de igualdad) {
                        .
                        .
                        .
                  return testenumeration;
            }
      }
      return null;
}
```

Código 5. Forma del método searchEnumeration en la clase CheckTools

Anexo 2. Código de la herramienta

Class PatternApplier

```
public class PatternApplier extends PatternTools {

        public void plugletmain(String[] args) {


       out("\n\nApplying Pattern ");
       out("----------------");

        String undoLabel = "PatternApplier";
        TransactionalEditingDomain editDomain = UMLModeler.getEditingDomain();

        editDomain.getCommandStack().execute(new RecordingCommand(editDomain, undoLabel) {


        protected void doExecute() {



        List<EObject> selectedElements = UMLModeler.getUMLUIHelper().getSelectedElements();
        PlugletParameters parameters = new PlugletParameters(selectedElements);

        if (parameters.correctparameters) {

                refactorTarget(parameters.target);


                for (Iterator<PatternBinding> sourceiter = parameters.source.iterator()
                                        ;sourceiter.hasNext();){

                        PatternBinding pattbind = sourceiter.next();

                        out("\nPatron = " + pattbind.pattern.getName());
                        out("Modelo con patron aplicado = " + parameters.target.getName()+"\n");


                        organizePackageImports(pattbind.pattern.getPackageImports(),
                                                                        parameters.target);



                        if (!CheckTools.checkSubstitutions(pattbind.binding)) {
                                out.println("\n AVISO: Faltan parametros necesarios por asignar en
                                                                                el patron\n");
                        }

                        solveTemplates(pattbind.binding);



                        TreeIterator<EObject> tree = pattbind.pattern.eAllContents();

                        for (Iterator<EObject> iter = tree;iter.hasNext();) {

                                handleElement(iter.next(), parameters.target, pattbind.pattern);
                        }


                        tree = pattbind.pattern.eAllContents();
                        loop++;

                        for (Iterator<EObject> iter = tree;iter.hasNext();) {

                                handleElement(iter.next(), parameters.target, pattbind.pattern);
                        }


                        tree = parameters.target.eAllContents();

                        for (Iterator<EObject> iter = tree;iter.hasNext();) {

                                renameElements(iter.next(), parameters.target, pattbind.pattern);

                        }


                        tree = parameters.target.eAllContents();

                        for (Iterator<EObject> iter = tree;iter.hasNext();) {
```

```
                                renameProperties(iter.next(), parameters.target);

                        }

                        loop--;
                }


                out("\nguardado modelo...");
                save(parameters.target);

                }
                else {
                        out("Modelo incorrecto");
                }


        }

});
}
}
```

Class PatternParameters

```
public class PatternParameters extends Pluglet{

        protected void solveTemplates(TemplateBinding binding) {


                ParameterableElement value = null;
                List<TemplateParameterSubstitution> list = binding.getParameterSubstitutions();

                for (Iterator<TemplateParameterSubstitution> iter = list.iterator();
                                                                        iter.hasNext();;) {

                        TemplateParameterSubstitution subs = iter.next();
                        ParameterableElement formal = subs.getFormal().getParameteredElement();
                        List<ParameterableElement> actuals = subs.getActuals();

                        for (Iterator<ParameterableElement> iter1 = actuals.iterator();
                                                                        iter1.hasNext();;) {

                                value = iter1.next();

                                if ((value instanceof NamedElement)
                                                &&(CheckTools.getStereotype(formal)>4)) {
                                        NamedElement namedvalue = (NamedElement) value;
                                        namedvalue.setName(((NamedElement) formal).getName());
                                }


                                if (value instanceof Class) {

                                        solveClass((Class) formal, (Class) value);

                                }


                                else if (value instanceof Interface) {

                                        solveInterface((Interface) formal, (Interface) value);

                                }

                                /**else if (    ) {

                                }**/

                                // ampliar para extender a mas elementos parametrizados

                                else {

                                        out.println("\n AVISO: Elemento parametrizado no soportado
                                                                                por la herramienta\n");
                                }
```

```
                }
        }
}


protected void solveClass(Class formal, Class actual) {

        out.print("  " + actual.getName() + " --> " + formal.getName() + "\n");

        actual.setName(actual.getName());
        actual.setIsAbstract(formal.isAbstract());

        List<Property> properties = actual.getOwnedAttributes();

        for (Iterator<Property> iter = formal.getOwnedAttributes().iterator();
                                                                iter.hasNext();) {

                Property attribute = iter.next();
                if (attribute.getAssociation()==null){

                        switch(CheckTools.getStereotype((EObject) attribute)) {

                        case 1: //new
                                Property newattribute =
                                                actual.createOwnedAttribute(null,null);
                                UMLTools.copyAttribute(attribute, newattribute);
                                break;

                        case 2: //add
                                if (CheckTools.searchProperty(attribute,
                                                                properties)==null){
                                        Property addattribute =
                                                actual.createOwnedAttribute(null,null);
                                        UMLTools.copyAttribute(attribute, addattribute);
                                }
                                break;

                        case 3: //delete
                                if (CheckTools.searchProperty(attribute,
                                                                properties)!=null){
                                        Property propout =
                                        CheckTools.searchProperty(attribute, properties);
                                        propout.destroy();
                                }
                                break;
                        }
                }
        }

        List<Operation> operations = actual.getOwnedOperations();

        for (Iterator<Operation> iter = formal.getOwnedOperations().iterator();
                                                                iter.hasNext();) {

                Operation  operation = iter.next();


                switch(CheckTools.getStereotype((EObject) operation)) {

                case 1: //new
                        Operation newoperation =
                                        actual.createOwnedOperation(null,null,null);
                        UMLTools.copyOperation(operation, newoperation);
                        break;

                case 2: //add
                        if (CheckTools.searchOperation(operation, operations)==null){
                                Operation addoperation =
                                        actual.createOwnedOperation(null,null,null);
                                UMLTools.copyOperation(operation, addoperation);
                        }
                        break;

                case 3: //delete
                        if (CheckTools.searchOperation(operation, operations)!=null){
                                Operation operout = CheckTools.searchOperation(operation,
                                                                        operations);
                                operout.destroy();
                        }
                        break;
                }
        }

        actual.setVisibility(formal.getVisibility());
        actual.setIsActive(formal.isActive());
        actual.setIsLeaf(formal.isLeaf());

}
```

```
protected void solveInterface(Interface formal, Interface actual) {

        out.print("  " + actual.getName() + " --> " + formal.getName() + "\n");

        actual.setName(actual.getName());

        List<Operation> operations = actual.getOwnedOperations();

        for (Iterator<Operation> iter = formal.getOwnedOperations().iterator();
                                                                iter.hasNext();;) {

                Operation  operation = iter.next();


                switch(CheckTools.getStereotype((EObject) operation)) {

                case 1: //new
                        Operation newoperation =
                                        actual.createOwnedOperation(null,null,null);
                        UMLTools.copyOperation(operation, newoperation);
                        break;

                case 2: //add
                        if (CheckTools.searchOperation(operation, operations)==null){
                                Operation addoperation =
                                        actual.createOwnedOperation(null,null,null);
                                UMLTools.copyOperation(operation, addoperation);
                        }
                        break;

                case 3: //delete
                        if (CheckTools.searchOperation(operation, operations)!=null){
                                Operation operout = CheckTools.searchOperation(operation,
                                                                        operations);
                                operout.destroy();
                        }
                        break;
                }
        }

        actual.setVisibility(formal.getVisibility());
        actual.setIsAbstract(formal.isAbstract());
        actual.setIsLeaf(formal.isLeaf());

