UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: MagicUWE4R: Una herramienta de refactoring en el modelado de aplicaciones Web
Autores: Miguel José Djebaile
Director: Alejandra Garrido
Codirector: Gustavo Rossi
Asesor profesional: -
Carrera: Licenciatura en Sistemas

Resumen

El punto central de esta tesis es la inclusión de la práctica de refactoring dentro de una metodología de desarrollo de aplicaciones Web existente. Es decir, se utiliza la técnica de refactoring (que siempre se relacionó con las metodologías ágiles y el código fuente) en el contexto del desarrollo de software dirigido por modelos (MDD).

Ante la ausencia de herramientas de refactoring aplicadas al desarrollo Web dirigido por modelos, se desarrolla MagicUWE4R, que implementa los refactorings para el modelo de navegación y presentación de la metodología UWE.

A su vez, se pone énfasis en el buen diseño de la herramienta, de manera que el motor de refactoring sea extensible para otros refactorings más complejos, mediante una implementación simple y elegante. Esto a causa de principalmente dos puntos claves: el uso de patrones de diseño y la composición de refactorings.

Palabras Claves

MagicUWE4R, refactoring, MDD, UWE, patrones de diseño, MagicDraw, plugin, modelo.

Conclusiones

La idea de esta tesis es conjugar el concepto de refactoring con el de MDD (desarrollo de software dirigido por modelos), aplicado a una metodología Web como es UWE. Es que, para que la industria adopte nuevos procesos que en la teoría suenan enriquecedores o más productivos, es necesario contar con herramientas adecuadas, y MagicUWE4R es el puntapié inicial en la integración de estas dos potentes metodologías

Trabajos Realizados

- Estudio del desarrollo de software dirigido por modelos (MDD), la metodología de modelado UWE, y la técnica de refactoring.
- Relevamiento de la API de MagicDraw, para implementar la herramienta de refactoring MagicUWE4R.
- Estudio y adaptación de un catálogo de diferentes refactorings que aplicará MagicUWE4R.
- Definición de arquitectura e implementación de la herramienta MagicUWE4R.

Trabajos Futuros

- Sincronización de una capa a otra. Es decir, que un refactoring aplicado a un modelo sea reflejado en otro modelo
- Construcción instantánea de una aplicación Web a partir de la generación automática de código. Construir una implementación base de manera automática a partir de los modelos diseñados y que los refactorings también se reflejen a nivel de código de manera automática.
- Implementación de otros refactorings

Fecha de la presentación: 07/12/2011

INDICE

Agradecimientos

La realización de mi Tesis implicó un esfuerzo conjunto donde participaron muchas personas. Agradezco a mis padres por haberme dado la oportunidad de formarme como persona y de darme acceso a la educación. Por su apoyo constante y sus consejos. Por disfrutar y sufrir a la par mío. Doy gracias a mi hermana quien fue mi modelo a seguir a lo largo de mi carrera universitaria.

Agradezco a mi Directora de Tesis, la Dra. Alejandra Garrido, por ser mi guía a lo largo del extenso proceso, ofreciéndome siempre su tiempo, sus consejos y sus conocimientos de tanta riqueza. Gracias al Dr. Gustavo Rossi, Co-Director de la Tesis, por sus aportes de jerarquía.

Se agradece a la Facultad de Informática de la UNLP, a todos los profesores, ayudantes y colaboradoes, por haberme cultivado y forjado quien soy hoy a nivel profesional.

No quiero olvidarme de aquellas personas que indirectamente vivieron junto a mí todo el proceso, dándome sus palabras de apoyo y confianza, y muchas veces resignando cosas por mi causa: amigos, familiares, compañeros de trabajo.

Doy gracias a Dios por haberme permitido caminar este sendero.

CAPÍTULO 1
Introducción

1.1 Presentación

Presenciamos una constante y veloz evolución de las aplicaciones Web, impulsada por diferentes factores: nuevos requerimientos emergentes, requerimientos existentes que necesitan adaptarse a la necesidad de los usuarios, nuevas tecnologías que ofrecen la oportunidad de mejorar tanto la interfaz de las aplicaciones como la interacción con las mismas, etc.

Sin embargo, en muchas ocasiones, esta evolución surge de los mismos desarrolladores, sobre la estructura de la aplicación, comportamiento o código fuente. Esto es crucial para conservar una aplicación mantenible. En estos casos, las modificaciones no aportan nueva funcionalidad sino que mejoran la existente hacia un modelo adaptable y escalable en el tiempo.

Pasemos a analizar dos conceptos importantes en las nuevas tendencias del desarrollo Web: refactoring y Model Driven Development (MDD), y veamos cómo actualmente se relacionan.

La técnica de **refactoring** [1] surge de la necesidad por parte de los desarrolladores de un cambio constante en la aplicación, manteniendo la calidad, reusabilidad, testeabilidad y confiabilidad ante cada uno de estos cambios. Martin Fowler define refactoring como "…técnica para la reestructuración de un sector de código existente, modificando su estructura interna sin cambiar el comportamiento externo. Su esencia se basa en una serie de pequeñas transformaciones que preservan su comportamiento. Cada transformación realiza poco, pero una secuencia de transformaciones puede producir una reestructuración importante." [2].

Refactoring es entonces el proceso de cambiar un sistema de software de tal manera que no altere el comportamiento externo del código y aún así mejora su estructura interna. Esto permite a las metodologías ágiles desarrollar software en dos pasos iterativos. En el primer paso se desarrolla el comportamiento esperado y en el segundo se incrementa la calidad y la estructura del código sin cambiar el comportamiento original [3].

Por otro lado, las metodologías de desarrollo de aplicaciones Web existentes, como por Ej. UWE [4], UWA, OOHDM [5], WebML, son *Model-Driven*, es decir

metodologías que siguen el desarrollo dirigido por modelos (**Model Driven Development -MDD-** [6]**)**. Éste se ha convertido en un nuevo paradigma de desarrollo de software. MDD promete mejorar el proceso de construcción de software basándose en un proceso guiado por modelos y soportado por potentes herramientas.

1.2 Motivación

Dada la importancia en la tendencia actual de permanente cambio y adaptación en las aplicaciones Web, es deseable poder contar con una herramienta de software que permita plasmar de manera automática los refactorings. En la actualidad, las aplicaciones Web forman parte del mainstream de un negocio, movimiento o entidad. Por medio de este recurso, la llegada a los destinatarios es mucho más directa, accesible y rica, ofreciendo una amplia gama de opciones a medida del cliente. Es por ello que la usabilidad de una aplicación Web pasa a ser un factor clave de la misma y el éxito de un negocio puede estar directamente relacionado con mantener una usabilidad óptima [7].

Existe una extensa investigación sobre patrones para mejorar la usabilidad [8], que detectan una serie de factores que contribuyen a la usabilidad de una aplicación Web.

A pesar de que el refactoring originalmente fue pensado sobre código, el concepto de refactoring puede ser generalizado a una técnica que mejore la estructura del software y no solamente a la representación del código. Se ha propuesto que esta técnica se aplique sobre los modelos de una aplicación, dando origen al concepto de **refactoring de modelos** [9].

Por otro lado, el refactoring, si no es automatizado, es muy difícil de lograr. Hoy en día hay muchas herramientas de refactoring de código, pero no así para modelos. De hecho la herramienta comercial MagicDraw [10], de gran potencia para modelar diagramas UML (Lenguaje Unificado de Modelado) [11], no provee ningún tipo de refactoring de modelos. Si hablamos del caso del desarrollo MDD para aplicaciones Web es prácticamente nula la existencia de este tipo de recurso.

Estudiando las herramientas MDD para aplicaciones Web vigentes, se llegó a la conclusión que la más completa y estándar es MagicUWE (para **UWE** -UML-Based Web Engineering- [4], escrita sobre MagicDraw); pero como, repetimos, MagicDraw no tiene ningún soporte de refactoring, se diseñó un motor de refactoring desde cero, y de manera tal que sea extensible fácilmente.

Lo que propone esta tesis es la construcción de una herramienta para utilizar refactoring, una práctica que se asocia a las metodologías ágiles y al código fuente, dentro del proceso de desarrollo de una aplicación Web dirigida por modelos. En particular nos centraremos en refactorings sobre los modelos de la metodología UWE.

1.3 Objetivo

El objetivo de esta tesis es incluir la práctica de refactoring dentro de una metodología de desarrollo de aplicaciones Web existente. Es decir, se utiliza la técnica de refactoring en el contexto del desarrollo de software dirigido por modelos (MDD).

Ante la ausencia de herramientas de refactoring aplicada a MDD, se desarrolla una denominada MagicUWE4R, que implementa los refactorings para el modelo de navegación y presentación de la metodología UWE, extendiendo la herramienta existente MagicUWE [12].

A su vez, se pone énfasis en desarrollar los refactorings en unidades atómicas, de manera que puedan componerse, y que el motor de refactorings sea extensible para otros refactoring más complejos. Es decir, se focalizó en poder crear un refactoring complejo a partir de la composición de refactorings más sencillos [13].

1.4 Organización de la Tesis

Esta tesis se organiza en los siguientes capítulos:

1. Introducción

Se presenta la tesis, narrando motivaciones, objetivos.

2. Trabajos Relacionados

Se hace una breve reseña de los conceptos y aplicaciones más importantes sobre los cuales se basó la tesis.

3. Arquitectura base

Se explica el mecanismo de extensión, a través de la API de MagicDraw, y la arquitectura de MagicUWE.

4. Arquitectura de MagicUWE4R

Se describe detalladamente la arquitectura de la herramienta desarrollada en la presente tesis, basada en la composición de refactorings y en el diseño orientado a patrones, y que llamamos MagicUWE4R.

5. Refactorings en el Modelo de Navegación

Se elucidan los refactorings en el modelo de navegación soportados por MagicUWE4R, detallando motivación, mecanismo y un ejemplo gráfico en la aplicación.

6. Refactorings en el Modelo de Presentación

Se ilustran los refactorings en el modelo de presentación soportados por MagicUWE4R, detallando motivación, mecanismo y un ejemplo gráfico en la aplicación.

7. Conclusiones

Se detallan las conclusiones y aportes de la tesis. A su vez se explican las limitaciones existentes y los trabajos futuros a desarrollarse.

CAPÍTULO 2
Trabajos Relacionados

Reseña de los conceptos y metodologías más importantes sobre los cuales se basó la tesis: Model Driven Development, Refactoring, Refactoring de Modelos, Refactoring en aplicaciones Web para mejorar usabilidad, UWE.

2.1 Model Driven Architecture (MDA)

2.1.1 Introducción a MDA

La OMG (Object Management Group) [14] es un consorcio dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos. Es una organización sin fines de lucro que promueve el uso de tecnologías orientadas a objetos mediante guías y especificaciones para las mismas.

Model-Driven Architecture (MDA) [15] es una iniciativa de OMG. Es una implementación de MDD (Model Driven Development). MDD no define tecnologías, herramientas, procesos o secuencia de pasos a seguir. La implementación MDA se ocupa de ello mediante el uso del UML [11] y meta-pasos a seguir en el desarrollo de sistemas. MDA se basa en la construcción y transformación de modelos.

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

2.1.2 El proceso MDA

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

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

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

Uno de los principales puntos de MDA, es el concepto de plataforma y el de independencia de plataformas. Una plataforma en MDA es un conjunto de subsistemas y tecnologías que proveen un conjunto coherente de funcionalidades.

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

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

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

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

El proceso de MDA se muestra a continuación. En la Figura 1 pueden verse los sucesivos refinamientos y transformaciones aplicados al modelo.

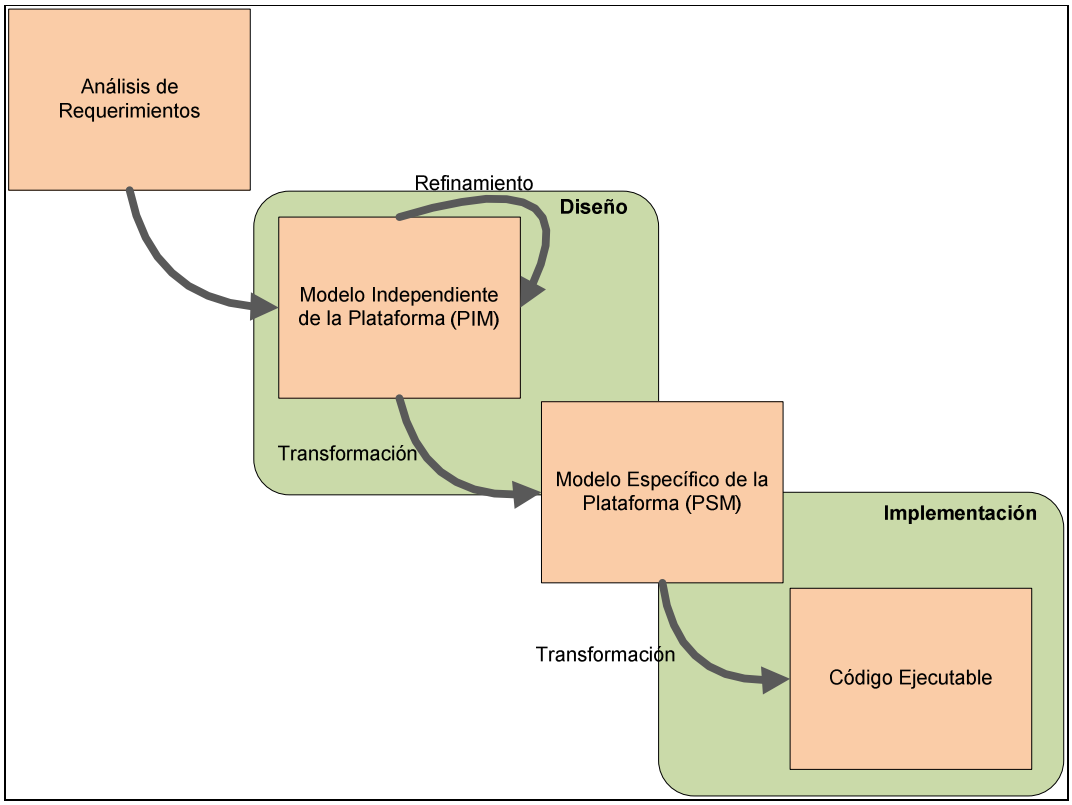


Figura 1. El proceso de MDA

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

La situación inversa de llevar el código hacia un modelo concreto es lo que se conoce como ingeniería inversa. MDA promueve la separación entre las responsabilidades de requerimientos del negocio y las responsabilidades tecnológicas.

La ventaja de esta **separación de responsabilidades o concerns** es que ambos aspectos pueden evolucionar individualmente sin generar dependencias entre sí.

Los diferentes tipos de modelos en MDA [15] son:

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

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

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

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

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

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

2.1.3 Ventajas de MDA

La ventaja principal de MDA está en la separación de responsabilidades o concerns. Por un lado se modelan los PIMs que representan los modelos del negocio,

y por otro lado se modelan los PSMs con los detalles tecnológicos. Esto permite que ambos modelos evolucionen por separado.

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

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

2.1.4 Desarrollo tradicional vs. Desarrollo con MDA

DESARROLLO TRADICIONAL

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

- Especificación de requerimientos
- Análisis detallado
- Diseño de una solución
- Codificación de la solución
- Pruebas
- Instalación

El siguiente gráfico muestra el proceso:

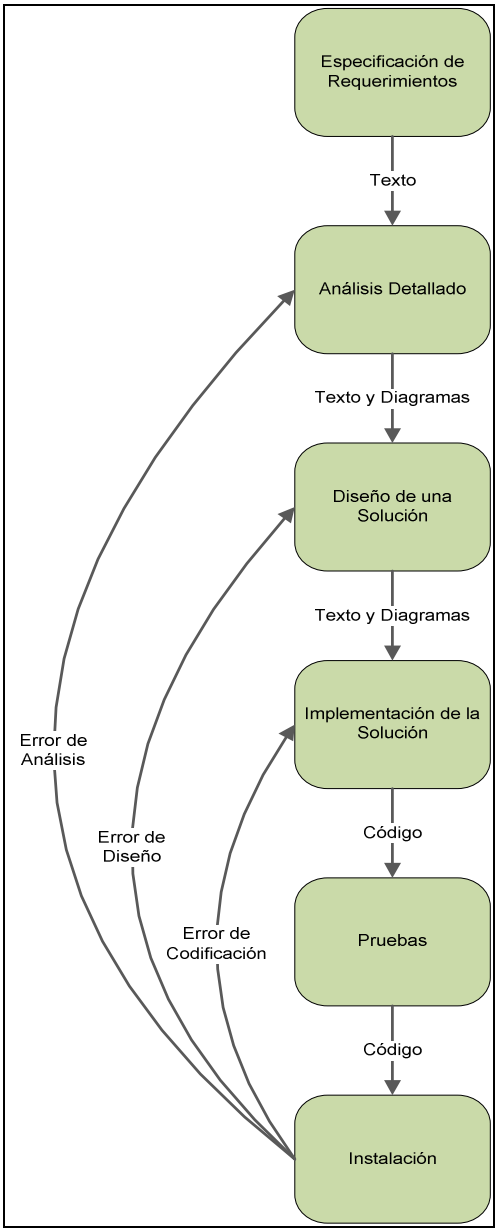


Figura 2. Proceso tradicional de desarrollo de software

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

- **El problema de la Productividad:** Durante el proceso tradicional se producen gran cantidad de documentos y diagramas, los cuales especifican requerimientos, diseño de componentes, etc. Gran parte de estos documentos

pierden valor durante la fase de codificación, y finalmente se llega a un punto en el que pierden relación con el sistema. Cuando el sistema cambia a lo largo del tiempo el problema es aún mayor: se hace imposible realizar los cambios en todas las fases (requerimientos, análisis, diseño, etc.), y las modificaciones se terminan realizando directamente en el código. Para sistemas complejos, los diagramas y documentación de alto nivel son imprescindibles, pero está faltando un soporte para que los cambios en cualquiera de las fases se trasladen fácilmente al resto.