}


@SuppressWarnings("unchecked")
protected void renameElements (EObject object, Package target, Package pattern) {

        /* Debe ser un elemento con nombre que no sea una propiedad */

        if ((object instanceof NamedElement)&&(!(object instanceof Property))) {

                NamedElement elementrenamed = (NamedElement) object;
                NamedElement patternreference = null;

                String originalstring = elementrenamed.getName();
                String begin, referencestring,  end;
                String main = "";

                int ind1, ind2;
                int length = 0;


                /* Buscamos el simbolo "$" que nos indica si hay que renombrar el elemento

                try {
                        ind1 = originalstring.indexOf("$");
                        length =originalstring.length();
                } catch (NullPointerException e) {
                        ind1 = -1;
                }

                if (ind1 != -1){ //condicion final

                        /* troceamos el originalstring para tomar las subcadenas de los
                                                                                        extremos
                         * y la cadena de la referencia en el patron */

                        ind2 = originalstring.lastIndexOf("$");
                        begin = originalstring.substring(0, ind1);
                        referencestring = originalstring.substring(ind1+1, ind2);
                        end = originalstring.substring(ind2+1, length);
```

```
                                /* mediante la cadena "referencestring" obtenemos el elemento
                                                                        referencia
                                 * (elemento parametrizado) del patron*/


                                patternreference = CheckTools.searchNamedElement(referencestring,
                                                                        pattern);

                                        /* una vez hemos localizado el elemento referencia en el
                                                                        patron, tomamos
                                         *  el nombre del elemento del modelo que substituye a
                                                                dicho elemento de referencia*/

                                try {
                                        List <NamedElement> referenceactuals =
                                CheckTools.searchBinding((Element) patternreference, target);
                                        if (referenceactuals!=null) {
                                                main = referenceactuals.get(0).getName();
                                        }

                                } catch (NullPointerException e) {

                                        out.println("\n ERROR: Imposible renombrar elemento.");
                                        out.println("        Referecia no valida\n");

                                }
                                elementrenamed.setName(begin+main+end);

                        }
                }
        }
}
```

Class PatternTools

```
class PatternTools extends  PatternParameters {


        protected boolean delete;
        protected int loop = 1;

        protected void out(String output) {
                        out.println(output);
        }

        protected static void err(String error) {
                System.err.println(error);
        }


        protected class PatternBinding {

                protected Package pattern;
                protected TemplateBinding binding;

                PatternBinding(Package package_, TemplateBinding binding_) {
                        this.pattern = package_;
                        this.binding = binding_;
                }

        }


        protected class PlugletParameters{

                protected List<PatternBinding> source = new ArrayList<PatternBinding>();
                protected Package target;
                protected boolean correctparameters = false;

                /*recibe de entrada los elementos seleccionadas que debe ser solo uno*/

                public PlugletParameters(List<EObject> list) {

                        if (list.size()==1){

                                if (list.get(0) instanceof Package){  /*el elemento seleccionado
                                                                        debe ser un
                                                                paquete o un modelo*/
```

```
                                target = (Package) list.get(0);

                                out("\nComprobando parametros...");

                                TreeIterator<EObject> elements = target.eAllContents();
                                for (Iterator<EObject> iter = elements;iter.hasNext();) {
                                        EObject element = iter.next();

                                        /* buscamos los enlaces en nuestro modelo para
                                                                comprobar que son
                                         * correctos y obtener el paquete del patron */

                                        if (element instanceof TemplateBinding) {

                                                TemplateBinding bind = (TemplateBinding)
                                                                                        element;
                                                Package patt = (Package)
                                                        (bind.getSignature()).getOwner();
                                                PatternBinding pattbind = new
                                                        PatternBinding(patt, bind);
                                                source.add(pattbind);
                                                correctparameters = true;

                                        }
                                }
                        }
                }
        }
}


protected void refactorTarget(Package target){

        String path = target.eResource().getURI().path();
        String ext = target.eResource().getURI().fileExtension();

        path = path.substring(1,path.length()-1-ext.length());
        int index = path.lastIndexOf("/");
        String initial_name = path.substring(index + 1);
        path = path.substring(0,path.length() - initial_name.length());

        String model_name = target.getName();
        String final_name = prompt("Si desea que los patrones se apliquen sobre una copia
                                                                        del modelo" +
                        ", modifique el nombre del modelo de destino:",model_name);
        URI final_path = URI.createURI("platform:/" + path + final_name + "." + ext);

        if (!final_name.equals(model_name)) {
                save(target, final_path);
                target.setName(final_name);
        }

}


protected void organizePackageImports(List<PackageImport> imports, Package target) {

        for(Iterator<PackageImport> iter = imports.iterator();iter.hasNext();){

                PackageImport packimp = iter.next();
                Package pack = packimp.getImportedPackage();

                if (!CheckTools.isPackageImported(pack, target)) {

                        target.createPackageImport(pack);

                }
        }
}


protected void handleElement(EObject object, Package target, Package pattern) {

        if (CheckTools.validObject(object)){

                List<Stereotype> list = ((Element) object).getAppliedStereotypes();


                for (Iterator<Stereotype> iter = list.iterator();iter.hasNext();){

                        String ster = iter.next().getName();

                        if (ster.equals("add")) {
                                delete = false;
                                createElement(object, CheckTools.assignPackage(object,
                                                                        target, pattern));
                                break;
                        }
```

```
                        if (ster.equals("new")) {
                                delete = false;
                                createElement(object, CheckTools.assignPackage(object,
                                                                target, pattern));
                                break;
                        }
                        if (ster.equals("delete")) {
                                delete = true;
                                createElement(object, CheckTools.assignPackage(object,
                                                                target, pattern));
                                break;
                        }
                }
        }
}


protected void createElement(EObject object, Package target) {


        switch(loop){
        case 1:

                if (object instanceof Class) {
                        UMLTools.copyClass((Class) object, target);
                        break;
                }

                if (object instanceof Interface){
                        UMLTools.copyInterface((Interface) object, target);
                        break;
                }

                if (object instanceof Package){
                        UMLTools.createPackage((Package) object, target);
                        break;
                }
                break;

        case 2:

                if (object instanceof Association) {

                        TypesLists types = new TypesLists((Association) object, target);

                        if ((types.types1!=null)&&(types.types2!=null)) {

                                for (Iterator<Type> iter1 =
                                                types.types1.iterator();iter1.hasNext();){
                                        Type type1 = iter1.next();

                                        for (Iterator<Type> iter2 =
                                                types.types2.iterator();iter2.hasNext();){
                                                Type type2 = iter2.next();

                                                Association existasso =
                                                        CheckTools.searchAssociation(
                                                                (Association)
                                                        object,type1,type2,target);

                                                if (delete) {
                                                        if (existasso != null) {
                                                                existasso.destroy();
                                                        }
                                                } else {
                                                        if (existasso == null) {

                        UMLTools.copyAssociation((Association) object, type1, type2);
                                                        }
                                                }
                                        }
                                }
                        }

                        break;
                }

                if (object instanceof Generalization){
                        CreateRelations(object, target);
                        break;
                }

                if (object instanceof InterfaceRealization){
                        CreateRelations(object, target);
                        break;
                }
                break;
```

```
		}
}


protected void CreateRelations (EObject object, Package target) {

		ClassifiersLists classifiers = new ClassifiersLists((Relationship) object, target);

		if ((classifiers.subclasses!=null)&&(classifiers.superclasses!=null)) {

			for (Iterator<Classifier> iter1 =
					classifiers.subclasses.iterator();iter1.hasNext();){
				Classifier subclass = iter1.next();

				for (Iterator<Classifier> iter2 =
					classifiers.superclasses.iterator();iter2.hasNext();){
					Classifier superclass = iter2.next();

					if (object instanceof InterfaceRealization) {
						UMLTools.copyIRealization((InterfaceRealization)
											object,
								(Class) subclass, (Interface)
											superclass);
					}

					if (object instanceof Generalization) {
						UMLTools.copyGeneralization((Generalization)
											object,
								(Classifier) subclass,
									(Classifier) superclass);
					}

				}
			}
		}
}


@SuppressWarnings("unchecked")
protected void renameProperties (EObject object, Package target) {



		if ((object instanceof Property) || (object instanceof Parameter)){

			TypedElement element = (TypedElement) object;
			Type type = element.getType();

			try {

				if (type.isTemplateParameter()) {

					List <Type> actualtypes =
						CheckTools.searchBinding((Element) type, target);

					if (actualtypes!=null) {
						element.setType(actualtypes.get(0));
					}

				}
			} catch (NullPointerException ee) { }

		}

		if (object instanceof Property) {

			Property atri = (Property) object;

			if (atri.getAssociation()!=null) {
				atri.setName(atri.getType().getName().toLowerCase());
			}
		}

}


protected void save(Package model) {

		try {
			UMLModeler.saveModelResource(model);
			out("Modelo Guardado");

		} catch (IOException e) {
			err(e.getMessage());
		}

}
```

```
	protected void save(org.eclipse.uml2.uml.Package final_model, URI final_path) {
		try {
			UMLModeler.saveModelResourceAs(final_model, final_path);

		} catch (IOException e) {
			err(e.getMessage());
		}
	}


	protected class TypesLists {

		protected List<Type> types1 = new ArrayList<Type>();
		protected List<Type> types2 = new ArrayList<Type>();

		@SuppressWarnings("unchecked")
		protected TypesLists(Association association, Package package_){

			List<Type> types = association.getEndTypes();

			if (types.get(0).isTemplateParameter()){
				types1 = CheckTools.searchBinding((Element) types.get(0),
									package_);
			} else{
				Type type1 = (Type) CheckTools.searchNamedElement(types.get(0),
									package_);
				types1.add(type1);
			}

			if (types.size()==1) {
				types2 = types1;
			}
			else {

				if (types.get(1).isTemplateParameter()){
					types2 = CheckTools.searchBinding((Element) types.get(1),
									package_);
				} else {
					Type type2 = (Type) CheckTools.searchNamedElement(
										types.get(1), package_);
					types2.add(type2);
				}
			}
		}

	}


	protected class ClassifiersLists {

		protected List<Classifier> subclasses = new ArrayList<Classifier>();
		protected List<Classifier> superclasses = new ArrayList<Classifier>();

		@SuppressWarnings("unchecked")
		protected ClassifiersLists(Relationship relation, Package package_){

			List<Element> classifiers = relation.getRelatedElements();

			if (((Classifier) classifiers.get(0)).isTemplateParameter()){
				subclasses = CheckTools.searchBinding(classifiers.get(0),
									package_);
			} else{
				Classifier classifier1 = (Classifier)
								CheckTools.searchNamedElement(
						((Classifier) classifiers.get(0)), package_);
				subclasses.add(classifier1);
			}

			if (((Classifier) classifiers.get(1)).isTemplateParameter()){
				superclasses = CheckTools.searchBinding(classifiers.get(1),
									package_);
			} else {
				Classifier classifier2 = (Classifier)
								CheckTools.searchNamedElement(
						((Classifier) classifiers.get(1)), package_);
				superclasses.add(classifier2);
			}
		}
	}
}
```

Class UMLTools

```
class UMLTools {

        static void copyParameter(Parameter parameter, Parameter newparameter) {

                newparameter.setName(parameter.getName());
                newparameter.setType(parameter.getType());
                newparameter.setDirection(parameter.getDirection());
                newparameter.setDefaultValue(parameter.getDefaultValue());
                newparameter.setVisibility(parameter.getVisibility());

                if (parameter.isMultivalued()){
                        newparameter.setLower(parameter.lowerBound());
                        newparameter.setUpper(parameter.upperBound());
                }

                newparameter.setIsOrdered(parameter.isOrdered());
                newparameter.setIsUnique(parameter.isUnique());
                newparameter.setIsException(parameter.isException());
                newparameter.setIsStream(parameter.isStream());
                newparameter.setEffect(parameter.getEffect());
        }


        static void copyAttribute(Property attribute, Property newattribute) {

                newattribute.setName(attribute.getName());
                newattribute.setType(attribute.getType());
                newattribute.setDefaultValue(attribute.getDefaultValue());
                newattribute.setVisibility(attribute.getVisibility());

                if (attribute.isMultivalued()){
                        newattribute.setLower(attribute.lowerBound());
                        newattribute.setUpper(attribute.upperBound());
                }

                newattribute.setIsLeaf(attribute.isLeaf());
                newattribute.setIsOrdered(attribute.isOrdered());
                newattribute.setIsStatic(attribute.isStatic());
                newattribute.setIsUnique(attribute.isUnique());
                newattribute.setIsDerived(attribute.isDerived());
                newattribute.setIsDerivedUnion(attribute.isDerivedUnion());
                newattribute.setIsReadOnly(attribute.isReadOnly());
        }


        static void copyOperation(Operation operation, Operation newoperation) {

                newoperation.setName(operation.getName());
                newoperation.setVisibility(operation.getVisibility());
                newoperation.setIsLeaf(operation.isLeaf());
                newoperation.setIsStatic(operation.isStatic());
                newoperation.setIsAbstract(operation.isAbstract());
                newoperation.setIsQuery(operation.isQuery());
                newoperation.setConcurrency(operation.getConcurrency());

                for (Iterator<Parameter> iter = operation.getOwnedParameters().iterator();
                                                                        iter.hasNext();) {
                        Parameter newparameter = newoperation.createOwnedParameter(null, null);
                        copyParameter(iter.next(), newparameter);
                }

        }


        static Class copyClass(Class class_, Package target) {

                Class copy = target.createOwnedClass(class_.getName(),class_.isAbstract());

                for (Iterator<Property> iter = class_.getOwnedAttributes().iterator();
                                                                        iter.hasNext();) {

                        Property attribute = iter.next();

                        if (attribute.getAssociation()==null){//si es miembro de una asociacion se
                                                                        creara en la asociacion
                                Property newattribute = copy.createOwnedAttribute(null,null);
                                copyAttribute(attribute, newattribute);
                        }
                }

                for (Iterator<Operation> iter = class_.getOwnedOperations().iterator();
                                                                        iter.hasNext();) {
```

```
                Operation newoperation = copy.createOwnedOperation(null,null,null);
                copyOperation(iter.next(),newoperation);
        }

        copy.setVisibility(class_.getVisibility());
        copy.setIsActive(class_.isActive());
        copy.setIsLeaf(class_.isLeaf());

        return copy;
}


static Interface copyInterface(Interface interface_, Package target) {

        Interface copy = target.createOwnedInterface(interface_.getName());

        for (Iterator<Operation> iter = interface_.getOwnedOperations().iterator();
                                                                iter.hasNext();) {
                Operation newoperation = copy.createOwnedOperation(null,null,null);
                copyOperation(iter.next(),newoperation);
        }

        copy.setVisibility(interface_.getVisibility());
        copy.setIsAbstract(interface_.isAbstract());
        copy.setIsLeaf(interface_.isLeaf());

        return copy;