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

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

- **El problema del Mantenimiento y la Documentación.** Documentar un sistema es una tarea costosa y que demanda mucho tiempo. Los desarrolladores de software no le dan a la documentación la importancia que realmente tiene, debido a que es mucho más usada en las etapas de mantenimiento y correcciones que durante el desarrollo en sí. Esto hace que no se le preste la atención necesaria a la documentación, y que la que se produce no sea de buena calidad. Una solución es que la documentación se genere directamente del código fuente, asegurándose que esté siempre actualizada. Sin embargo, la documentación de alto nivel (diagramas y texto) debe ser mantenida a mano.

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

DESARROLLO CON MDA

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

- ✓ **Productividad.** En MDA el desarrollo se basa en los PIM. Los PSMs son derivaciones semiautomáticas de éstos. Es necesario definir las transformaciones exactas, lo cual es una tarea especializada y difícil. Pero una vez implementada una transformación, puede reutilizarse en muchos desarrollos. En la generación de código a partir de los PSMs ocurre lo mismo.
  Este enfoque aísla los problemas específicos de las plataformas y ataca mejor las necesidades de los usuarios finales, ya que es posible agregar funcionalidades con menos esfuerzo. Gran parte del trabajo lo realizan las herramientas de transformación, y no los desarrolladores.

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

- ✓ **Interoperabilidad.** De un mismo PIM pueden generarse múltiples PSMs, cada uno abarcando una funcionalidad específica. En MDA, a las relaciones entre los diferentes PSM se las denomina *puentes*.
  Los PSMs no siempre pueden comunicarse entre sí, debido a que pueden pertenecer a distintas tecnologías. En MDA esto se soluciona generando tanto los PSMs como los puentes entre ellos. Estos puentes también son construidos por las herramientas de transformación.

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

2.1.5 Modificaciones al proceso de desarrollo con MDA

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

- **Análisis:** Los analistas del proyecto crean el PIM. En este modelo plasman las necesidades del cliente y las funcionalidades que el sistema debe proveer.
- **Diseño:** Un equipo especializado realiza la transformación del PIM a uno o más PSMs. Para este paso se requieren conocimientos específicos de distintas plataformas y arquitecturas, así como también de las herramientas y transformaciones disponibles. Con este conocimiento se logra una mejor elección de las plataformas y arquitecturas para cada caso en particular.
- **Codificación:** Esta fase se reduce a la generación del código fuente mediante herramientas especializadas. Los desarrolladores sólo deberán añadir las características que no pudieron ser reflejadas en los modelos, así como también realizar optimizaciones sobre el código generado.

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

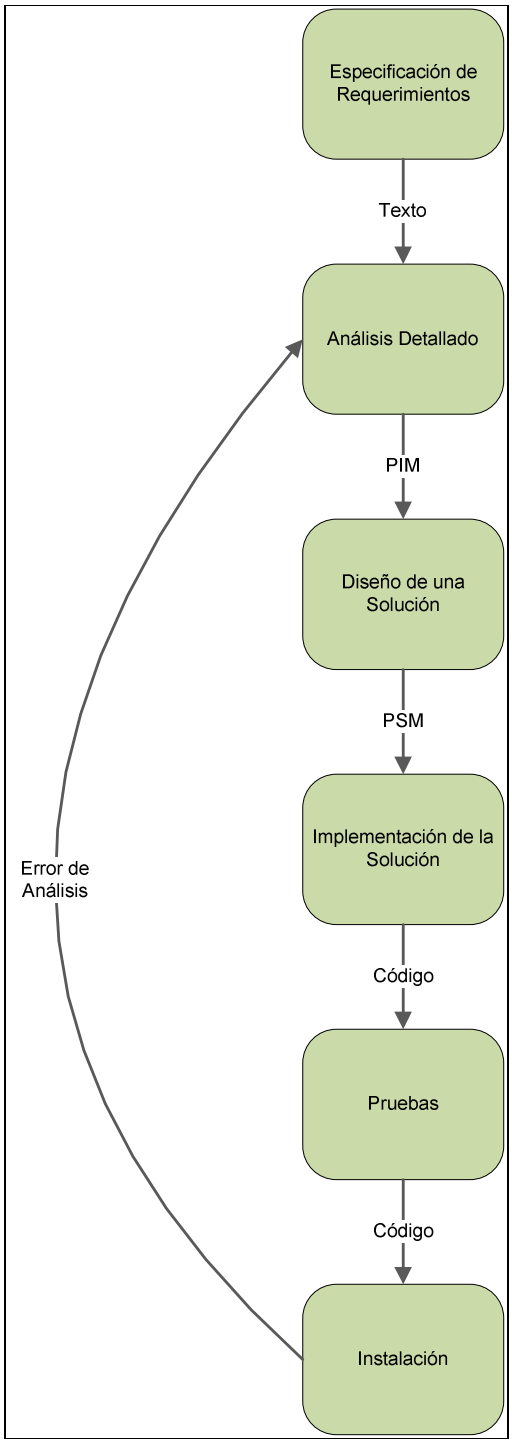


Figura 3. MDA en el proceso de desarrollo tradicional

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

2.2 Refactoring

Martin Fowler define al término **refactoring** [2] como:

- Refactoring (sustantivo): un cambio en la estructura interna del software para hacerlo más fácil de entender y menos costoso de modificar, sin cambiar el comportamiento observable del sistema.
- Refactoring (verbo): es la reestructuración de software mediante la aplicación de una serie de refactorings sin cambiar el comportamiento observable del sistema.

Refactoring desde el punto de vista de Fowler puede ser caracterizado por lo siguiente:

- Trata de la estructura interna del software
- Preserva el comportamiento observable
- Mejora una situación dada de acuerdo a un objetivo expresado informalmente; ejemplos de dichos objetivos son la reducción de costo de desarrollo, mejoras en la legibilidad, mantenimiento y velocidad de ejecución, demanda de memoria, etc.
- Los pasos de refactoring son pequeños y pueden ser combinados sistemáticamente permitiendo construir estrategias más sofisticadas
- Es una técnica constructiva basada en reglas que parte de una situación dada, un objetivo y una serie de pasos constructivos, para lograr dicho objetivo
- Es aplicado por desarrolladores de software
- La corrección de la aplicación de las reglas de refactoring es responsabilidad del desarrollador.

Esto significa que los refactorings son considerados transformaciones de software que reestructuran un programa mientras preservan su comportamiento. Por ejemplo, en el paradigma orientado a objetos, podría significar una redistribución de atributos y métodos en la jerarquía de clases para adecuar al software a futuras extensiones y cambios.

La aplicación de refactorings sobre código tiene las siguientes ventajas:

- ✓ *Mejora el diseño del software*: cuando se realizan cambios para alcanzar objetivos a corto plazo sin tener una completa visión del diseño total se pierde la estructura del código. La pérdida de estructura tiene un efecto acumulativo,

dificultando visualizar el diseño en el código, haciendo más difícil preservarlo y así más rápidamente decae el sistema. Aplicar regularmente refactorings ayuda a que el código mantenga su forma. Otro aspecto importante para mejorar el diseño es la eliminación de la duplicación de código. Los diseños pobres usualmente poseen más código innecesario, porque encontramos código que hace lo mismo en varios lugares.

- ✓ *Mejora el entendimiento del software*: al generar código más legible, éste comunica más fácilmente el propósito para el cual fue diseñado.

- ✓ *Ayuda a encontrar errores*: al mejorar el entendimiento del software se pueden ver aspectos sobre el diseño que antes no se observaban, por lo tanto se pueden detectar errores más claramente.

- ✓ *Agiliza el desarrollar código*, ya que los buenos diseños ayudan a desarrollar más rápidamente código al no tener que perder demasiado tiempo detectando y depurando errores.

Refactoring entonces, pasa a ser una actividad importante en los procesos de desarrollo de software. A continuación se detallan los procesos en los cuales se adecua la aplicación de la técnica de refactoring:

- → En la **propuesta MDA**, los modelos son los artefactos primarios del desarrollo de sistemas y las transformaciones de modelos bien definidas son claves para soportar la evolución de los modelos, su refinamiento y su realización en código.
- → En los procesos de **desarrollo ágiles,** como Scrum o XP [3] (eXtreme Programming, metodología desarrollada por Kent Beck). La idea central en XP es trabajar sobre un único caso de uso por vez y sólo diseñar el software para manejar dicho caso de uso. Si un caso de uso particular no se adecua correctamente a su diseño, se reestructura el diseño hasta lograr que el caso de uso sea implementado de manera razonable. Por lo tanto, los refactorings continuos y agresivos, son un aspecto clave en XP.

2.3 Refactoring de Modelos

Se define *refactoring de modelo* al proceso de reestructurar un modelo orientado a objetos aplicando una secuencia de transformaciones que preservan la funcionalidad del mismo a fin de mejorar algún factor de calidad [9]. Es una propuesta transformacional para el desarrollo de software. Se basa en la idea de introducir cambios en un modelo en pasos pequeños y sistemáticos donde cada paso mejora el modelo de acuerdo a alguna métrica específica. Los refactorings resultan una técnica poderosa cuando son aplicados repetidamente en el modelo.

Este tipo de refactorings define un conjunto de reglas que:

- ✓ Permiten transformar gradualmente y en forma automática tanto una jerarquía de clases como otros elementos de un modelo para lograr mejoras en el mismo
- ✓ Preservan el comportamiento del modelo resultante
- ✓ Garantizan la consistencia del modelo resultante
- ✓ Permiten la intervención del diseñador para aplicar distintas estrategias o alternativas de reestructuración.

La transformación de modelos [16] es un proceso que posee las siguientes características:

- Se basa en la aplicación de reglas para el refactoring de modelos.
- Cada refactoring se realiza sobre un subconjunto del modelo fuente seleccionado por el diseñador.
- Utiliza un conjunto de reglas para la reestructuración de modelos que permiten la transformación gradual y automática garantizando consistencia y equivalencia funcional.
- La aplicación de cada refactoring puede crear nuevos elementos en el modelo, actualizar o eliminar elementos existentes.
- La aplicación de cada refactoring debe producir un modelo destino funcionalmente equivalente al modelo fuente.

Las transformaciones de modelos pueden ser categorizadas según dos dimensiones:

- *Transformación vertical*: ocurre cuando un modelo fuente es transformado en otro modelo destino en diferente nivel de abstracción. Por ejemplo,

cuando se transforma un modelo a código fuente en un lenguaje de programación. En el contexto de MDA, las transformaciones verticales son útiles para transformar un modelo independiente de plataforma (PIM) en un modelo dependiente de plataforma (PSM), y éste en un modelo dependiente de la implementación (ISM).

- *Transformación horizontal*: ocurre cuando un modelo fuente es transformado en otro modelo destino en el mismo nivel de abstracción. Estas transformaciones son realizadas para soportar la evolución de los modelos. Se identifican tres tipos de evolución de modelos:
  - Evolución correctiva: se refiere a la corrección de errores en el diseño.
  - Evolución adaptativa: se refiere a la modificación de un diseño para contemplar cambios en los requerimientos.
  - Evolución perfectiva: se refiere a la modificación de un diseño para mejorar ciertas características del modelo.

La ingeniería forward, dentro del contexto de MDA, consiste en el proceso de transformación de modelos que va desde modelos independientes de la implementación a modelos dependientes de ella. Los refactorings son importantes para reestructurar los modelos generados en cada una de las etapas del proceso. Análogamente, los refactorings tienen la misma importancia en el proceso inverso, es decir, en el proceso de ingeniería reversa, donde se construyen modelos de alto nivel de abstracción a partir de modelos de bajo nivel de abstracción. También resulta una técnica provechosa en los procesos de reingeniería de software donde se parte de modelos de código dependientes de una plataforma específica y se pretende migrarlo hacia otra plataforma construyendo modelos más abstractos durante este proceso.

Los distintos modelos generados en cada una de las etapas de los procesos de desarrollo pueden describirse por medio de un lenguaje de modelado. El estándar propuesto por OMG y más ampliamente usado por la comunidad de desarrollo de software orientado a objetos es UML. En esta tesis se muestran los refactorings aplicados a modelos UML. La aplicación de refactorings sobre estos modelos permite la evolución de los mismos mejorando factores de calidad como extensibilidad, modularidad, reusabilidad, complejidad y legibilidad. Las reestructuraciones de modelos tienen que ver con el agregado, movimiento o eliminación de elementos de un modelo, como por ejemplo clases, atributos y asociaciones, con la incorporación de patrones de diseño [17], y en modelos dependientes de una plataforma específica se aplican refactorings que tratan de mejorar cuestiones vinculadas a dicha plataforma.

La aplicación de refactorings a nivel de modelos tiene las ventajas, además de las expuestas anteriormente, de que las modificaciones son realizadas en etapas tempranas del desarrollo de software y no están sujetas a ningún lenguaje de programación en particular.

2.4 Refactoring en aplicaciones Web para mejorar usabilidad

Como se mencionó anteriormente al comienzo de esta tesis, es de suma importancia poner foco en la usabilidad de una aplicación Web. Existe una interesante investigación de patrones para mejorar la usabilidad en [8], donde se han detectado una serie de factores que contribuyen a la usabilidad de una aplicación Web y que pueden ser logradas mediante la técnica de refactoring:

- *Accesibilidad*: el grado en el que una aplicación Web puede ser utilizada por usuarios con impedimentos de algún tipo como puede ser el físico.
- *Navegabilidad*: calidad en la estructura de navegación para facilitar el acceso al contenido mediante links.
- *Eficiencia*: medida de cuán rápido y preciso es para un usuario acceder a lo que está buscando.
- *Credibilidad*: capacidad de la aplicación de transmitir y establecer una confianza con el usuario.
- *Intuición*: refleja si la organización y el diseño del contenido permite al usuario entender fácilmente qué es lo que la aplicación provee y cómo encontrar aquello que está buscando.
- *Personalización*: habilidad de ofrecer recomendaciones de relevancia para el usuario, basado en su perfil o experiencias anteriores.

Para mantener estos factores presentes en una aplicación a lo largo del tiempo, es importante entonces la aplicación de refactorings [18]. La detección de la carencia en alguno de estos factores, y la atención al feedback del cliente son de suma relevancia para determinar el grado de usabilidad de nuestra aplicación, y el refactoring es una oportunidad excelente para cambiarla y mejorarla. En [13] y [18] se explican algunos refactorings para mejorar la usabilidad, como pueden ser: agregar un link entre dos puntos dentro de la aplicación para acortar la ruta de acceso a un recurso de interés,

permitir el autocompletado de campos de texto para ahorrarle tiempo al usuario, mantener un registro de navegabilidad para que el usuario pueda saber en qué posición se encuentra dentro del árbol de navegación (*breadcrumbs)* y así darle la posibilidad de ir hacia delante o atrás, etc.

Esta tesis se basa principalmente en la aplicación de este tipo de refactorings.

2.5 UWE

UWE (UML-Based Web Engineering) [4] es una propuesta basada en UML y en el proceso unificado para modelar aplicaciones Web. Esta propuesta está formada por una notación para especificar el dominio (basada en UML) y un modelo para llevar a cabo el desarrollo del proceso de modelado [19]. Los sistemas adaptativos y la sistematización son dos aspectos sobre los que se enfoca UWE.

Los principales aspectos en los que se fundamenta UWE son los siguientes:

- Uso de una notación estándar UML para todos los modelos
- Definición de métodos o pasos para la construcción de los diferentes modelos.
- Especificación de restricciones: se recomienda el uso de restricciones escritas (OCL, lenguaje de restricciones de objetos) para aumentar la exactitud de los modelos.

2.5.1 UWE y su relación con UML

UWE define una extensión de UML con perfiles y estereotipos bien definidos, e incluye en su definición tipos, etiquetas de valores y restricciones para las características especificas del diseño Web, las cuales, unidas a las definiciones de UML, forman el conjunto de objetos de modelado que se usarán para el desarrollo del modelo utilizado en UWE.

Las funcionalidades que cubre UWE abarcan áreas relacionadas con la Web como la navegación, presentación, los procesos de negocio y los aspectos de adaptación.

Una de las ventajas de que UWE extienda el estándar UML es la flexibilidad de éste para la definición de un lenguaje de modelado específico para el dominio Web y la aceptación universal de dicho estándar en el campo de la ingeniería del software.

Otra gran ventaja es que actualmente existen múltiples herramientas CASE basadas en UML, con lo cual es relativamente sencilla su utilización y ampliación para utilizar los objetos de modelado definidos en UWE.

2.5.2 Modelos de UWE

UWE hace un uso exclusivo de estándares reconocidos como UML y el lenguaje de especificación de restricciones asociado *Object Constraint Language* (OCL) [20]. Para simplificar la captura de las necesidades de las aplicaciones Web, UWE propone una extensión que se utiliza a lo largo del proceso de modelado [21]. Este proceso está dividido en cuatro pasos o actividades:

Análisis de Requerimientos

Fija los requisitos funcionales de la aplicación Web para reflejarlos en un modelo de casos de uso.

Modelo Conceptual

Materializado en un modelo de dominio, considerando los requisitos reflejados en los casos de uso. Lo componen elementos como Clases, Atributos, Métodos, Asociaciones.

Modelo Navegacional

Este modelo indica cómo el sistema de páginas Web de la aplicación está relacionado internamente. Es decir, cómo es el flujo de navegación. Para ello, se mapean las clases del modelo conceptual en elementos de navegación llamados *nodos,* que representan unidades de información y comportamiento percibido por el usuario, y que se conectan mediante enlaces de navegación [4].

Modelo de Presentación

Este modelo lo componen las vistas de la interfaz de usuario, expresadas mediante modelos estándares de interacción UML. Se constituye principalmente de una colección de páginas con sus respectivos componentes (widgets). Estos widgets pueden tener la función de mostrar atributos de nodos, desencadenar operaciones o actuar de enlace entre nodos. Representa una interfaz abstracta ya que no define la posición exacta de los widgets o la especificación gráfica de ellos, pero sí su tipo [4].

CAPÍTULO 3
Arquitectura Base

En este capítulo veremos cómo extender la funcionalidad de MagicDraw mediante su OpenAPI. Además presentaremos MagicUWE, el plugin para implementar UWE en MagicDraw, que sirvió de base para nuestra herramienta MagicUWE4R.

3.1 Caso de estudio: MagicDraw OpenAPI

¿Qué es MagicDraw?

MagicDraw [10] es una completa aplicación para el modelado en UML de procesos de negocio, arquitectura o software. Diseñado para analistas de negocio, programadores, testers y escritores de documentación, esta herramienta facilita el análisis y diseño de aplicaciones orientadas a objetos.

MagicDraw provee facilidades para los mecanismos de ingeniería de código, modelado de esquemas de base de datos e ingeniería inversa. Esto implica, entre otras cosas, generación automática de código y sincronización entre el código y el modelo

¿Por qué MagicDraw?

Pasemos a listar algunos motivos por los cuales se optó por MagicDraw:

- ✓ Da soporte al perfil de UWE
- ✓ MagicUWE se implementa como una extensión para MagicDraw (detallado en la Sección 3.3)
- ✓ Permite una organización en paquetes según el tipo de modelo (para una mejor organización, respetando la separación de concerns)
- ✓ Portabilidad. Corre sobre la JVM (Java Virtual Machine), por lo que puede operar en gran diversidad de sistemas operativos (Windows, Linux, MacOS)

- ✓ Deriva código a partir de un modelo ya existente
- ✓ Deriva modelos a partir de código ya existente
- ✓ Facilita la semi-generación de aplicaciones Web
- ✓ Fácil de usar

Nos debe quedar claro que lo más importante es que, en un entorno Model Driven, MagicDraw es excelente ya que, mediante su OpenAPI [22], permite extender la manipulación de modelos y diagramas de forma sencilla.

3.2 Extensibilidad: Plugins

Como mencionamos recientemente, la forma en que MagicDraw nos ofrece extensibilidad es mediante su OpenAPI. Empleando esta interfaz de programación de aplicaciones (API), la única forma de modificar la funcionalidad existente es a través de Plugins. El objetivo principal de la arquitectura del Plugin es agregar nueva funcionalidad a MagicDraw.

Un Plugin debe contener los siguientes recursos:

- Directorio
- Archivos Java compilados y empaquetados en un JAR
- Archivo descriptor del Plugin
- Archivos externos, opcionales, a utilizar por el plugin

Generalmente, un Plugin crea componentes gráficos en la aplicación, de manera que el usuario pueda acceder mediante GUI a la funcionalidad que el Plugin ofrece. Sin embargo, esto no es estrictamente necesario ya que el Plugin puede escuchar, o actuar de listener ante cambios en el proyecto.

Funcionamiento

MagicDraw, en cada arranque de la aplicación, escanea el directorio correspondiente a los plugins, buscando en sus subdirectorios lo siguiente:

1. Si el subdirectorio contiene un archivo descriptor, el Plugin Manager lo lee.
   Ejemplo de descriptor:

```
<plugin  id="MagicUWE4R" name="Miguel Djebaile Thesis" version="1.0"
         provider-name="Facultad de Informática - Universidad Nacional de La Plata"
         class="magicUWE.core.PluginManager">
         <requires>        <api version="1.0"/>        </requires>
         <runtime>          <library name="MagicUWE4R.jar"/>          </runtime>
</plugin>
```

2. Si los requerimientos declarados en el archivo descriptor son los necesarios, el Plugin Manager carga la clase especificada. Esta clase debe extender la clase com.nomagic.magicdraw.plugins.Plugin.

3. Se invoca al método init() de la clase especificada. A través de este método es cómo podemos agregar componentes GUI, y utilizando la arquitectura de Actions podemos realizar las operaciones que queramos. Es importante aclarar que init() sólo es llamado si el método isSupported() devuelve true. En la Tabla 1 podemos observar cómo estos métodos son definidos abstractos en la superclase, por lo cual estamos obligados a implementarlos.

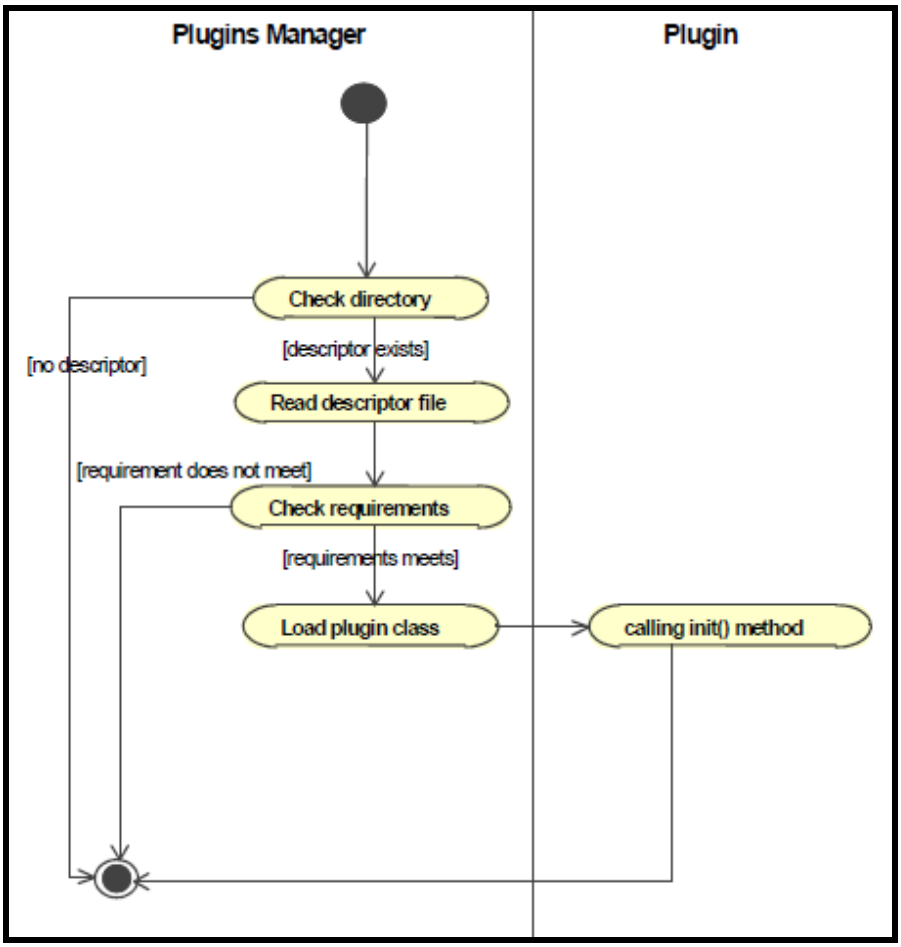


Figura 4. Diagrama de estados correspondiente a la carga de plugins en el arranque de MagicDraw.

4. Cuando se desea cerrar MagicDraw, el método *close()* se ejecuta. Si este devuelve false, el cierre se cancela.

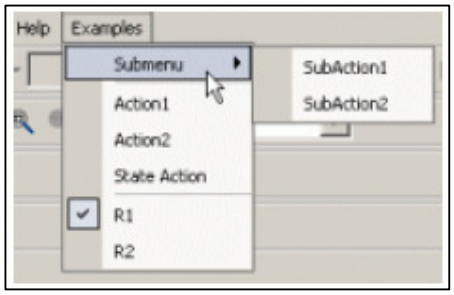
Method Summary	
abstract boolean	close() MagicDraw llama a este método antes de cerrar la aplicación.
PluginDescriptor	getDescriptor() Devuelve el descriptor del plugin.
abstract void	init() Método de inicialización del plugin.
abstract boolean	isSupported() MagicDraw llama a este método para saber si el plugin está habilitado.
(package private) void	setDescriptor(PluginDescriptor descriptor) Setea el descriptor.

Tabla 1. Métodos de la clase Plugin

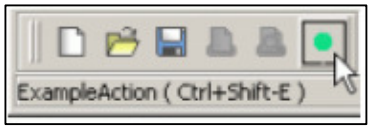
3.2.1 Actions

Pasemos a hablar a nivel de implementación del plugin. Mediante el código existe un mecanismo, con una estructura y conjunto de clases asociadas, que nos permite agregar la nueva funcionalidad. Este es el mecanismo de Actions. Una vez implementado, la manera de invocarlo es mediante GUI, y puede hacerse desde diversos sectores:

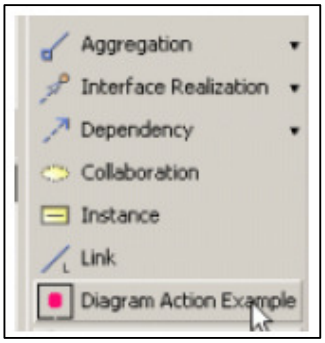
Menú principal ⟶



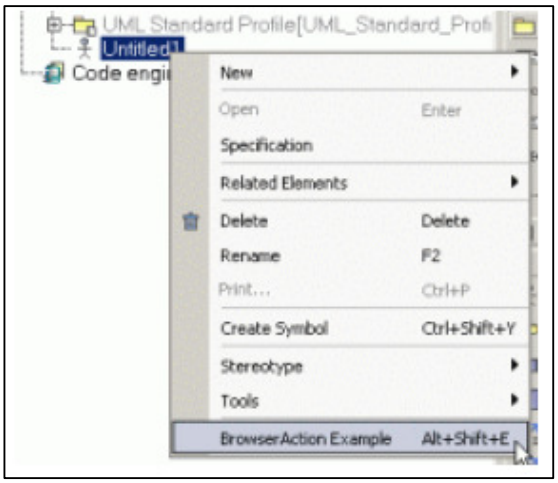
Barra de Menú ⟶



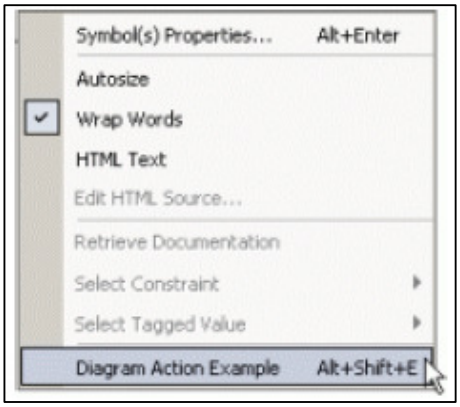
Barra de Diagrama ⟶



Menú del browser ⟶



Menú contextual del diagrama ⟶



En el caso de MagicUWE4R, la herramienta de refactoring desarrollada en esta tesis, se hizo uso del menú contextual del diagrama para ejecutar gran parte de la funcionalidad.

Creación de Actions

Existe una jerarquía de clases de actions disponibles y son utilizadas por propósitos particulares. Nosotros podemos heredar de estas clases según nuestras necesidades. Por ejemplo:

- ***DefaultBrowserAction***: permite acceder al árbol de componentes del browser. Recomendado para la ejecución de actions al seleccionar nodos de él.
- ***DefaultDiagramAction***: para acceder a los elementos del diagrama. Recomendado para ejecutar actions con los elementos seleccionados en el diagrama
- ***PropertyAction***: action para cambiar propiedades de elementos o del proyecto.
- ***NMStateAction***: action que también posee un estado a ser manipulado.
- ***DrawPathDiagramAction***: se debe heredar de esta action cuando se requiera dibujar un link entre dos elementos del diagrama
- ***DrawShapeDiagramAction***: nos permite dibujar una figura al ser ejecutada la action.

Este tipo de clases nos facilitan agregar la nueva funcionalidad que deseamos de manera muy sencilla, ya que establece un marco de desarrollo bien definido. Por ejemplo, si se desea extender de NMStateAction, sabemos que debemos sobrescribir el método actionPerformed(), o que para el caso de un DrawShapeDiagramAction, el método que se ejecuta cuando la action es disparada es createElement().

Categoría de actions

Cada action debe estar dentro de una ActionsCategory. Una ActionsCategory o categoría de actions comprende un pequeño grupo de actions. Puede verse representada gráficamente como un separador o un submenú.

Las categorías son agregadas dentro del ActionsManager, quien actúa como una especie de contenedor de actions. Visualmente puede ser tanto una barra de menú, menú contextual o barra de herramienta.

En la Tabla 2 podemos ver cómo se mapean estas clases MagicDraw en elementos GUI:

	ActionsManager	**ActionCategory**	**Action**
Menú	Barra de menú	Menú	Ítem del menú
Barra de herramientas	Todas las barras	Una barra	Botón
Menú contextual	Menú contextual	Submenú	Ítem del menú

Tabla 2. Relación entre clases y elementos GUI

3.2.2 Configurators

Actions dentro de un ActionsManager son configuradas por Configurators. Un Configurator es responsable de aspectos como agregar o quitar actions dentro de un lugar determinado o posicionarlo dentro del mismo.

Existen tres tipos de configurators:

- *AMConfigurator*: de propósito general. Empleado en menús, barras de herramientas, atajos de teclado.
- *BrowserContextAMConfigurator*: para la configuración de managers que contengan actions a ser mostradas en el menú contextual del browser
- *DiagramContextAMConfigurator*: configura actions para el menú contextual (pop up) dentro del diagrama. Puede acceder al diagrama, elementos seleccionados en él.

Vale aclarar que los ActionsManager para el menú principal y todas las barras son creados y configurados sólo una vez, de manera que luego de esto únicamente es posible deshabilitarlos pero no eliminarlos. En cambio, los menús contextuales son creados en cada invocación, por lo que los ActionsManager son creados y configurados en cada ocasión, y las actions pueden agregarse y eliminarse dinámicamente.

Es importante esto último, ya que en MagicUWE4R se hace uso del DiagramContextConfigurator por dos motivos principales: las actions de refactoring son ejecutadas mediante el menú contextual, y es necesario contar con la dinámica de poder hacer visible ciertas actions en determinadas circunstancias, y ocultas ciertas otras ante diferentes situaciones. Por ejemplo, es deseable contar con un conjunto de actions de refactoring en el modelo de navegación, y es necesario que al pasar al modelo de presentación, otras actions sean mostradas, ocultando las primeras.

Por último, todos los Configurators son registrados dentro de un ActionsConfiguratorManager, quien actúa como un contenedor de configurators.

3.3 MagicUWE

Hasta ahora, en el presente capítulo, hemos visto cuál es el camino a seguir a la hora de extender MagicDraw. Lo que no hemos dicho es que MagicUWE4R, nuestro plugin de refactoring, es de hecho una extensión de otro plugin ya creado, denominado MagicUWE [12].

Como dijimos anteriormente, la metodología elegida para el modelado de aplicaciones Web fue UWE. UWE emprendió y construyó el proyecto MagicUWE, un plugin que da soporte al perfil de UWE sobre la herramienta UML, MagicDraw.