}


static Generalization copyGeneralization(Generalization general,
                Classifier subclass, Classifier superclass) {

        Generalization copy = subclass.createGeneralization(superclass);
        copy.setIsSubstitutable(general.isSubstitutable());

        return copy;

}


static InterfaceRealization copyIRealization(InterfaceRealization irealiza,
                Class subclass, Interface superclass) {

        InterfaceRealization copy = subclass.createInterfaceRealization(irealiza.getName(),
                                                                        superclass);

        copy.setVisibility(irealiza.getVisibility());
        copy.setMapping(irealiza.getMapping());

        return copy;
}


static Association copyAssociation(Association association,
                Type copytype1, Type copytype2) {


        List<Property> members = association.getMemberEnds();
        Property member1 = (Property) (members.get(0));
        Property member2 = (Property) (members.get(1));

        Association copy = copytype2.createAssociation(member1.isNavigable(),
                        member1.getAggregation(), "", 1, 1, copytype1,
                                                        member2.isNavigable(),
                        member2.getAggregation(),"",1, 1);

        List<Property> copymembers = copy.getMemberEnds();
        Property copymember1 = (Property) (copymembers.get(0));
        Property copymember2 = (Property) (copymembers.get(1));

        copyAttribute(member1, copymember1);
        copyAttribute(member2, copymember2);

        copy.setName(association.getName());
        copy.setVisibility(association.getVisibility());
        copy.setIsLeaf(association.isLeaf());
        copy.setIsAbstract(association.isAbstract());

        copymember1.setType(copytype1);  // se redefinen los tipos que han sido
        copymember2.setType(copytype2); // modificados al copiar los attributos

        copymember1.setName(copytype1.getName().toLowerCase());  //y tambien los
        copymember2.setName(copytype2.getName().toLowerCase());  // nombres

        return copy;
```

```
	}


	static Package createPackage(Package package_, Package target) {

		Package copy = target.createNestedPackage(package_.getName());
		copy.setVisibility(package_.getVisibility());

		return copy;
	}

}
```

Class CheckTools

```
public class CheckTools {


	static boolean checkSubstitutions(TemplateBinding binding) {

		boolean check = false;

		TemplateParameterSubstitution subs = null;
		ParameterableElement formal, formalsubs;
		String formalID, formalsubsID;

		TemplateSignature signature = binding.getSignature();
		List<TemplateParameter> templates = signature.getOwnedParameters();

		List<TemplateParameterSubstitution> substitutions =
							binding.getParameterSubstitutions();


		for(Iterator<TemplateParameter> iter1 = templates.iterator();iter1.hasNext();) {

			check = false;
			formal = (ParameterableElement) iter1.next().getParameteredElement();
			formalID = formal.eResource().getURIFragment(formal);

			if ((getStereotype(formal)==4)||(getStereotype(formal)==9)) {
				check = true;
			}
			else {
				for(Iterator<TemplateParameterSubstitution> iter2 =
						substitutions.iterator();iter2.hasNext();) {

					subs = iter2.next();
					formalsubs = subs.getFormal().getParameteredElement();
					formalsubsID =
						formalsubs.eResource().getURIFragment(formalsubs);

					if (formalID.equals(formalsubsID)) {
						check = true;
					}

				}
			}

			if (!check)
				return false;

		}

		return true;
	}


	static boolean validObject(EObject object) {

		if (object instanceof Element) {
			if (object instanceof ParameterableElement) {
				if (((ParameterableElement) object).isTemplateParameter()) {
					return false;
				}
				else return true;
			}
			else return true;
		}
		else return false;
```

```
}

static int getStereotype(EObject object) {

	int value = 0;


	if (object instanceof Element){

		List<Stereotype> list = ((Element) object).getAppliedStereotypes();


		for (Iterator<Stereotype> iter = list.iterator();iter.hasNext();){

			String ster = iter.next().getName();

			if (ster.equals("new")) {
				value++;
			}

			if (ster.equals("add")) {
				value+= 2;
			}

			if (ster.equals("delete")) {
				value+= 3;
			}

			if (ster.equals("optional")) {
				value+= 4;
			}

			if (ster.equals("rename")) {
				value+= 5;
			}

			//PatternSpec extension point
		}
	}

	return value;
}


@SuppressWarnings("unchecked")
static List searchBinding(Element element, Package target){

	String templateID, bindingID;
	TemplateBinding binding = null;
	List<ParameterableElement> actuals = null;


	for (Iterator<EObject> iter = target.eAllContents();iter.hasNext();) {
		EObject eobject = (EObject) iter.next();

		if (eobject instanceof TemplateBinding) {
			binding = (TemplateBinding) eobject;

			TemplateParameter template = ((ParameterableElement)
                                                element).getTemplateParameter();
			templateID = template.eResource().getURIFragment(template);

			for (Iterator<TemplateParameterSubstitution> iter2 = binding
                                .getParameterSubstitutions().iterator();
                                                iter2.hasNext();;) {

				TemplateParameterSubstitution subs = iter2.next();
				bindingID =
                 subs.getFormal().eResource().getURIFragment(subs.getFormal());

				if (templateID.equals(bindingID)){
                                actuals = subs.getActuals();
				}
			}
		}
	}

	return actuals;
}


static NamedElement searchNamedElement(NamedElement element, Package target) {

	NamedElement search = element;
	TreeIterator<EObject> modelElements = target.eAllContents();

	for (Iterator<EObject> iter = modelElements;iter.hasNext();) {
```

```
                EObject eobject = (EObject) iter.next();

                if (eobject instanceof NamedElement) {

                        String name = ((NamedElement) eobject).getName();

                        try{
                                if (name.equals(element.getName())){
                                        search = (NamedElement) eobject;
                                }
                        } catch (NullPointerException e) { }
                }
        }

        return search;
}


static NamedElement searchNamedElement(String searchedname, Package target) {

        NamedElement search = null;
        TreeIterator<EObject> modelElements = target.eAllContents();

        for (Iterator<EObject> iter = modelElements;iter.hasNext();) {
                EObject eobject = (EObject) iter.next();

                if (eobject instanceof NamedElement) {

                        String name = ((NamedElement) eobject).getName();

                        try{
                                if (name.equals(searchedname)){
                                        search = (NamedElement) eobject;
                                }
                        } catch (NullPointerException e) { }
                }
        }

        return search;
}


static Package assignPackage(EObject object, Package target, Package pattern) {

        Package location = null;
        Package owner = ((Element) object).getNearestPackage();

        if (object instanceof Package) {
                owner = ((Package) object).getNestingPackage();
        }

        String ownerID = owner.eResource().getURIFragment(owner);
        String patternID = pattern.eResource().getURIFragment(pattern);

        if (patternID.equals(ownerID)) {
                return target;

        } else {
                location = (Package) CheckTools.searchNamedElement((NamedElement) owner,
                                                                        target);
        }

        return location;
}


static Association searchAssociation(Association association, Type type1,
                Type type2, Package target) {

        String type1ID = type1.eResource().getURIFragment(type1);
        String type2ID = type2.eResource().getURIFragment(type2);

        List<Property> members = association.getMemberEnds();
        Property member1, member2;