Usabilidad, Adaptabilidad y Extensibilidad

MagicUWE soporta la anotación propia de UWE y todos sus procesos de desarrollo. Provee extensiones en la barra de herramientas para el uso confortable de los elementos de UWE, incluyendo atajos de teclado para alguno de ellos. También ofrece un menú específico para la creación de nuevos paquetes y nuevos diagramas para las diferentes vistas o concerns de la aplicación (conceptual, navegación, presentación).

El objetivo de MagicUWE es que el diseño de aplicaciones Web con UWE y MagicDraw sea un proceso simple. Para lograrlo, fue esencial una interfaz de usuario intuitiva, como también los mensajes de advertencia para mantener los modelos válidos (por ejemplo, si se están dibujando elementos de presentación en el diagrama navegacional).

La adaptabilidad es también un objetivo importante de MagicUWE. Los mensajes con *tips* o consejos de MagicUWE pueden ser configurados (o desactivados) para cada tipo de tipo de diagrama de UWE por separado.

MagicUWE tiene un diseño altamente modular. El código está muy bien documentado, y posee un script potente para crear e instalar nuevas versiones del plugin de manera automática.

En conclusión, MagicUWE4R fue concebido a partir de las facilidades de extensión y adaptabilidad de MagicUWE, así logrando agregar capacidades de refactoring al plugin.

CAPÍTULO 4
Arquitectura de MagicUWE4R

En esta sección veremos la arquitectura de MagicUWE4R y cómo establece un framework para futuros refactorings a desarrollar. Ahondaremos en la implementación de este plugin y veremos un ejemplo de cómo agregar un nuevo refactoring utilizando las facilidades que ofrece.

4.1 Arquitectura

La arquitectura de MagicUWE4R fue rediseñada en varias ocasiones, de manera de lograr en cada paso, una mayor flexibilidad y adaptabilidad.

Gran parte de este resultado fue producto de un diseño orientado a patrones, que será explicado en detalle en el presente capítulo. Como anticipo, lo que se hizo fue encapsular cada tipo de refactoring en un objeto Action. Este mecanismo de actions tiene relación con el patrón Command [17]. A su vez, se diagramó una jerarquía de clases de refactorings que, mediante un Template Method [17]. permitió la composición de los mismos, con un correcto uso de la herencia, encapsulamiento y polimorfismo. Por último, encontramos que el patrón Visitor [17] está muy bien implementado por la API de MagicDraw [23], por lo que nos permitió ahorrarnos mucho esfuerzo, y de la manera más elegante.

En la Figura 5 se observa el diagrama de clases de los refactorings a alto nivel

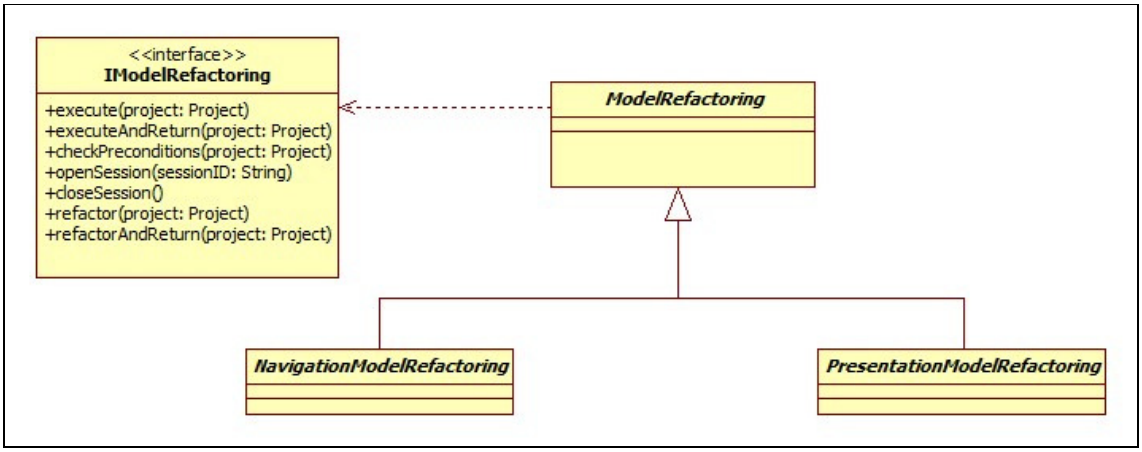


Figura 5. Diagrama de clases de refactorings a alto nivel

Podemos observar que existe una interfaz IModelRefactoring la cual es implementada por la clase abstracta ModelRefactoring. NavigationModelRefactoring y PresentationModelRefactoring heredan de esta última. Estas dos subclases serán superclases de los refactorings (Figuras 6 y 7)

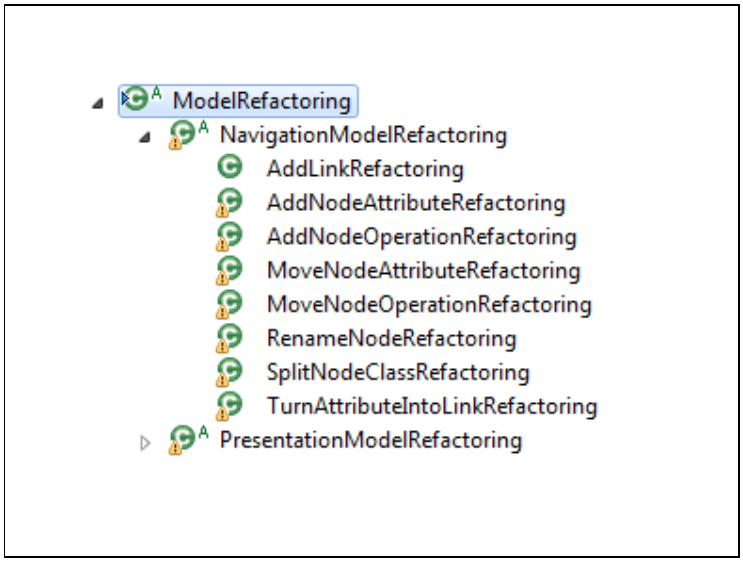


Figura 6. Jerarquía de refactorings del modelo de navegación

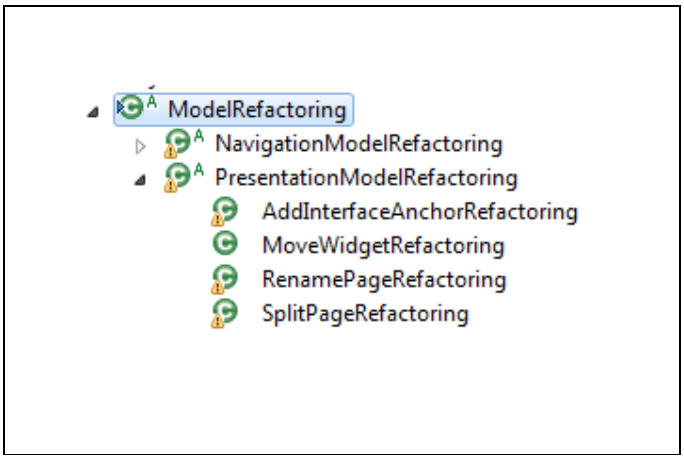


Figura 7. Jerarquía de refactorings del modelo de presentación

Pasemos a analizar detalladamente cada uno de los puntos en los cuales se enfocó el diseño, y cuáles son los beneficios en cada caso.

4.2 Composición de refactorings en el código

Como se mencionó en el primer capítulo, uno de los aspectos en los que se hizo hincapié fue en desarrollar los refactorings en unidades atómicas, de manera que puedan componerse, para que el motor de refactoring sea extensible a otros nuevos refactorings.

Esta es una de las principales ventajas del refactoring, dado que a partir de pequeños cambios, que son más seguros, se pueden llegar a realizar reestructuraciones complejas [24]. Es decir, poder tener la posibilidad de crear un nuevo refactoring más complejo, a partir de la composición de refactorings ya existentes [13].

Veamos un ejemplo para clarificar el concepto: un refactoring simple y a su vez muy empleado en el modelo de navegación de UWE es el llamado *Move Node Operation*. Básicamente lo que hace es mover una operación desde un nodo origen hacia un nodo destino. Otro refactoring sencillo es el *Move Node Attribute*, que realiza lo mismo que el anterior, pero moviendo un atributo en lugar de una operación. En cambio, existe un refactoring más complejo, *Split Node Class*, que se aboca a desacoplar un nodo repleto de información o saturado de funcionalidades ajena a él. En el capítulo 5 lo veremos detalladamente, pero podemos decir que este refactoring sigue los siguientes pasos:

1. Agregar una nueva clase de nodo vacía
2. Para cada atributo que se decida, mover desde la clase del nodo origen a la clase del nuevo nodo
3. Para cada operación que se decida, mover desde la clase del nodo origen a la clase del nuevo nodo
4. Agregar un link bidireccional entre el nodo origen y el nuevo nodo para permitir al usuario poder acceder a la información original y su conjunto de operaciones.
5. En caso opcional, renombrar el nodo origen

Claramente podemos ver que en el punto 2 y 3 es posible hacer uso de los refactorings explicados anteriormente (específicamente para el punto 2 se puede utilizar *Move Node Attribute* mientras que para el punto 3, el *Move Node Operation*). A su vez, para el punto 4 también existe un refactoring que puede reutilizarse para

ejecutar dicha función, el refactoring *Add Link*. Por último, el punto 5 se apoya en el refactoring atómico *Rename Node* [25].

Podemos ver entonces el gran potencial que tiene este enfoque. Nos permite introducir conceptos tan importantes como la extensibilidad, flexibilidad y escalabilidad, a analizar más adelante.

4.3 Diseño extensible basado en patrones

El concepto de *patrón de diseño* instalado en Ingeniería del Software se ha tomado prestado de la Arquitectura. La creación del concepto se atribuye a un profesor de Arquitectura de la Universidad de Berkeley llamado Christopher Alexander. En [26] menciona que "cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo siquiera dos veces de la misma forma". No fue hasta principios de la década de 1990 cuando los patrones de diseño tuvieron un gran éxito en el mundo de la informática a partir de la publicación de [17], en el que se presenta un catálogo de 23 patrones de diseño. El patrón es un esquema de solución que se aplica a un conjunto de problemas semejantes. Esta aplicación del patrón no es mecánica, sino que requiere de adaptación. Facilitan la reusabilidad, extensibilidad y mantenimiento. La clave es anticiparse a los nuevos requisitos y cambios, de modo que los sistemas evolucionen de forma adecuada. Cada patrón permite que algunos aspectos de la estructura del sistema puedan cambiar independientemente de otros.

Algunas ventajas del uso de patrones de diseño son:

- ✓ Proporcionar catálogos de modelos reusables en el diseño de 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.

Veamos ahora cuáles fueron los patrones de diseño empleados, cómo fueron empleados y con qué objetivos.

4.3.1 El Template Method como generador de un framework

El Template Method es un patrón de diseño de comportamiento, cuyo propósito es definir el esqueleto de un algoritmo en una operación, dejando la posibilidad de redefinir algunos pasos en las subclases, sin cambiar la estructura del método [17].

Veamos la estructura que define el Template Method:

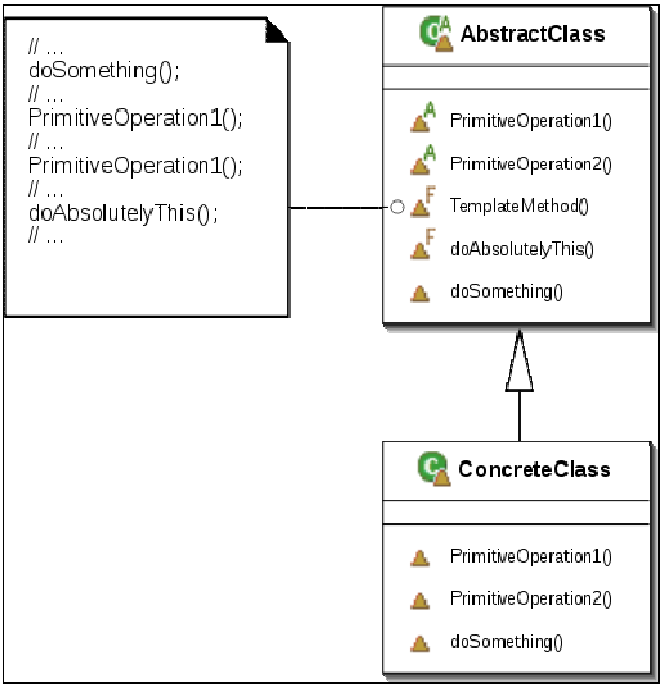


Figura 8. Estructura del patrón de diseño Template Method

Es decir, se compone de una clase abstracta, quien:

- Define las operaciones primitivas abstractas que suponen las partes variantes, obligando a las subclases a implementarlas.
- Implementa métodos template definiendo el esqueleto de un algoritmo. Los métodos pueden llamar tanto a las operaciones primitivas como a cualquier otro tipo de operación.

A su vez, se compone de una clase concreta, cuya función es implementar las operaciones primitivas para determinar el comportamiento específico de la clase concreta.

En cuanto a la aplicabilidad, el Template Method se usa para:

- Implementar partes invariantes de un algoritmo, y dejar a las subclases codificar el comportamiento que puede variar.
- Cuando tenemos un comportamiento común en diferentes clases en donde las diferencias vienen dadas por la naturaleza de cada clase, se refactoriza en un Template Method para evitar la duplicación de código e introducir nuevos cambios de manera flexible.

Para controlar la redefinición de operaciones en las subclases, es posible definir métodos *hook*. Estos generalmente son métodos default en la superclase (implementación predeterminada o vacía en muchos casos), pero pueden ser redefinidos en las subclases. Por ejemplo:

Usualmente, se hace que una subclase herede el comportamiento de la superclase redefiniendo la operación y luego llamando a la operación del padre explícitamente:

```
class Subclass extends Superclass {
    ...
    void something() {
       // código que extiende funcionalidad
       super.something ();
       // código que extiende funcionalidad    }
}
```

Desafortunadamente, es fácil olvidar invocar el método de la superclase, aparte de estar obligados a conocer el código del padre.

En lugar de sobrescribir, es posible agregar *métodos hook*. Así, en las subclases sólo nos preocupamos por implementar los hooks, sin importarnos del método que lo contiene:

```
class Superclass {
    protected void preSomethingHook(){}
    protected void postSomethingHook(){}
    void something() {
        preSomethingHook();
        // alguna implementación
        postSomethingHook();
    }
}
class Subclass extends Superclass {
    protected void preSomethingHook() {
        // código customizado
    }
    protected void postSomethingHook() {
        // código customizado
    }
}
```

Template Method en MagicUWE4R

Veamos ahora cómo se aplica esto a MagicUWE4R. Tomando como ejemplo algunos de los refactorings del modelo de navegación, tenemos el siguiente diseño:

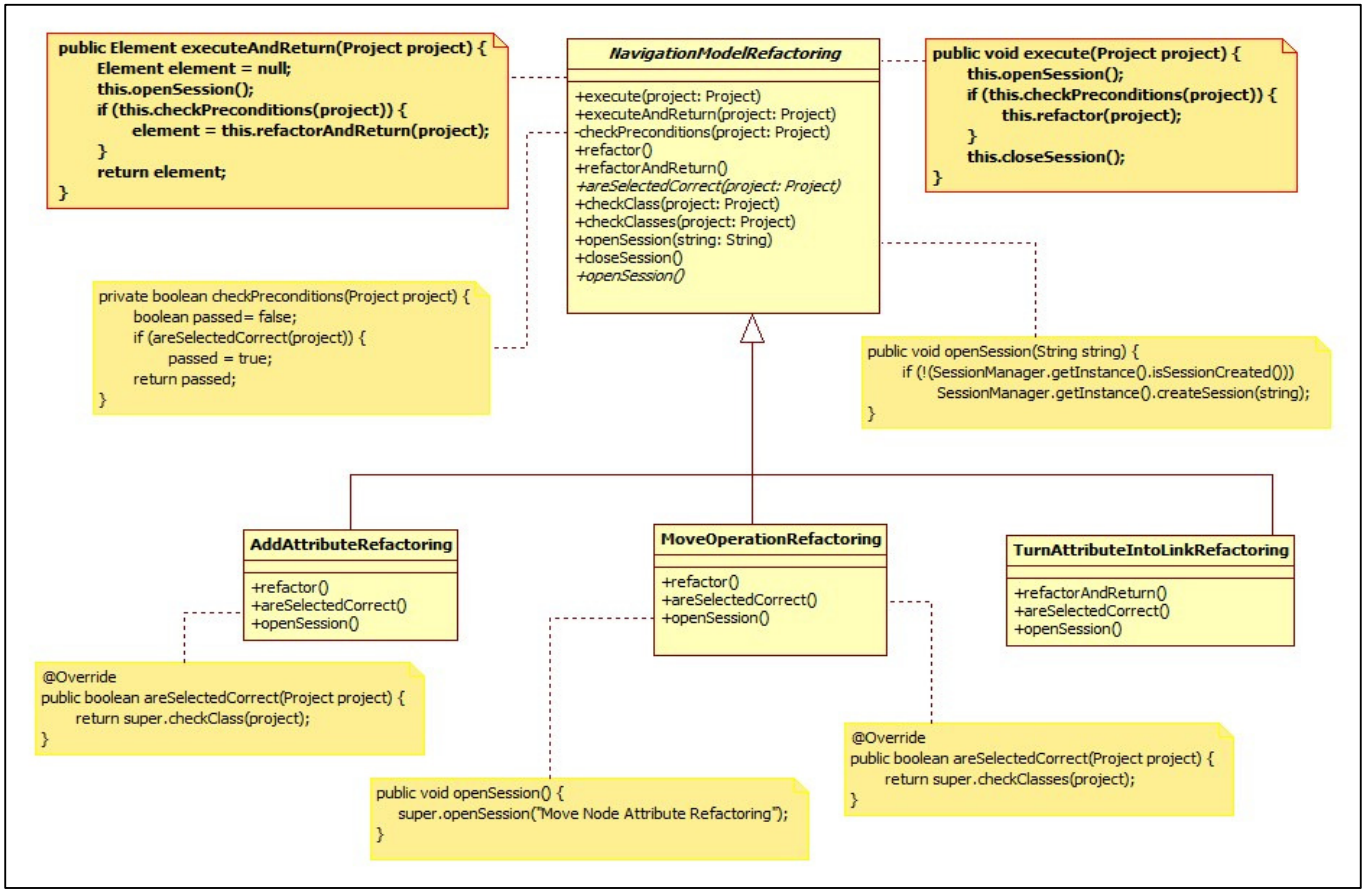


Figura 9. Template Method en MagicUEW4R

Si hacemos un mapeo con la estructura que define el Template Method, nuestra clase abstracta es NavigationModelRefactoring, que implementa 2 métodos template, *execute()* y *executeAndReturn()*, dependiendo si el refactoring requiere de la intervención del usuario para finalizar o no.