        Type testtype1, testtype2;
        String test1ID = null;
        String test2ID = null;


        for (Iterator<EObject> iter = target.eAllContents();iter.hasNext();) {
                        EObject object = iter.next();

                if (object instanceof Association) {

                        Association testasso = (Association) object;
                        List<Type> testtypes = testasso.getEndTypes();
```

```
                                int testsize = testtypes.size();

                                switch(testsize) {

                                case 1:
                                        testtype1 = testtypes.get(0);
                                        test1ID = testtype1.eResource().getURIFragment(testtype1);
                                        testtype2 = testtype1;
                                        test2ID = test1ID;
                                        break;

                                case 2:
                                        testtype1 = testtypes.get(0);
                                        test1ID = testtype1.eResource().getURIFragment(testtype1);
                                        testtype2 = testtypes.get(1);
                                        test2ID = testtype2.eResource().getURIFragment(testtype2);
                                        break;

                                }

                                if ((test1ID.equals(type1ID)&&(test2ID.equals(type2ID)))||

        (test1ID.equals(type2ID)&&(test2ID.equals(type1ID)))) {

                                        List<Property> testmembers = testasso.getMemberEnds();
                                        Property testmember1 = testmembers.get(0);
                                        Property testmember2 = testmembers.get(1);

                                        if (test1ID.equals(type1ID)&&(test2ID.equals(type2ID))) {
                                                member1 = members.get(0);
                                                member2 = members.get(1);
                                        } else {
                                                member1 = members.get(1);
                                                member2 = members.get(0);
                                        }

                                        if ((member1.isNavigable()==testmember1.isNavigable())&&

        (member2.isNavigable()==testmember2.isNavigable())) {

                                                if
((member1.getAggregation().getValue()==testmember1.getAggregation().getValue())

        &&(member2.getAggregation().getValue()==testmember2.getAggregation().getValue())){

                                                        if
((member1.lowerBound()==testmember1.lowerBound())

        &&(member2.lowerBound()==testmember2.lowerBound())){

                                                                if
((member1.upperBound()==testmember1.upperBound())

        &&(member2.upperBound()==testmember2.upperBound())){

                                                                return testasso;

                                                                }
                                                        }
                                                }
                                        }
                                }
                        }
                }

                return null;
        }


        static Property searchProperty(Property attribute, List<Property> properties) {

                for (Iterator<Property> iter = properties.iterator(); iter.hasNext();) {

                        Property prop = iter.next();


                        if ((attribute.getName().equals(prop.getName()))
                                        &&(attribute.getType()==prop.getType())
                                        &&(attribute.getVisibility()==prop.getVisibility())
                                        &&(attribute.isStatic()==prop.isStatic())
                                        //imponer mas condiciones para chequear la igualdad de
                                                                                        atributos
                                        ){

                                        return prop;
                        }
                }
```

```
        return null;
}


static Operation searchOperation(Operation operation, List<Operation> operations){

        for (Iterator<Operation> iter = operations.iterator();iter.hasNext();){

                Operation oper = iter.next();

                if ((operation.getName().equals(oper.getName()))
                                &&(operation.getType()==oper.getType())
                                &&(operation.getVisibility()==oper.getVisibility())
                                &&(operation.isStatic()==oper.isStatic())
                                &&(operation.isAbstract()==oper.isAbstract())
                                //imponer mas condiciones para chequear la igualdad de
                                                                                operaciones
                ){

                                return oper;
                }

        }

        return null;
}


static boolean isPackageImported(Package pack, Package target) {

        String packID =  pack.eResource().getURIFragment(pack);
        String modelpackID;
        Package modelpack = null;

        List<PackageImport> targetimports = target.getPackageImports();

        for(Iterator<PackageImport> iter = targetimports.iterator(); iter.hasNext();) {

                modelpack = iter.next().getImportedPackage();
                modelpackID = modelpack.eResource().getURIFragment(modelpack);

                if (packID.equals(modelpackID))
                        return true;

        }

        return false;
}
                                                                                                }
```

Anexo 3. Documentación Java

Package org.aslab.asys.models.patterns.tools

Class Summary

CheckTools

Coleccion de metodos para realizar comprobaciones internas durante la ejecucion de PatternApplier

PatternApplier

Clase principal de la herramienta PatternApplier.

PatternParameters

Metodos para manejar los parametros que se le pasan al patron

PatternTools

Metodos y atributos usados por la herramienta para analizar los elementos y ordenar las transformaciones

PatternTools.ClassifiersLists

Listas de extremos de relaciones

PatternTools.PatternBinding

Clase que consta de una paquete y un enlace.

PatternTools.PlugletParameters

Clase que contiene los parametros del pluglet: - Modelo al que se le aplica el patron.

PatternTools.TypesLists

Listas de extremos de asociaciones

UMLTools

Coleccion de metodos para copiar elementos de UML

---

org.aslab.asys.models.patterns.tools

Class CheckTools

```
java.lang.Object
    |
    +--org.aslab.asys.models.patterns.tools.CheckTools
```

---

public class **CheckTools**
extends java.lang.Object

Coleccion de metodos para realizar comprobaciones internas durante la ejecucion de PatternApplier

Author:

Eusebio Alarcon Romero

Constructors

CheckTools

```
public  CheckTools()
```

Methods

assignPackage

```
static org.eclipse.uml2.uml.Package
assignPackage(org.eclipse.emf.ecore.EObject object,
org.eclipse.uml2.uml.Package target,
org.eclipse.uml2.uml.Package pattern)
```

Asigna el paquete donde un elemento deberia ser creado en el modelo de destino

Parameters:

object - Elemento del patron que podria copiarse en el modelo de destino
target - Modelo de destino
pattern - Paquete del patron

Returns:

Paquete en el modelo de destino donde se copiaria el elemento del patron

checkSubstitutions

```
static boolean
checkSubstitutions(org.eclipse.uml2.uml.TemplateBinding
binding)
```

Comprueba si hay una substituicion para cada plantillas del patron en la que la substitucion no sea opcional

Parameters:

binding - Enlace del modelo al patron del que se comprueban las substituiciones

Returns:

True si no faltan substituciones, false en caso contrario

---

getStereotype

```
static int getStereotype(org.eclipse.emf.ecore.EObject object)
```

Asigna un valor numerico conforme al estereotipo aplicado

Parameters:

object - Objecto al que se le analizan los estereotipos

Returns:

Entero relacionado con el estereotipo aplicado

---

isPackageImported

```
static boolean isPackageImported(org.eclipse.uml2.uml.Package
pack,
                                 org.eclipse.uml2.uml.Package
target)
```

Comprueba si un paquete concreto esta importado en un modelo

Parameters:

pack - El paquete que se comprueba si esta importado
target - El modelo donde se busca la importacion

Returns:

True si el paquete esta importado, false si no.

---

searchAssociation

```
static org.eclipse.uml2.uml.Association
searchAssociation(org.eclipse.uml2.uml.Association association,
org.eclipse.uml2.uml.Type type1,
org.eclipse.uml2.uml.Type type2,
org.eclipse.uml2.uml.Package target)
```

Comprueta la existencia en el modelo de destino de una asociacion entre dos tipos concretos con las mismas caracteristicas a una asociacion dada

Parameters:

association - Asociacion de la que se comprueba la existencia
type1 - Tipo de uno de los extremos de la asociacion
type2 - Tipo de uno de los extremos de la asociacion
target - Asociacion encontrada o null si no existe

Returns:

---

searchBinding

```
static java.util.List
searchBinding(org.eclipse.uml2.uml.Element element,
org.eclipse.uml2.uml.Package target)
```

Busca en un modelo los elementos que realizan el rol de un elemento parametrizado

Parameters:

element - Elemento parametrizado
target - Paquete donde busca los elementos

Returns:

Lista de elementos que realizan el rol del elemento parametrizado

---

searchNamedElement

```
static org.eclipse.uml2.uml.NamedElement
searchNamedElement(java.lang.String searchedname,
org.eclipse.uml2.uml.Package target)
```

Busca en un modelo un elemento con un nombre concreto

Parameters:

searchedname - Nombre de referencia para la busqueda
target - Paquete donde se busca el elemento

Returns:

Elemento con el mismo nombre, o null si no existe un elemento con el mismo nombre

---

searchNamedElement

```
static org.eclipse.uml2.uml.NamedElement
searchNamedElement(org.eclipse.uml2.uml.NamedElement element,
org.eclipse.uml2.uml.Package target)
```

Busca en un modelo un elemento con el mismo nombre a uno dado

Parameters:

element - Elemento cuyo nombre se usa de referencia
target - Paquete donde se busca el elemento

Returns:

Elemento con el mismo nombre, o el propio elemento de referencia si no existe un elemento con el mismo nombre

---

searchOperation

```
static org.eclipse.uml2.uml.Operation
searchOperation(org.eclipse.uml2.uml.Operation operation,
java.util.List operations)
```

Busca entre una lista de operaciones una similar a otra dada

Parameters:

operation - Operacion a buscar
operations - Lista de operaciones para chequear

Returns:

La operacion encontrada o nul si no la hay

---

searchProperty

```
static org.eclipse.uml2.uml.Property
searchProperty(org.eclipse.uml2.uml.Property attribute,
java.util.List properties)
```

Busca entre una lista de propiedades una similar a otra dada

Parameters:

attribute - Propiedad a buscar
properties - Lista de propiedades para chequear

Returns:

La propiedad encontrada o null si no la hay

---

validObject

```
static boolean validObject(org.eclipse.emf.ecore.EObject
object)
```

Comprueba si el elemento es valido para manejarlo en el caso de estar estereotipado

Parameters:

Object - Objeto obtenido del patron que se analiza

Returns:

Devuelve true si es un objeto valido

---

org.aslab.asys.models.patterns.tools

Class PatternApplier

```
java.lang.Object
    |
    +--com.ibm.xtools.pluglets.Pluglet
        |
        +--PatternParameters
            |
            +--PatternTools
                |
                +--org.aslab.asys.models.patterns.tools.PatternApplier
```

< Constructors > < Methods >

---

public class **PatternApplier**
extends PatternTools

Clase principal de la herramienta PatternApplier. Controla el flujo de ejecucion

Author:
EuMola

Constructors

PatternApplier

```
public  PatternApplier()
```

Methods

plugletmain

```
public void plugletmain(java.lang.String[] args)
```

---

org.aslab.asys.models.patterns.tools

Class PatternParameters

```
java.lang.Object
    |
    +--com.ibm.xtools.pluglets.Pluglet
        |
        +--org.aslab.asys.models.patterns.tools.PatternParameters
```

Direct Known Subclasses:
PatternTools

< Constructors > < Methods >

---

public class **PatternParameters**
extends com.ibm.xtools.pluglets.Pluglet

Metodos para manejar los parametros que se le pasan al patron

Author:
EuMola

Constructors

PatternParameters

```
public  PatternParameters()
```

Methods

renameElements

```
protected void renameElements(org.eclipse.emf.ecore.EObject
object,
                              org.eclipse.uml2.uml.Package
target,
                              org.eclipse.uml2.uml.Package
pattern)
```

Renombra elementos nuevos creados en el modelo a partir de nombres elementos del modelo que substituyen roles en las plantillas. Los elementos que seran renombrados estan indicados en la especificacion del patron.

Parameters:

object - Elemento analizado del modelo para renombrar si tiene el nombre parametrizado
target - Modelo de destino en donde se renombra los elementos
pattern - Patron

---

solveClass

```
protected void solveClass(org.eclipse.uml2.uml.Class formal,
                          org.eclipse.uml2.uml.Class actual)
```

Resuelve los Template Parameter de tipo Class

Parameters:

formal - Clase en el patron (formal)
actual - Clase del modelo que asume el rol (actual)

---

solveInterface

```
protected void solveInterface(org.eclipse.uml2.uml.Interface
formal,
                              org.eclipse.uml2.uml.Interface
actual)
```

Resuelve los Template Parameter de tipo Interface

Parameters:

formal - Interfaz en el patron (formal)
actual - Interfaz del modelo que asume el rol (actual)

---

solveTemplates

```
protected void
solveTemplates(org.eclipse.uml2.uml.TemplateBinding binding)
```

Maneja las transformaciones de los elementos en plantillas

Parameters:

binding - Enlace del modelo sobre el que se aplica el patron

---

org.aslab.asys.models.patterns.tools

Class PatternTools

```
java.lang.Object
    |
    +--com.ibm.xtools.pluglets.Pluglet
        |
        +--PatternParameters
            |
            +--org.aslab.asys.models.patterns.tools.PatternTools
```

Direct Known Subclasses:
PatternApplier

< Fields > < Constructors > < Methods >

---

class **PatternTools**
extends PatternParameters

Metodos y atributos usados por la herramienta para analizar los elementos y ordenar las transformaciones

Author:
Eusebio Alarcon Romero

Fields

delete

```
protected boolean delete
```

---

loop

```
protected int loop
```

Constructors

PatternTools

```
PatternTools()
```

Methods

CreateRelations

```
protected void CreateRelations(org.eclipse.emf.ecore.EObject
object,
                               org.eclipse.uml2.uml.Package
target)
```

Gestiona la creacion de las relaciones Generalization y InterfaceRealization

Parameters:

object - La relacion a crear
target - El paquete donde se crea la relacion

---

createElement

```
protected void createElement(org.eclipse.emf.ecore.EObject
object,
                             org.eclipse.uml2.uml.Package
target)
```

Ordena la creacion de los elementos del patron que se quieren incluir o eliminar en el modelo La variable loop nos indica en que ciclo de transformacion de elementos se encuentra el programa

Parameters:

object - Elemento a crear en el modelo de destino
target - Modelo de destino

---

err

```
protected static void err(java.lang.String error)
```

---

handleElement

```
protected void handleElement(org.eclipse.emf.ecore.EObject
object,
                             org.eclipse.uml2.uml.Package
target,
                             org.eclipse.uml2.uml.Package
pattern)
```

Analiza los estereotipos de los elementos del patron

Parameters:

object - Elemento de un patron
target - Modelo sobre el que se aplica el patron
pattern - Paquete principal del patron

---

organizePackageImports

```
protected void organizePackageImports(java.util.List imports,
org.eclipse.uml2.uml.Package target)
```

Organiza las paquetes importados del modelo final aÃ±adiendo los necesarios para aplicar el patron

Parameters:

pattern - Modelo del patron que puede tener paquetes importados que no estan en el modelo de destino
target - Modelo en el que se importan los paquetes

---

out

```
protected void out(java.lang.String output)
```

---

refactorTarget

```
protected void refactorTarget(org.eclipse.uml2.uml.Package
target)
```

Puede crear una copia del modelo sobre la que se aplicara el patron si se desea mantener intacto el modelo original

Parameters:

target - Modelo sobre el que se aplica el patron

---

renameProperties