Básicamente, un refactoring se comprende de las siguientes operaciones:

1. Se abre la sesión (todas las modificaciones al modelo deben hacerse dentro del contexto de una sesión)
2. Se chequean las precondiciones (condiciones que deben ser cumplidas para que el refactoring sea efectivo)
3. Si las precondiciones son cumplidas, se hace el refactoring
4. Se cierra la sesión.

Y esta es la estructura del Template Method *execute()*.

Como se puede ver en las notas de la Figura 9, para abrir la sesión, es necesario extender el método en las subclases ya que son dependientes de cada una de ellas. El chequeo de las precondiciones hace uso de métodos hook, ya que tenemos sección de código común, y para las partes variantes se hace uso del método hook *areSelectedCorrect()*. El refactoring en sí claramente es implementado en las subclases. Y el cierre de sesión es común a todas (definido en la superclase)

Ahora, por qué en el subtítulo dice “El Template Method como generador de un framework”? Porque el diseño que se acaba de explicar ofrece una estructura conceptual y de soporte ya definida, con módulos concretos de base, que hacen la tarea de agregar un nuevo refactoring un paso muy simple y sencillo. Por ejemplo, y hablando desde el modelo de dominio, si queremos crear un nuevo refactoring de presentación, sólo tenemos que crear una clase que extienda PresentationModelRefatoring, implementar los métodos necesarios y hacer uso de otros ya definidos. Rápido, sencillo, flexible, modular y escalable.

4.3.2 Ejecutando actions: patrón Command

El Command es también un patrón de comportamiento. Su propósito es encapsular un mensaje como un objeto, con lo que permite gestionar colas o registro de mensajes (por ejemplo para hacer undo - o deshacer - de las operaciones) [17]. Ofrece una interfaz común que permite invocar las acciones de forma uniforme y extender el sistema con nuevas operaciones de forma más sencilla. De gran utilidad cuando se necesita poder enviar solicitudes a objetos sin tener conocimiento de la operación solicitada ni del receptor de la solicitud.

Pensemos por ejemplo en la opción 'Copiar' del procesador de textos Microsoft Word. Esta acción puede ser ejecutada accediendo tanto a un botón de la barra de herramientas (), como al menú contextual (Clic derecho > Copiar), o como al menú principal (Edición > Copiar). Cada una de estas acciones podría ser encapsulada en un objeto.

Podremos llegar a la conclusión que, aun siendo objetos distintos, realizan la misma funcionalidad. En otras palabras, estos objetos invocan la misma acción.

Siguiendo la conclusión anterior y analizando ahora la funcionalidad de rehacer o deshacer que implementa el editor, nos damos cuenta que estas funciones se basan en acciones, sin importar el objeto que las realizó. Veamos entonces, desde la perspectiva de un diseñador de software, cómo implementar lo anteriormente visto: podemos pensar que cada acción esté implementada en las múltiples clases que las invocan (botón o menú), pero esto finalmente sería una mala práctica ya que haría dependiente a las acciones de cada clase y terminaríamos escribiendo acciones iguales (copiar el texto) en elementos distintos (botón o menú).

Sería mejor si lográramos independizar las acciones de las clases que las invocan, y de esta manera podemos conseguir que:

- Un objeto pueda desencadenar una acción
- Múltiples objetos puedan desencadenar la misma acción
- Un objeto pueda encapsular múltiples acciones

Tal como refleja la Figura 10, el problema anterior lo resolvemos implementando una interfaz (Command) común para todas las acciones. Una clase que implementa una interfaz es independiente al objeto que la utiliza.

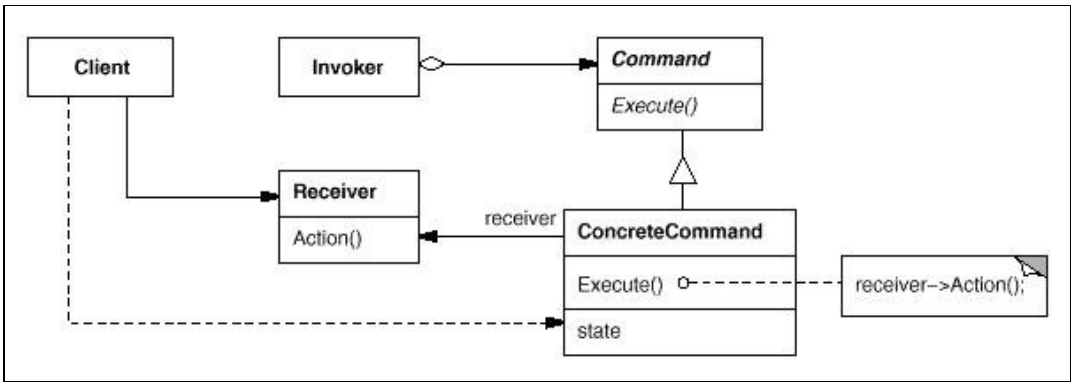


Figura 10. Estructura del patrón de diseño Command

El hecho de poder encapsular las acciones en objetos (ConcreteCommand), nos da la ventaja que las acciones puedan implementar herencia o formar grupos de objetos que realicen un conjunto de acciones comunes.

Otro de los problemas resuelto con este diseño son las operaciones de rehacer o deshacer, ya que podemos colocar los objetos Command en una estructura de datos que nos permita mantener una historia de las acciones realizadas.

Consideraciones

- ✓ Independizar la parte de la aplicación que invoca las acciones de la implementación de las mismas.
- ✓ Tratar a los comandos como objetos.
- ✓ Proporcionar una interfaz común para todas las acciones.

La arquitectura de Actions en MagicUWE4R

Hablando ahora particularmente de MagicUWE4R, la arquitectura de Actions representa una implementación del patrón Command. Así se logra separar la lógica del controlador de la representación visual. Esto resulta muy útil ya que fácilmente podemos configurar el uso de diferentes elementos gráficos para la misma lógica, sin tener que copiar y pegar el código, como también la invocación o ejecución de la acción será independiente del elemento grafico que la represente.

Pasemos a analizar a nivel de implementación: algunos componentes gráficos Java como JButton o JTextField permiten que nos "suscribamos" a eventos que pasan en ellos, de forma que cuando ocurre este evento, el componente nos avisa. Por ejemplo, podemos estar interesados en cuando se pulsa un botón, cuando un componente gana el foco, cuando se pasa el mouse por encima, cuando se cierra una ventana, etc.

Para enterarnos de todos estos eventos, los componentes tienen métodos del estilo add...Listener() donde los puntos suspensivos, de alguna forma, representan el nombre del tipo de evento. Así, por ejemplo, los componentes pueden tener métodos addActionListener(), addMouseListener(), addWindowListener(), etc.

Nosotros nos centraremos en el ActionListener.

Cuando usamos el addActionListener() de un componente, nos estamos suscribiendo a la "acción típica" o que Java considera más importante para ese componente. Por ejemplo, la acción más típica de un JButton es pulsarlo. Para

un JTextField, se considera que es pulsar la tecla *Enter* indicando que hemos terminado de escribir el texto, para un JComboBox es seleccionar una opción, etc.

Entonces cuando llamamos al addActionListener() de cualquiera de estos componentes, nos estamos suscribiendo a la acción más típica de ese componente.

Como parámetro, debemos pasar una clase que implemente la interfaz ActionListener. Por tanto, debemos hacer una clase que implemente ActionListener.

Esta implementación es una clara aplicación de patrón Observer [17]. Logra desacoplar la dependencia entre el objeto componente (el *observado*) y el listener que espera a ser notificado de un evento. De esta manera, se produce una inversión de control (IoC) [27]: el observador que tenía la tarea de estar pendiente del cambio, ahora está a la espera de que el observado le avise, aumentando la modularidad del lenguaje, y evitando bucles de actualización (espera activa o polling).

Veamos la Figura 11 para observar la estrategia planteada en MagicUWE4R:

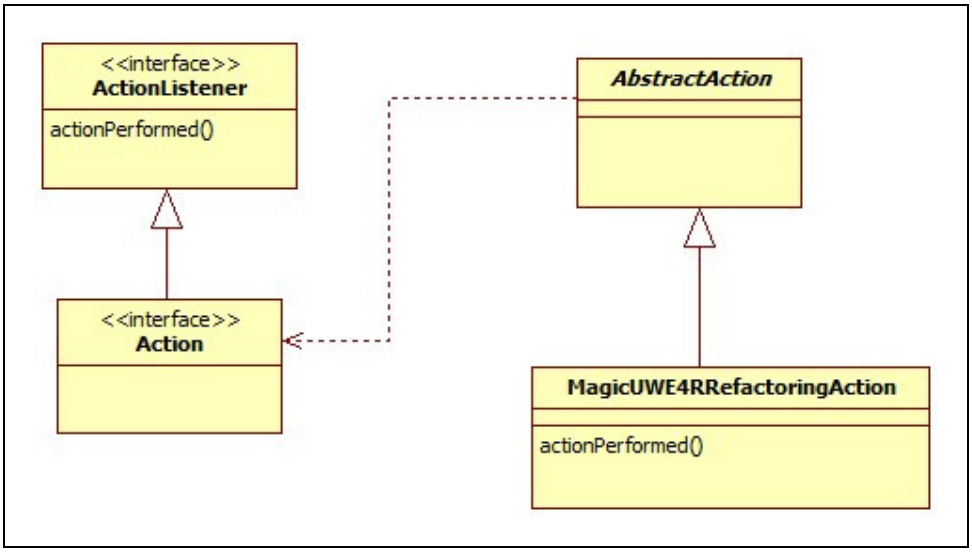


Figura 11. Patrón Command en MagicUWE4R

- ***ActionListener***: Interfaz que actúa como listener para la recepción de eventos de acciones. La clase que tenga interés en procesar este tipo de eventos, implementa esta interfaz, y codifica el método actionPerformed() a ser disparado en la ocurrencia del evento. El objeto que quiera hacer uso de esta implementación, se registra a través del método addActionListener(anActionListener).

- ***Action***: Esta interfaz extiende ActionListener y representa una abstracción de un comando, sin tener explícitamente un componente grafico asociado
- ***AbstractAction***: Clase abstracta que provee una implementación default de Action, definiendo comportamiento estándar como pueden ser los setters y getters de las propiedades del objeto (icono, texto, habilitado). El desarrollador sólo necesita heredar de esta clase y definir el método actionPerformed()
- **MagicUWE4RRefactoringAction**: Clase definida por nosotros que hereda de AbstractAction. Por lo tanto, debe implementar el método actionPerformed(). En él, se describirá el comportamiento del componente ante la ocurrencia de un evento (cuando la acción es disparada). En este caso aquí hacemos la llamada al modelo que realiza las operaciones de refactoring

Ejemplo de flujo de ejecución

Cuando el usuario pulsa un ítem del menú de refactoring:

1. El ítem sobre el que se produce una acción genera un evento ActionEvent
2. En caso que el objeto tenga registrado un listener -de la forma menuItem.addActionListener(ActionListener actionListener)-, ActionEvent invoca al método actionPerformed del listener .
3. Como el listener debe implementar la interfaz ActionListener, el método actionPerformed(ActionEvent e) es invocado de manera que se realiza el procesamiento deseado para el manejo de la acción de refactoring disparada.

Pudimos notar como este patrón ofrece de forma simple una manera de implementar un sistema basado en operaciones que permita su extensibilidad y mantenimiento.

4.3.3 Accediendo a los elementos a través del Visitor

El patrón Visitor es otro de los patrones de comportamiento. Su intención es principalmente proporcionar una forma fácil y sostenible de ejecutar acciones en una familia de clases. Este patrón centraliza los comportamientos y permite que sean modificados o ampliados sin cambiar las clases sobre las que actúan. Representa una forma de separar el algoritmo de la estructura de un objeto [17].

Veamos el diagrama de clases en la Figura 12:

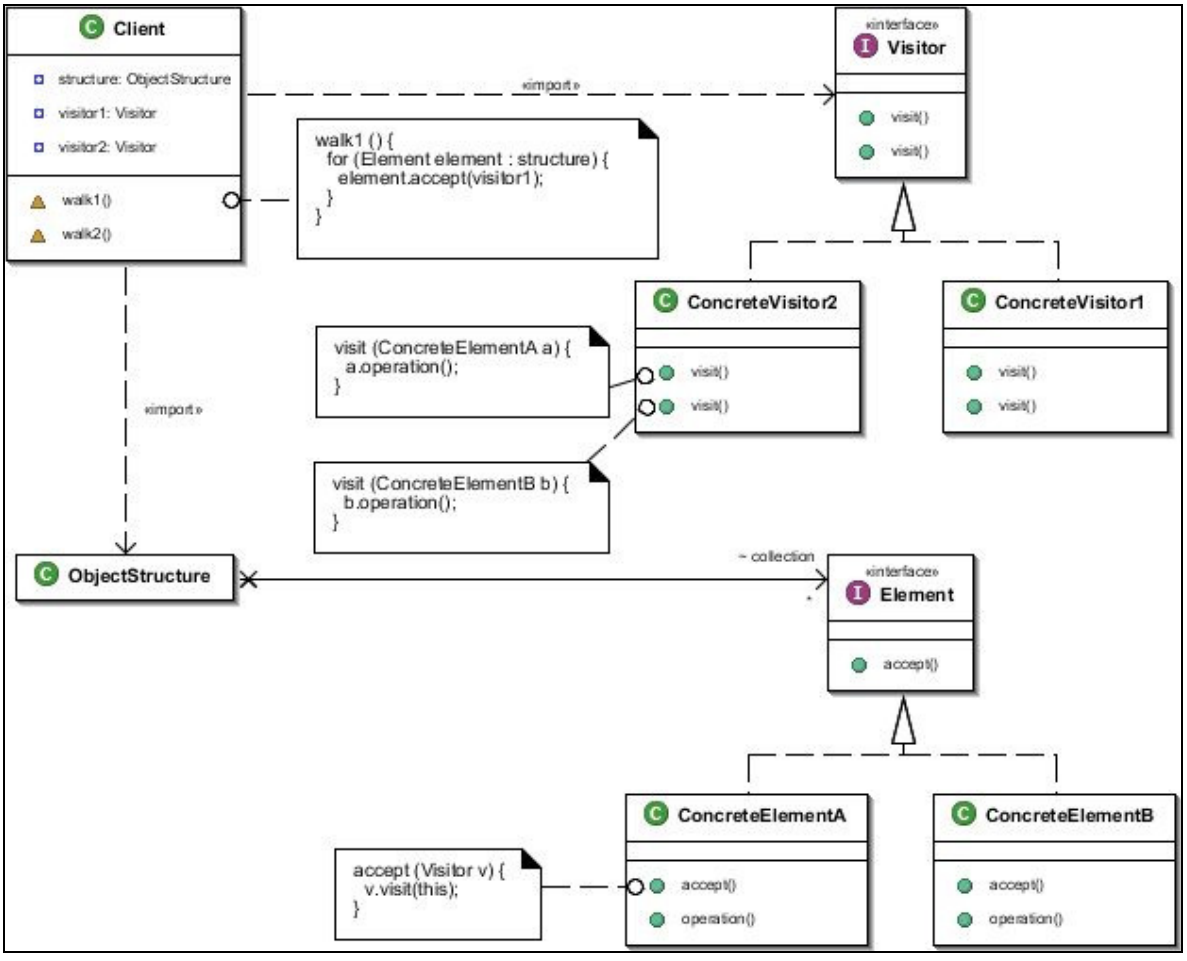


Figura 12. Diagrama de clases del patrón Visitor

La idea básica es que se tiene un conjunto de clases Element que conforman la estructura de un objeto. Cada una de estas clases Element tiene un método *accept()* que recibe al objeto visitador -Visitor- como argumento. El visitador es una interfaz que

tiene un método *visit()* diferente para cada clase elemento; por tanto habrá implementaciones de la interfaz Visitor de la forma: VisitorClase1, VisitorClase2 ... VisitorClaseN. El método *accept()* de una clase Element llama al método *visit()* de su clase. Clases concretas de un visitador pueden entonces ser escritas para hacer una operación en particular.