```
protected void renameProperties(org.eclipse.emf.ecore.EObject
object)
```

Renombra los miembros de las asociaciones a partir de sus tipos

Parameters:

object - Elemento del modelo que puede ser renombrado

---

save

```
protected void save(org.eclipse.uml2.uml.Package model)
```

Guarda las modificaciones realizadas sobre un modelo

Parameters:

model - Modelo que se guarda

---

save

```
protected void save(org.eclipse.uml2.uml.Package final_model,
                    org.eclipse.emf.common.util.URI final_path)
```

Guarda un modelo como un recurso nuevo

Parameters:

final_model - Modelo que se guarda
final_path - Ruta del nuevo recurso

---

org.aslab.asys.models.patterns.tools

Class PatternTools.ClassifiersLists

```
java.lang.Object
    |
    +--org.aslab.asys.models.patterns.tools.PatternTools.ClassifiersLists
```

< Fields > < Constructors >

---

protected class **PatternTools.ClassifiersLists**
extends java.lang.Object

Listas de extremos de relaciones

Author:

EuMola

Fields

subclasses

```
protected java.util.List subclasses
```

superclasses

```
protected java.util.List superclasses
```

Constructors

PatternTools.ClassifiersLists

```
protected
PatternTools.ClassifiersLists(org.eclipse.uml2.uml.Relationship
relation,
org.eclipse.uml2.uml.Package package_)
```

---

org.aslab.asys.models.patterns.tools

Class PatternTools.PatternBinding

```
java.lang.Object
    |
    +--org.aslab.asys.models.patterns.tools.PatternTools.PatternBinding
```

< Fields > < Constructors >

protected class **PatternTools.PatternBinding**
extends java.lang.Object

Clase que consta de una paquete y un enlace. Usada para relacionar un patron con un enlace de un modelo a dicho patron

Fields

binding

```
protected org.eclipse.uml2.uml.TemplateBinding binding
```

pattern

```
protected org.eclipse.uml2.uml.Package pattern
```

Constructors

PatternTools.PatternBinding

```
  PatternTools.PatternBinding(org.eclipse.uml2.uml.Package
package_,
org.eclipse.uml2.uml.TemplateBinding binding_)
```

---

org.aslab.asys.models.patterns.tools

Class PatternTools.PlugletParameters

```
java.lang.Object
    |
    +--org.aslab.asys.models.patterns.tools.PatternTools.PlugletParameters
```

< Fields > < Constructors >

---

protected class **PatternTools.PlugletParameters**
extends java.lang.Object

Clase que contiene los parametros del pluglet: - Modelo al que se le aplica el patron. - Lista de patrones que se aplican al modelo con sus respectivos enlaces del modelo al patron

Fields

correctparameters

```
protected boolean correctparameters
```

---

source

```
protected java.util.List source
```

---

target

```
protected org.eclipse.uml2.uml.Package target
```

Constructors

PatternTools.PlugletParameters

```
public  PatternTools.PlugletParameters(java.util.List list)
```

---

org.aslab.asys.models.patterns.tools

Class PatternTools.TypesLists

```
java.lang.Object
    |
    +--org.aslab.asys.models.patterns.tools.PatternTools.TypesLists
```

< Fields > < Constructors >

---

protected class **PatternTools.TypesLists**
extends java.lang.Object

Listas de extremos de asociaciones

Author:
EuMola

Fields

types1

```
protected java.util.List types1
```

---

types2

```
protected java.util.List types2
```

Constructors

PatternTools.TypesLists

```
protected
PatternTools.TypesLists(org.eclipse.uml2.uml.Association
association,
                                    org.eclipse.uml2.uml.Package
package_)
```

---

org.aslab.asys.models.patterns.tools

Class UMLTools

```
java.lang.Object
    |
    +--org.aslab.asys.models.patterns.tools.UMLTools
```

< Constructors > < Methods >

---

class **UMLTools**
extends java.lang.Object

Coleccion de metodos para copiar elementos de UML

Author:
Eusebio Alarcon Romero

Constructors

UMLTools

```
UMLTools()
```

Methods

copyAssociation

```
static org.eclipse.uml2.uml.Association
copyAssociation(org.eclipse.uml2.uml.Association association,
org.eclipse.uml2.uml.Type copytype1,
org.eclipse.uml2.uml.Type copytype2)
```

Crea una asociacion con las caracteristicas de otra dada y entre unos tipos especificos

Parameters:

association - Asociacion a copiar
copytype1 - Tipo en un extremo de la asociacion
copytype2 - Tipo en el otro extremo de la asociacion

Returns:

Nueva asociacion creada con las caracteristicas de la asociacion a crear

---

copyAttribute

```
static void copyAttribute(org.eclipse.uml2.uml.Property
attribute,
                          org.eclipse.uml2.uml.Property
newattribute)
```

Copia las caracteristicas de un atributo en otro

Parameters:

attribute - Atributo a copiar
newattribute - Atributo modificado con las caracteristicas del atributo a copiar

---

copyClass

```
static org.eclipse.uml2.uml.Class
copyClass(org.eclipse.uml2.uml.Class class_,
org.eclipse.uml2.uml.Package target)
```

Crea una clase copia de otra dada

Parameters:

class_ - Clase que se copia
target - Modelo que contendra la nueva clase

Returns:

Nueva clase creada

copyGeneralization

```
static org.eclipse.uml2.uml.Generalization
copyGeneralization(org.eclipse.uml2.uml.Generalization general,
org.eclipse.uml2.uml.Classifier subclass,
org.eclipse.uml2.uml.Classifier superclass)
```

Crea una generalizacion entre unos clasificadores especificos

Parameters:

general - Generalizacion a copiar
subclass - Clasificador que extiende en la generalizacion
superclass - Clasificador que es extendido en la generalizacion

Returns:

Nueva generalizacion creada

copyIRealization

```
static org.eclipse.uml2.uml.InterfaceRealization
copyIRealization(org.eclipse.uml2.uml.InterfaceRealization
irealiza,
org.eclipse.uml2.uml.Class subclass,
org.eclipse.uml2.uml.Interface superclass)
```

Crea una realizacion de interfaz entre un clasificador y una interfaz especificos

Parameters:

irealiza - Realizacion de interfaz a copiar
subclass - Clasificador que implementa la interfaz
superclass - Interfaz que es implementada en la realizacion

Returns:

Nueva realizacion de interfaz creada

copyInterface

```
static org.eclipse.uml2.uml.Interface
copyInterface(org.eclipse.uml2.uml.Interface interface_,
org.eclipse.uml2.uml.Package target)
```

Crea una interfaz copia de otra dada

Parameters:

interface_ - Interfaz que se copia
target - Modelo que contendra la nueva interfaz

Returns:

Nueva interfaz creada

---

copyOperation

```
static void copyOperation(org.eclipse.uml2.uml.Operation
operation,
                          org.eclipse.uml2.uml.Operation
newoperation)
```

Copia las caracteristicas de una operacion en otra

Parameters:

operation - Operacion a copiar
newoperation - Operacion modificada con las caracteristicas de la operacion a copiar

---

copyParameter

```
static void copyParameter(org.eclipse.uml2.uml.Parameter
parameter,
                          org.eclipse.uml2.uml.Parameter
newparameter)
```

Copia las caracteristicas de un parametro en otro.

Parameters:

parameter - Parametro a copiar
newparameter - Parametro modificado con las caracteristicas del parametro a copiar

---

createPackage