A su vez, cada método *visit()* de un visitador concreto puede invocar métodos del elemento concreto -pasándose como parámetro- por lo que *visit(ConcreteElement e)* puede ser pensado como un método que no es de una sola clase, sino de un par de clases: el visitador concreto y clase elemento particular.

De esta manera, el patrón Visitor simula el Doble Dispatching (mecanismo que abstrae el envío de mensajes a diferentes clases concretas dependiendo de qué tipo son en tiempo de ejecución) en un lenguaje convencional orientado a objetos de Single Dispatch, como lo es Java.

A modo de ejemplo, el Visitor es empleado cuando tenemos una colección sobre la cual debemos iterar de manera uniforme, donde cada algoritmo necesita iterar de la misma forma, y donde el método *accept()* de un elemento contenedor, además de una llamada al método *visit()* del objeto Visitor, también pasa el objeto Visitor como argumento al llamar al método *accept()* de todos sus elementos hijos.

Este patrón es ampliamente utilizado en intérpretes, compiladores, procesadores de lenguajes, y sobre todo en herramientas de refactoring de código.

Visitor en MagicDraw OpenAPI

El Visitor está muy bien implementando en la Open API de MagicDraw y ahorra gran esfuerzo de manera eficiente y elegante, por lo que es importante usarlo.

MagicDraw posee la clase com.nomagic.magicdraw.uml.Visitor, cuya motivación es visitar de alguna manera particular cada subclase de Element. Esta clase por supuesto se basa en el patrón Visitor.

Veamos el conjunto de métodos que define.

Method Summary	
void	visitBaseElement(BaseElement o) Method visits given object.
void	visitDiagramPresentationElement(DiagramPresentationElement o) Method visits given object.
void	visitPathConnector(PathConnector o) Method visits given object.
void	visitPathElement(PathElement o) Method visits given object.
void	visitPresentationElement(PresentationElement o) Method visits given object.
void	visitProject(Project o) Method visits given object.
void	visitShapeElement(ShapeElement o) Method visits given object.

Tabla 3. Métodos de la Clase Visitor

A su vez, encontramos en las clases como BaseElement, DiagramPresentationElement, PathConnector, PathElement, ShapeElement el método accept():

```
public void accept(Visitor visitor)throws Exception
```

4.4 Extensión de MagicUWE4R.

Como dijimos anteriormente, nuestro plugin fue desarrollado en Java. Fue compilado con la JDK 1.6 [28] y se empleó como IDE de desarrollo Eclipse [29], por lo que las figuras o citas del código fuente se remitirán a dicho ambiente.

En la estructura del proyecto Java, encontramos una serie de 'hot-spots' (sectores del framework donde ocurre la adaptabilidad o extensibilidad según sea el caso. Figura 13):

- Clase **magicUWE.core.PluginManager.java**: representa el Plugin Manager. Responsable de inicializar el plugin -a través del método init()- y de agregar los Configurators necesarios.

- Clase **magicUWE.core.PluginManagerActions.java**: responsable de ensamblar los Configurators con sus respectivos Actions.

- Paquete **magicUWE.actions.refactoring**: aquí se definen los Actions que representarán las ejecuciones de refactoring.

- Paquete **magicUWE.configurators.context.refactoring**: aquí se encuentran los Configurators que contendrán los Actions.

- Paquete **magicUWE.core.model**: en este paquete irán las implementaciones de los refactorings. Aquí se implementa el Template Method visto en la sección 4.3.1

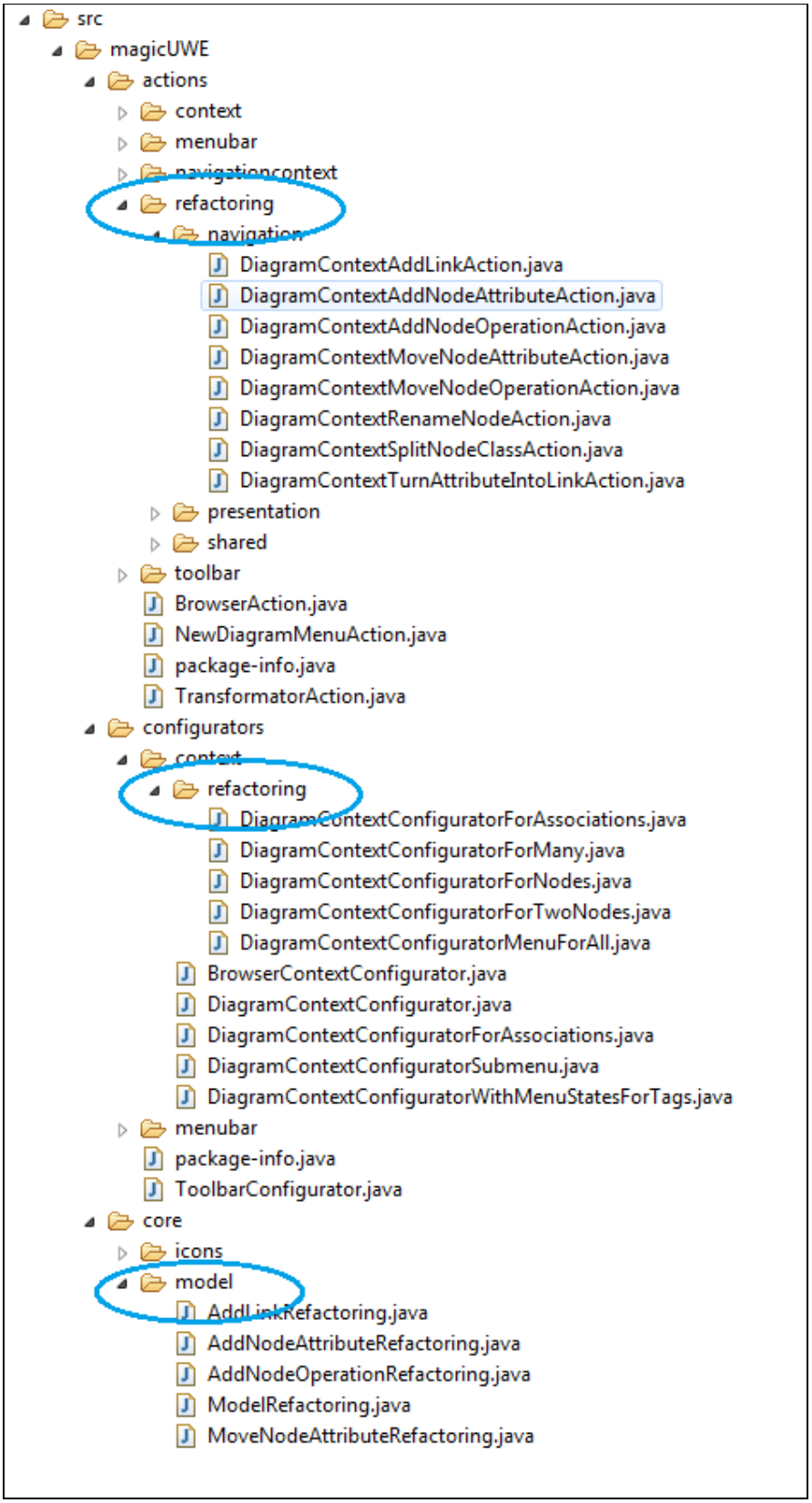


Figura 13. Hot-spots en MagicUWE4R

Básicamente, podríamos resumir la creación de un nuevo refactoring en los siguientes pasos:

1. Creación del Action
2. Desarrollo de la implementación del refactoring en el modelo
3. Actualización del Configurator a cargarse en la inicialización del plugin, para que incluya el nuevo action.

Tomemos el ejemplo del punto 4.2, a partir del cual explicaremos dos casos de extensión a nuestro plugin MagicUWE4R: un refactoring atómico como *Move Node Attribute* y un refactoring compuesto, de mayor complejidad, como lo es *Split Node Class*.

Extendiendo MagicUWE4R con un nuevo refactoring atómico

Veamos como extender nuestro plugin para el caso del refactoring *Move Node Attribute*.

Como mencionamos recientemente, tenemos el paquete src.magicUWE.actions.refactoring. Aquí encontramos los actions correspondiente a los refactorings disponibles (diferenciando los de navegación de los de presentación)

Nosotros desarrollaremos un nuevo Action, creando una nueva clase dentro del paquete src.magicUWE.actions.refactoring.navigation (ya que *Move Node Attribute* es propio del modelo de navegación). Le llamaremos DiagramContextMoveNodeAttributeAction (ver Figura 14)

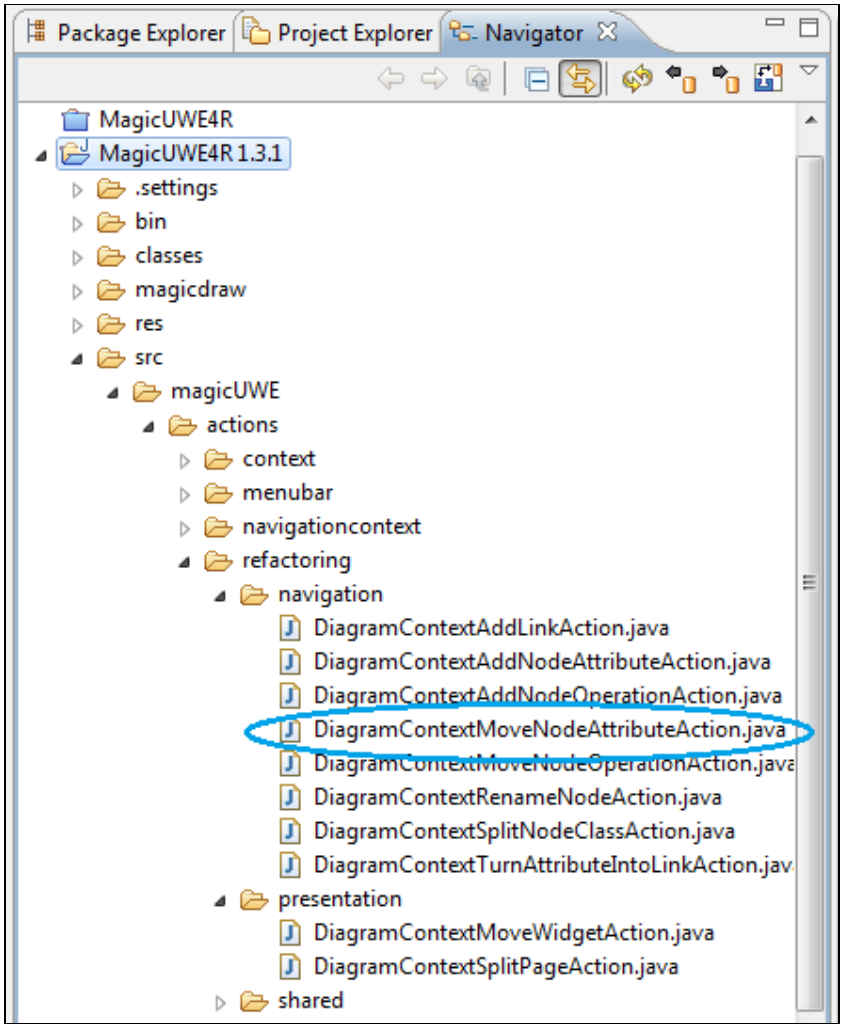


Figura 14. Action para el nuevo refactoring

Tenemos que identificar qué tipo de Action heredará nuestra clase (ver sección 3.2.1). Como no es necesaria la creación de ningún link entre clases, podemos extender NMStateAction, y por ende, implementar el método actionPerformed() que será disparado ante la ocurrencia del refactoring:

DiagramContextMoveNodeAttributeAction.java

```
1  package magicUWE.actions.refactoring.navigation;
2
3  import java.awt.event.ActionEvent;
12
13 public class DiagramContextMoveNodeAttributeAction extends NMStateAction {
14
15     private static final long serialVersionUID = 1L;
16
17     public DiagramContextMoveNodeAttributeAction(String name,
18             KeyStroke keyStroke, String group) {
19         super(name, name, keyStroke, group);
20
21     }
22
23     @SuppressWarnings("unchecked")
24     public void actionPerformed(ActionEvent event) {
25         super.actionPerformed(event);
26
27         MoveNodeAttributeRefactoring moveNodeAttributeRefactoring =
28             new MoveNodeAttributeRefactoring();
29
30         Project project = Application.getInstance().getProject();
31
32         moveNodeAttributeRefactoring.execute(project);
33
34     }
35 }
36
```

Figura 15. Implementación del Action creado. Delegación al modelo

Como podemos apreciar, lo que se hizo aquí es delegar el comportamiento de este refactoring a la clase magicUWE.core.model.MoveNodeAttributeRefactoring. Lo único que hace el Action entonces es obtener la instancia del proyecto actual, y delegar al modelo la ejecución del refactoring.

Para ello, y como vimos en la sección 4.3.1, ahora sólo lo que tenemos que hacer es crear la clase magicUWE.core.model.MoveNodeAttributeRefactoring, extender de NavigationModelRefactoring, e implementar los métodos que correspondan (apertura de sesión, chequeo de precondiciones, implementación del refactoring, etc). Por ejemplo:

```
public class MoveNodeAttributeRefactoring extends NavigationModelRefactoring {

    @SuppressWarnings("unchecked")
    public void refactor(Project project) {

        List selected = project.getActiveDiagram().getContainer().getSelected();

        PresentationElement sourcePresentationElement = (PresentationElement) selected
                .get(0);
        Element sourceElement = sourcePresentationElement.getElement();

        PresentationElement destPresentationElement = (PresentationElement) selected.get(1);

        Element destElement = destPresentationElement.getElement();

        this.refactor(sourceElement, destElement);

    }

    @SuppressWarnings("unchecked")
    public void refactor(Element sourceElement, Element destElement) {

        //Aquí la implementacion del refactoring
    }

    @Override
    public boolean areSelectedCorrect(Project project) {
        return checkClasses(project);
    }

    public void openSession() {
        openSession("Move Node Attribute Refactoring");

    }

}
```

Figura 16. Implementación del comportamiento del refactoring

Por último, necesitamos decirle al ActionsConfiguratorManager que cargue el nuevo Action en la inicialización del plugin. Para ello vamos a la clase PluginManagerActions, y en el método getNavigationDiagramContextTwoNodesRefactoringActions(), agregamos la nueva entrada de la siguiente manera:

```
// Move Node Attribute Refactoring
DiagramContextMoveNodeAttributeAction moveNodeAt =
    new DiagramContextMoveNodeAttributeAction(
        "Move Node Attribute" + ref, null,
        ActionsGroups.PRESENTATION_ELEMENT_SELECTION_RELATED);
setIcon(moveNodeAt, iconsPath + refacFolder + "UWEinsertMenu"
        + iconSuffix);
category.addAction(moveNodeAt);
```

Figura 17. Agregando el Action al ActionsConfiguratorManager

En el capítulo 5 veremos cómo se refleja esto gráficamente en la aplicación MagicDraw.

Extendiendo MagicUWE4R con un nuevo refactoring compuesto

En este caso, veremos cómo extender MagicUWE4R con un nuevo refactoring compuesto, *Split Node Class*, que reutiliza refactoring atómicos existentes.

Remontémonos a la sección 4.2. *Split Node Class* tiene como propósito desacoplar un nodo repleto de información o saturado de funcionalidad ajena a él. Hace uso de los refactorings Move *Node Attribute, Move Node Operation* y *Add Link*.
Vimos en el ejemplo anterior que la creación de un nuevo refactoring se resumía en:

1. Creación del Action
2. Desarrollo de la implementación del refactoring en el modelo
3. Actualización del Configurator a cargarse en la inicialización del plugin, para que incluya el nuevo action.

Pero el Split Node Class se diferencia en que el action deberá, luego de su ejecución, crear un link entre 2 nodos, por lo que es necesario heredar de la clase DrawPathDiagramAction (ver sección 3.2.1). De esta manera, necesitaremos implementar el método createElement()

```
@Override
protected Element createElement() {

    SplitNodeClassRefactoring splitNodeClassRefactoring = new SplitNodeClassRefactoring();
    Project project = Application.getInstance().getProject();
    return splitNodeClassRefactoring.executeAndReturn(project);

}
```

Figura 18. Delegación del Action al modelo

La implementación pura del refactoring en el modelo será una composición: se reutilizarán los refactorings *Move Node Attribute*, *Move Node Operation, Add Link* y *Rename Node.*

```
public Element refactorAndReturn(Project project) {
    Element sourceElement = null;
    MoveNodeAttributeRefactoring moveNodeAttributeRefactoring = new MoveNodeAttributeRefactoring();
    MoveNodeOperationRefactoring moveNodeOperationRefactoring = new MoveNodeOperationRefactoring();
    DiagramPresentationElement activeDiagram = project.getActiveDiagram();
    List selected = activeDiagram.getContainer().getSelected();
    if (!selected.isEmpty()) {
        PresentationElement sourcePresentationElement = (PresentationElement) selected.get(0);
        sourceElement = sourcePresentationElement.getElement();
        Class destClass = MagicDrawElementOperations.createClass(UWEStereotypeClassNav.NAVIGATION_CLASS.toString());
        ShapeElement shape = null;
        try {
            ModelElementsManager.getInstance().addElement(destClass, project.getModel());
            PresentationElementsManager presentationManager = PresentationElementsManager.getInstance();
            shape = presentationManager.createShapeElement(destClass, activeDiagram);
            Rectangle bounds = sourcePresentationElement.getBounds();
            bounds.setBounds(bounds.x + 150, bounds.y + 25, bounds.width, bounds.height);
            PresentationElementsManager.getInstance().reshapeShapeElement(shape, bounds);
        } catch (ReadOnlyElementException e) {
            e.printStackTrace();
        }
        if (RefactoringUtils.hasAttributes(sourceElement)){
            do {
                moveNodeAttributeRefactoring.refactor(sourceElement, destClass);
            } while (RefactoringUtils.wantToMoveAttribute(sourceElement));
        }
        if (RefactoringUtils.hasOperations(sourceElement)){
            do {
                moveNodeOperationRefactoring.refactor(sourceElement, destClass);
            } while (RefactoringUtils.wantToMoveOperation(sourceElement));
        }
    }
    if (sourceElement != null && RefactoringUtils.wantToRenameNode()){
        RenameNodeRefactoring renameNodeRefactoring = new RenameNodeRefactoring();
        renameNodeRefactoring.refactor(sourceElement);
    }
    AddLinkRefactoring addLinkRefactoring = new AddLinkRefactoring();
    return addLinkRefactoring.refactorAndReturn(project);
}
```

Move Node Attribute

Move Node Operation

Rename Node

Add Link

Figura 19. Split Node Class como composición de refactorings atómicos.

Como podemos apreciar en la Figura 19, vemos enmarcados los sectores donde se hace uso de refactorings ya implementados, reflejando la composición de refactorings en el código que se hace mención en la sección 4.2.

CAPÍTULO 5
Refactorings en el Modelo de Navegación

En este capítulo veremos los distintos refactorings del modelo de navegación implementados en MagicUWE4R. Para cada uno de ellos, se hará una breve descripción del mismo, y se ilustrará su funcionalidad en MagicDraw.

Como se dijo en el Capítulo 2, el modelo de navegación define diversos elementos que nos permiten modelar la navegabilidad de la aplicación, y que serán afectados por los refactorings correspondientes. Estos elementos son:

- Nodos.
- El contenido de los mismos (atributos, operaciones).
- Links entre nodos.

Según [25], un refactoring del modelo de navegación afecta a los elementos del modelo de navegación, conservando el comportamiento definido en el modelo conceptual o de la aplicación. Al definir comportamiento de la aplicación, nos referimos a los siguientes aspectos:

1. El conjunto de operaciones del modelo conceptual que son visibles desde el modelo de navegación y su semántica asociada.
2. El alcance de las operaciones a través de los links disponibles.

Los cambios aplicados por el refactoring deben preservar el comportamiento definido en el modelo conceptual y la navegabilidad del modelo de navegación [30].

Asimismo no deben introducir información, relaciones u operaciones que no estén en el modelo de aplicación existente.

5.1 Caso de Estudio

Antes de pasar a explicar los diferentes refactorings implementados en MagicUWE4R, veamos el siguiente caso de estudio que servirá de referencia para los ejemplos del presente capítulo y el siguiente.

Nuestro caso de estudio será una aplicación de ventas online de indumentaria deportiva llamada *SportsStore*. El modelo conceptual se puede ver en la Figura 20, e intenta modelar la tienda (clase *SportsStore*) con sus clientes (*Client*), donde cada uno de ellos realiza compras o pedidos (*Order*). Cada uno de estos pedidos se compone de ítems (*OrderItem*), es decir, de una serie de productos (*Product*) con una cantidad particular para cada uno de ellos. Cada producto tiene una marca asociada (*Brand*). A su vez, la empresa está interesada en la publicación tanto de promociones (*Promo*) y novedades (*News*).

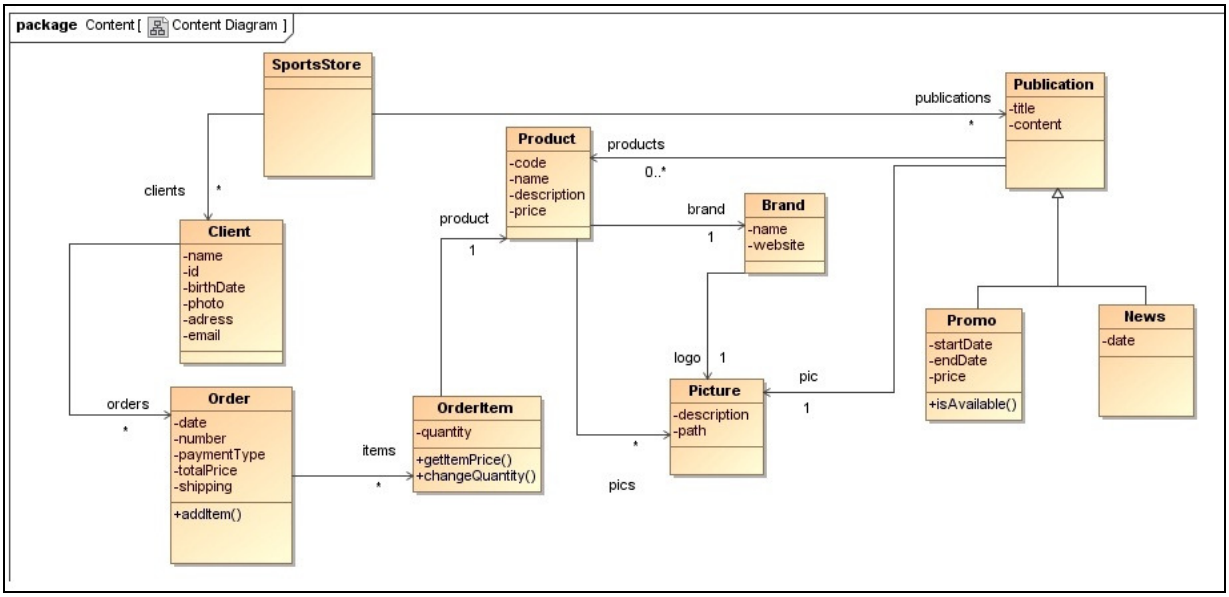


Figura 20. Modelo conceptual de la aplicación de ventas online de indumentaria deportiva

El modelo de navegación correspondiente a este modelo conceptual se muestra en la Figura 21. Este modelo consta de las siguientes clases de nodo:

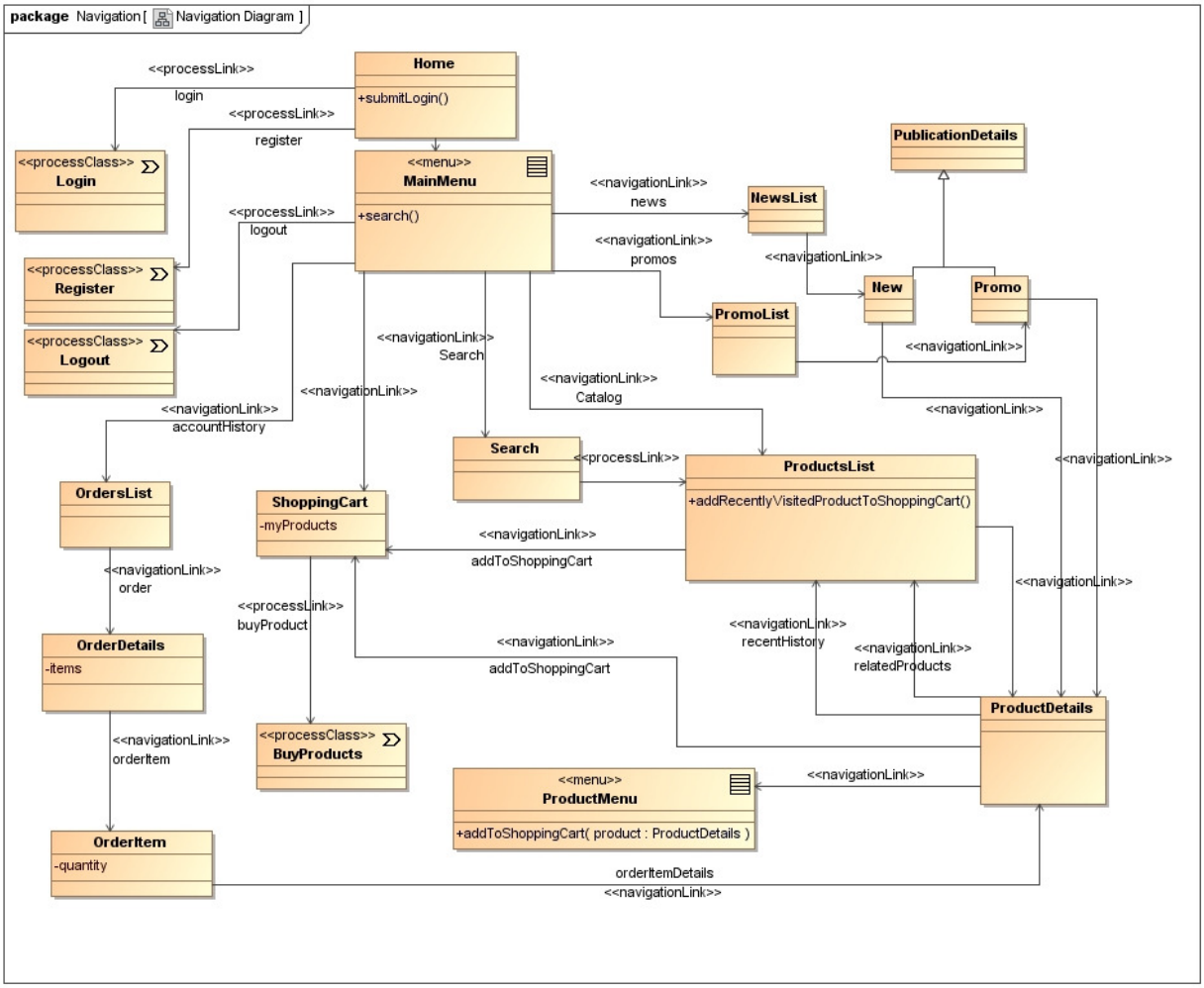


Figura 21. Modelo de navegación de la aplicación de ventas online de indumentaria deportiva

Ahora pasemos a analizar cada uno de los refactorings implementados en MagicUWE4R. Es importante aclarar que este grupo de refactorings es una selección sobre el catálogo que se describe en [25].

Para mejor organización, se seguirá la siguiente estructura para cada uno de ellos:

→ Motivación del refactoring

→ Mecanismo del refactoring

→ Referencias de implementación (en algunos casos de interés de los refactorings del modelo de navegación se muestra un extracto del código)

→ Ejemplo en MagicUWE4R sobre el caso de estudio

5.2 Add Node Operation

Motivación

Las operaciones deben siempre permanecer cerca de los datos sobre los que operan, y es importante tenerlo en cuenta al diseñar aplicaciones Web. Sin embargo, es usual que se agreguen operaciones en etapas posteriores al diseño. Algunos motivos de esto pueden ser:

- Operaciones agregadas al modelo conceptual por nuevos requerimientos, y que deben mapearse al navegacional.
- Querer acelerar procesos ofreciendo acciones tempranas, es decir, que el usuario no deba navegar hasta determinado nodo para ejecutar la operación en cuestión.

Con este refactoring se intenta también establecer una mejora en el diseño, atacando los posibles problemas que puede traer tener las operaciones desacopladas de la información sobre la cual trabaja.

Mecanismo

Refactoring atómico.

Precondiciones:

a) El nodo origen debe existir en el diagrama de navegación
b) La nueva operación no existe en el nodo origen.

Ejecución:

1. Seleccionar el nodo al cual queremos agregar la nueva operación
2. Darle un nombre a la operación.

Implementación

Veamos cuáles son las clases participantes de mayor relevancia, junto a los *hot-spots* para extender la herramienta:

CAPA DE ACTIONS

→ Clase: *DiagramContextAddNodeOperationAction*

→ Superclase: *NMStateAction*

→ Método a extender: *actionPerformed()*

CAPA DE MODELO

→ Clase: *MoveNodeAttributeRefactoring*.

→ Superclase: *NavigationModelRefactoring*

→ Método a extender: *refactor()*

Ejemplo en MagicUWE4R sobre el caso de estudio

Como pudimos ver en el modelo de navegación de la aplicación SportsStore, cuando se está navegando el nodo correspondiente a un producto (nodo ProductDetails), también se puede acceder a los productos recientemente visitados. Sería algo común que una persona esté comparando productos de la misma gama o para el mismo objetivo, y tenga que elegir entre uno de ellos. Si el usuario se encuentra visualizando un producto B, y recientemente había visitado el producto A, puede que se decida finalmente comprar el A. Para esto no sería necesario volver a navegar ese producto que ya analizó y decidió comprar. Sería interesante poder pasar al proceso de compra directamente desde el menú de Recientemente Visitados.