```
static org.eclipse.uml2.uml.Package
createPackage(org.eclipse.uml2.uml.Package package_,
org.eclipse.uml2.uml.Package target)
```

Crea un nuevo paquete con las caracteristicas de otro paquete dado

Parameters:

package_ - Paquete tomado como modelo
target - Paquete que contendra al nuevo paquete creado

Returns:

Nuevo paquete creado con las caracteristicas del paquete tomado como modelo

Bibliografía

[1] J.L. Fernández Sánchez, F.J. Sánchez Alejo. *La Ingeniería de Sistemas Soportada en Modelos.* U.D. Proyectos, ETSI Industriales. Universidad Politécnica de Madrid.

[2] A. Monzón Díaz, J.L. Fernández Sánchez. *Verificación de los modelos en la ingeniería de sistemas.* Septiembre 2006.

[3] Julita Bermejo-Alonso, Ricardo Sanz, Manuel Rodríguez, y Carlos Hernández. *An* ontological framework for autonomous systems modeling. Autonomous Systems Laboratory (ASLab). Universidad Politécnica de Madrid, España.

[4] J. Bermejo-Alonso. *"OASys: ontology for autonomous systems," Ph.D. dissertation*. E.T.S.I.I.M., Universidad Politécnica de Madrid, 2010.

[5] Christopher Z. Garrett. *Software Modeling Introduction - What do you need from a modeling tool?* Marzo 2003.

[6] Enrique Hernández Orallo. *El Lenguaje Unificado de Modelado (UML).* Departamento de Informática de Sistemas y Computadores. Universidad Politécnica de Valencia, España.

[7] Joseph Schmuller. *Aprendiendo UML en 24 horas*. Prentice Hall.

[8] *OMG Unified Modeling Language, Superstructure. Version 2.2.* Febrero 2009.

[9] *MDA Guide Version 1.0.1.* OMG. 2003.

[10] Colin Yu. *Model-driven and pattern-based development using Rational Software Architect, Part 1: Overview of the model-driven development paradigm with patterns.* Noviembre 2006.

[11] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, y Michael Stal. *Pattern-Oriented Software Architecture Volume 1, a System of Patterns.* Wiley.

[12] Peter Swithinbank, Mandy Chessell, Tracy Gardner, Catherine Griffin, Jessica Man, Helen Wylie, y Larry Yusuf. *Patterns: Model-Driven Development using IBM Rational Software Architect.* Redbooks. IBM.

[13] Krzysztof Czarnecki and Simon Helsen. *Classification of Model Transformation Approaches.* OOPSLA'03. University of Waterloo, Canada.

[14] Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. *Design Patterns Application in UML*. IRISA/CNRS, Campus de Beaulieu. Rennes Cedex, France.

[15] Markus Schmidt. *Transformations of UML 2 Models Using Concrete Syntax Patterns.* Real-Time Systems Lab. Darmstadt University of Technology, Germany.

[16] Lidia Fuentes y Antonio Vallecillo. *Una introducción a los Perfiles UML*. Depto. de Lenguajes y Ciencias de la Computación. Universidad de Málaga, España.

[17] Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley.

[18] Ahmed Makady. *Generating UML models programmatically by using IMB Rational Software Architect.* Gulf Business Machines. IMB. Agosto 2008.

[19] Kenn Hussey. *Getting Started with UML2*. IBM. Julio 2006.

[20] Dave Kelsey. *Using Pluglets in IBM Rational Software Architect.* IMB. Noviembre 2006.

[21] Dusko Misic. *Authoring UML profiles using Rational Software Architect and Rational Software Modeler.* IBM. Software group. Septiembre 2005.

[22] Wikipedia. http://es.wikipedia.org http://en.wikipedia.org

[23] Guía de ayuda de RSA. IBM.

[24] Guía de ayuda de Eclipse.

Abreviaturas y Acrónimos

- API: Application Programming Interface. *(Interfaz de programación de aplicaciones):*
- APT: Advanced Packing Tool. *(Herramienta avanzada de empaquetado).*
- ASys: Autonomous Systems. *(Sistemas autónomos).*
- ASLab: Autonomous Systems Laboratory. *(Laboratorio de sistemas autónomos).*
- AWT: Abstract Window Toolkit. *(Kit de herramientas de ventana abstracta).*
- CORBA: Common Object Request Broker Architecture. *(Arquitectura común de intermediarios en peticiones de objetos.*
- DISAM: División de ingeniería de sistemas y automática.
- DSL: Domain Specific Language. *(Lenguaje específico del dominio).*
- EFX: Entity Fragment for XML Schema. *(Fragmento de entidad de esquema XML).*
- EJB: Enterprise Java Beans.
- EMX: Entity Model for XML Schema. *(Modelo de entidad de esquema XML).*
- GNOME: GNU Network Object Model Environment. *(Entorno de modelado de objetos de red GNU).*
- GNU: GNU is not UNIX. *(GNU no es Unix).*
- GPL: General Public License. *(Licencia pública general).*
- HTTP: Hypertext Transfer Protocol. *(Protocolo de transferencia de hipertexto).*
- IBM: International Business Machines. *(Máquinas de negocios internacionales).*
- IDE: Integrated Development Environment. *(Entorno de desarrollo integrado).*
- IDL: Interface Description Language. *(Lenguaje de descripción de interfaz).*
- JDT: Java Development Tools. *(Herramientas de desarrollo Java).*
- JET: Java Emitter Templates. *(Emisor de plantillas de Java).*
- MDA: Model-Driven Architecture. (*Arquitectura basada en modelos).*
- MDD: Model-Driven Development. *(Desarrollo basado en modelos).*
- MOF: Meta-Object Facility. *(Recursos para meta-objetos).*

- OASys: Ontology for Autonomous Systems. *(Ontología para sistemas autónomos).*
- OCL: Object Constraint Language. *(Lenguaje de restricción de objetos).*
- OMG: Object Managment Group. (*Grupo de gestión de objetos).*
- OMT: Object Modelling Tool. *(Herramienta de modelado de objetos).*
- OOPSLA: Object-Oriented Programming, Systems, Languages & Applications. *(Programación, sistemas, lenguajes y aplicaciones orientadas a objetos).*
- OOSE: Object-Oriented Software Engineering. *(Ingeniería de software orientada a objetos).*
- OWL: Ontology Web Language. *(Lenguaje web de ontologías).*
- PDM: Plarform Definition Model. *(Modelo de definición de la plataforma).*
- PIM: Platform Independent Model. *(Modelo independiente de la plataforma).*
- PSM: Platform Specific Model. *(Modelo específico de la plataforma).*
- RPM: RPM Package Manager. *(Gestor de paquetes RPM).*
- RSA: Rational Software Architect. *(Arquitecto de software Rational).*
- SAR: Servlet ARchive. *(Archivo servlet).*
- SIP: Session Initiation Protocol. *(Protocolo de inicio de sesiones).*
- SOA: Service-oriented architecture. *(Arquitectura orientada a servicios).*
- SQL: Structured Query Language. *(Lenguaje de consulta estructurado).*
- SWT: Standard Widget Toolkit. *(Kit de herramientas de widget estándar).*
- SysML: Systems Modeling Language. *(Lenguaje de modelado de sistemas).*
- UML: Unified Modeling Language. (*Lenguaje unificado de modelado).*
- WSDL: Web Services Description Language. *(Lenguaje de descripción de servicios webs).*
- XMI: XML Metadata Interchange. *(XML de intercambio de metadatos).*
- XML: Extensible Markup Language. *(Lenguaje de marcas extensibles).*
- XSD: XML Schema Definition. *(Definición de esquema XML).*