Procedimiento:

PASO 1: Seleccionamos el nodo a refactorizar

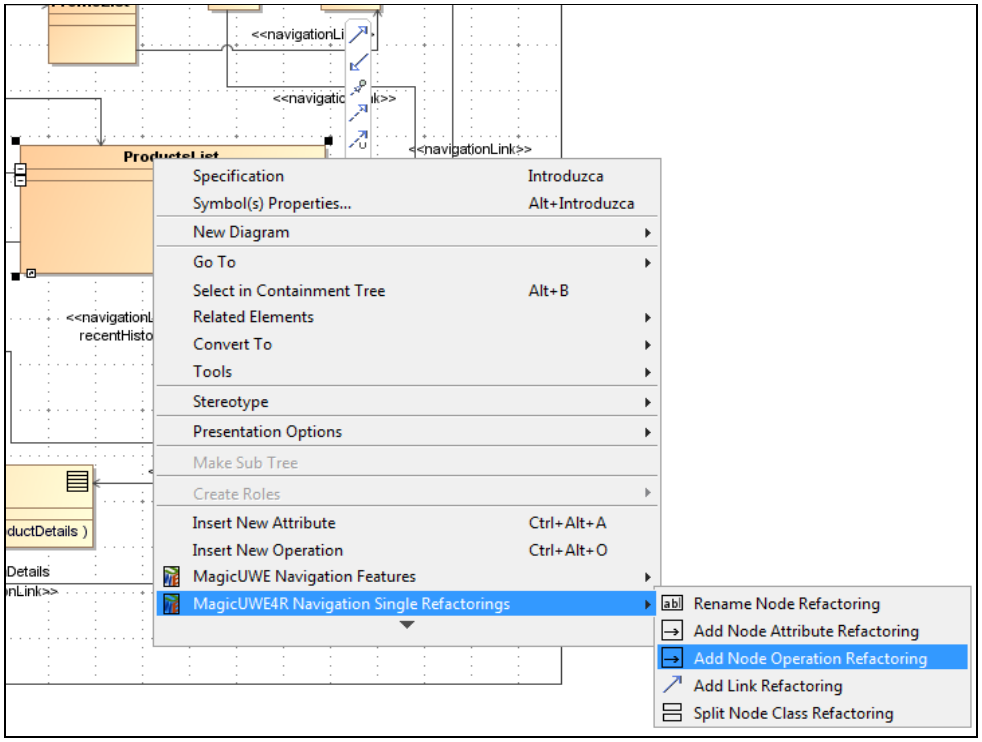


Figura 22. Add Node Operation - Paso 1

(Es importante hacer una observación sobre la figura anterior. Vemos que la categoría seleccionada del menú contextual se llama “MagicUWE4R Navigation Single Refactorings”. Veremos en otros casos que la categoría será “MagicUWE4R Navigation Complex Refactorings”. Esta diferencia viene dada según si se seleccionó un nodo o dos nodos para refactorizar. Entonces, por ejemplo, la primera categoría incluirá el refactoring *Add Node Operation*, mientras que la segunda será, por ejemplo, para el caso de *Move Node Operation*)

PASO 2: Le damos un nombre a la nueva operación

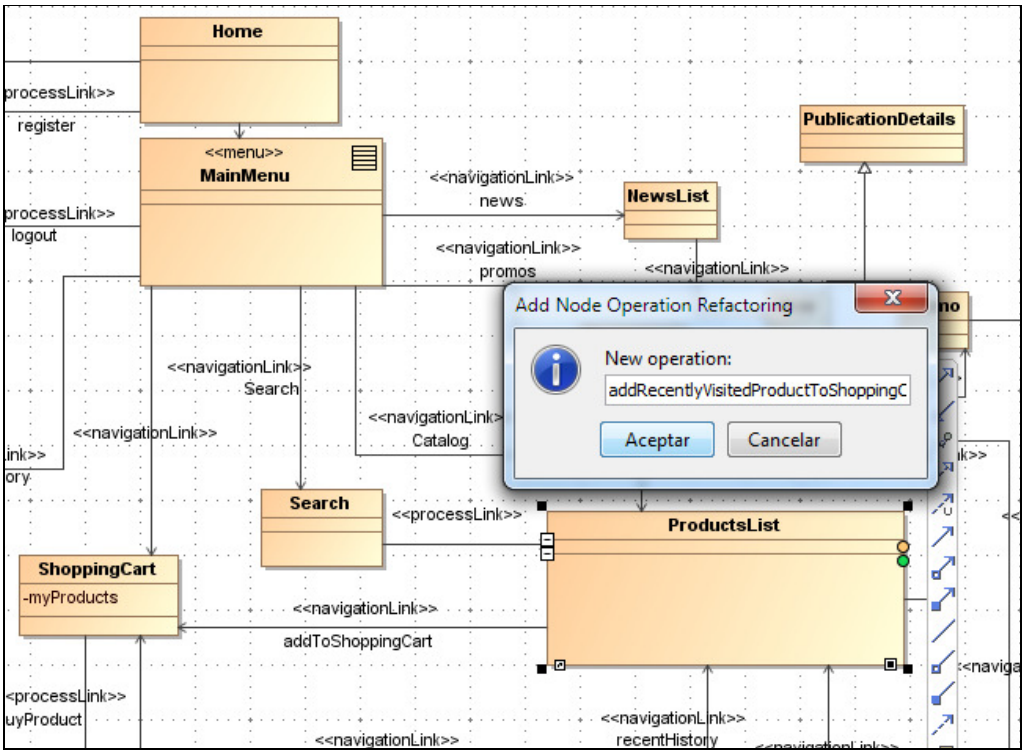


Figura 23. Add Node Operation - Paso 2

RESULTADO FINAL

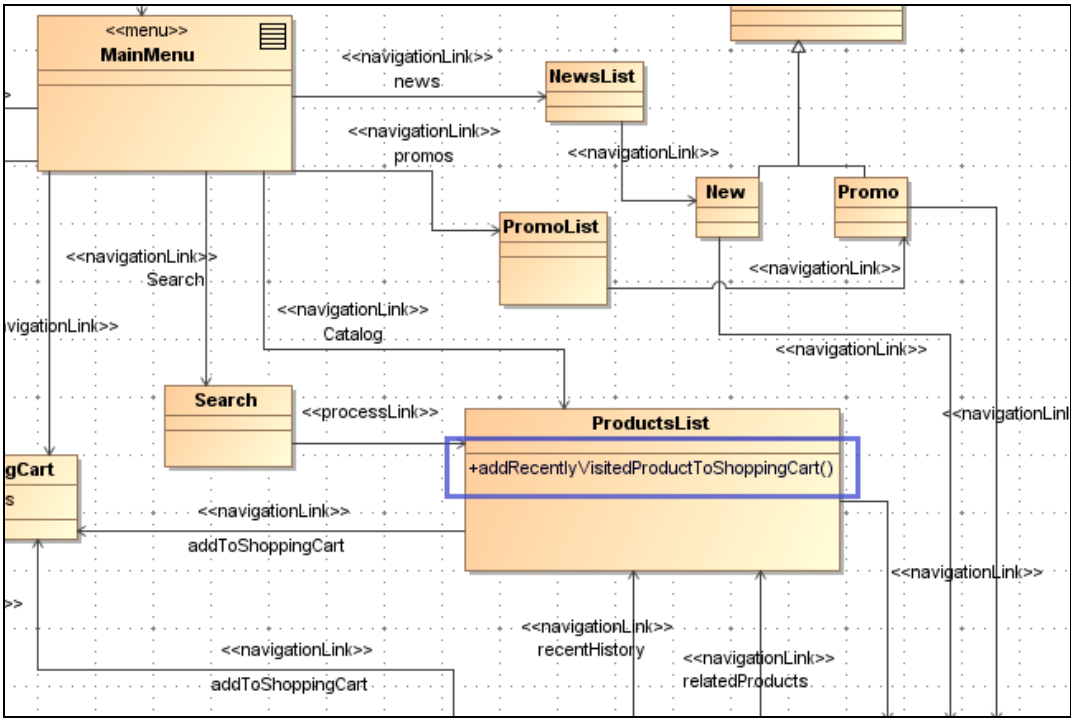


Figura 24. Add Node Operation - Resultado final

5.3 Move Node Attribute

Motivación

Se determina que un atributo de un nodo origen necesita ser movido a otro nodo destino.

Este refactoring es empleado también por otros de tipo compuesto. Por ejemplo, el *Split Node Class* lo utiliza para mover los atributos del nodo dividido y que se quiere desacoplar, al nuevo nodo.

Mecanismo

Refactoring atómico.

Precondiciones:

a) El nodo origen y el nodo destino deben existir en el diagrama navegacional

b) El atributo a mover no existe en el nodo de destino.

Ejecución:

1. Seleccionar nodo origen y nodo destino
2. Escoger el atributo a mover
3. Copiar el atributo elegido en el nodo destino.
4. Eliminar el atributo del nodo origen.

Implementación

Como dijimos, *Move Node Attribute* es empleado por *Split Node* Class, es decir que este último reutiliza la implementación del primero. Es por ello que resulta de interés adjuntar un extracto del código fuente.

CAPA DE ACTIONS

→ Clase: *DiagramContextMoveNodeAttributeAction*

→ Superclase: *NMStateAction*

→ Método a extender: *actionPerformed()*

```
public void actionPerformed(ActionEvent event) {
        super.actionPerformed(event);
        MoveNodeAttributeRefactoring moveNodeAttributeRefactoring = new
MoveNodeAttributeRefactoring();
        Project project = Application.getInstance().getProject();
        moveNodeAttributeRefactoring.execute(project);
}
```

CAPA DE MODELO

→ Clase: *MoveNodeAttributeRefactoring*.

→ Superclase: *NavigationModelRefactoring*

→ Método a extender: *refactor()*

```
public void refactor(Element sourceElement, Element destElement) {
	Class clazz = (Class) sourceElement;
	List<Property> attributes = null;
	List<Property> properties = clazz.getOwnedAttribute();
	if (properties.size() > 0) {
		attributes = RefactoringUtils.getProperties(properties);
	}
	if ((attributes != null) && (attributes.size() > 0)) {
		String pick = (String) JOptionPane.showInputDialog(null, "Choose an
attribute:",
				"Move Node Attribute Refactoring",
JOptionPane.QUESTION_MESSAGE, null, RefactoringUtils
						.toStringArray(attributes),
attributes.get(0).getName());
		if (pick != null) {
			Property property = null;
			int i = 0;
			boolean found = false;
			while ((i < attributes.size()) && (!found)) {
				if (pick.equalsIgnoreCase(attributes.get(i).getName())) {
					found = true;
					property = attributes.get(i);
				}
				i++;
			}
			clazz.getOwnedAttribute().remove(property);
			clazz = (Class) destElement;
			clazz.getOwnedAttribute().add(property);
		} else {
			JOptionPane.showMessageDialog(null, "No attribute selected", "Move
Node Attribute Refactoring", 1);
		}
	} else {
		JOptionPane.showMessageDialog(null, "There is no attributes in source
node",
				"Move Node Attribute Refactoring", 1);
	}
}
```

Ejemplo en MagicUWE4R sobre el caso de estudio

El nodo OrderDetails tiene un atributo *quantity*, que almacena el valor del tamaño de otro atributo *ítems*, que representa la colección de OrderItem. No se hace una distinción de los productos iguales. Es decir, si compramos 10 productos iguales, OrderDetails tendrá una colección de tamaño 10, y ese valor es asignado a la variable quantity.

Podemos introducir una mejora, donde el nodo OrderDetails sigue teniendo una colección de OrderItem, pero en este caso, OrderItem agrupa los atributos de venta de cada producto. Entonces, si compramos 10 productos iguales, vamos a tener una colección de tamaño 1 de OrderItem, y es necesario que esta clase registre el número de productos de ese tipo. Claramente tenemos que aplicar el refactoring *Move Node Attribute*.

Procedimiento:

PASO 1: Seleccionamos los nodos a refactorizar

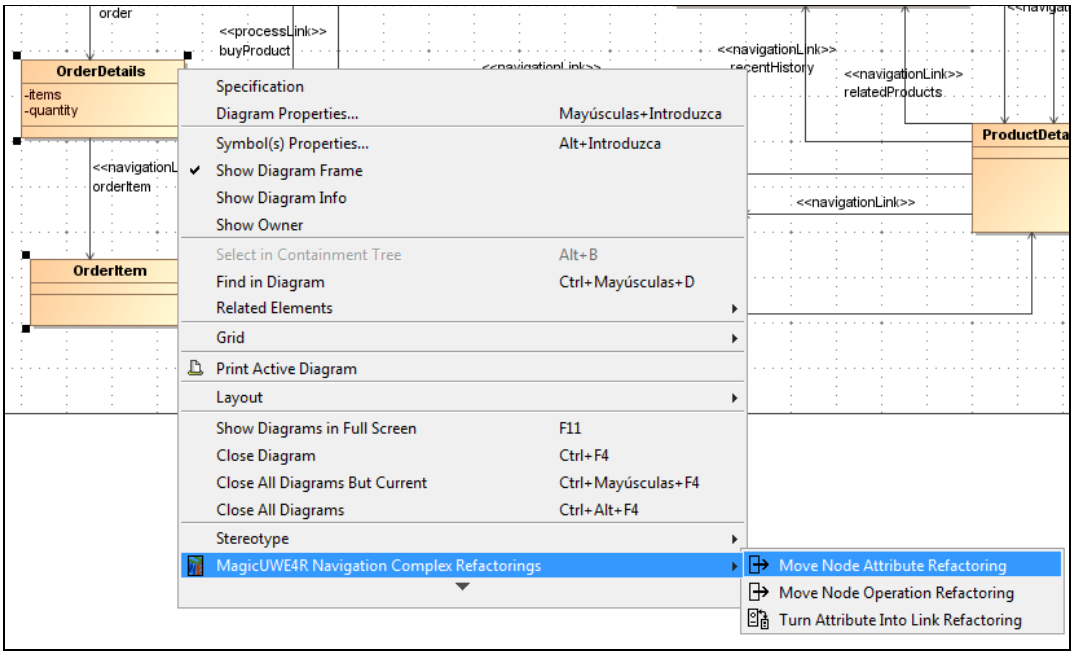


Figura 25. Move Node Attribute - Paso 1

PASO 2: Elegimos el atributo a mover

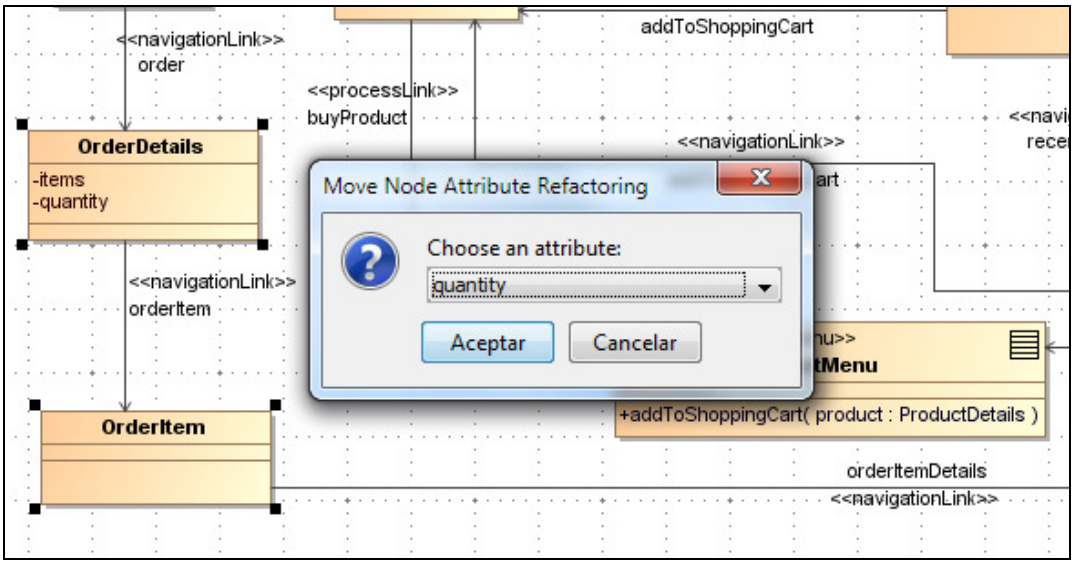


Figura 26. Move Node Attribute - Paso 2

RESULTADO FINAL

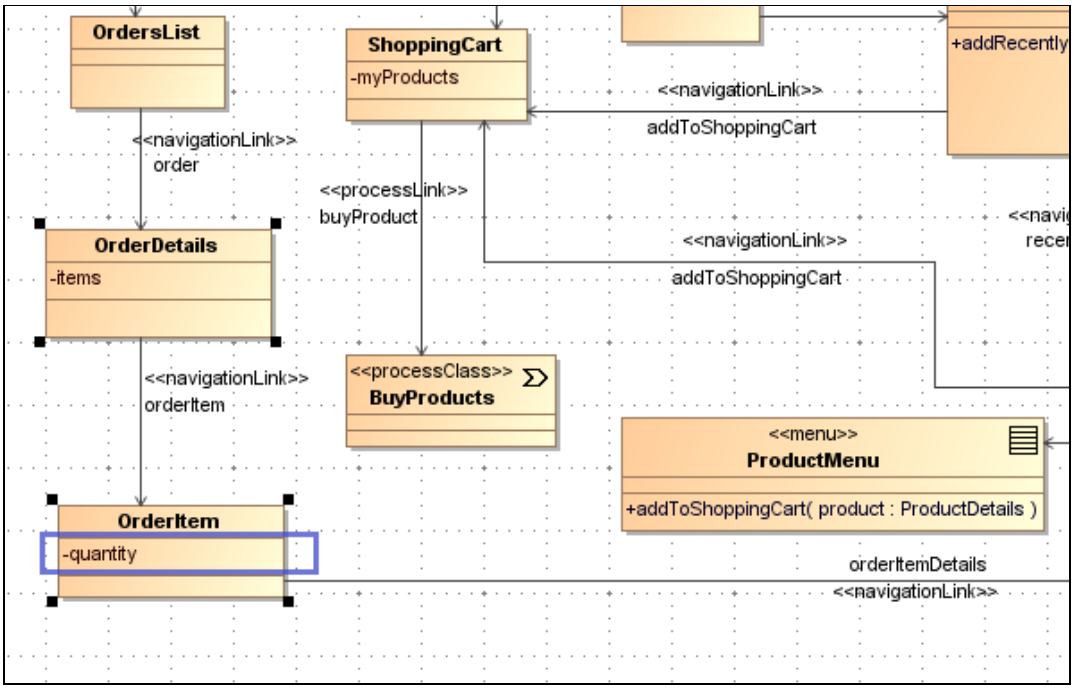


Figura 27. Move Node Attribute - Resultado Final

5.4 Move Node Operation

Motivación

Empleado para mover un método desde un nodo origen al nodo destino. Se utiliza en combinación de otros refactorings atómicos para componer el *Split Node Class*.

Mecanismo

Refactoring atómico.

Precondiciones:

a) El nodo origen y el nodo destino deben existir en el diagrama navegacional

b) El método a mover no existe en el nodo de destino

Ejecución:

1. Seleccionar nodo origen y nodo destino
2. Escoger el método a mover
3. Copiar el método elegido en el nodo destino.

4. Eliminar el método del nodo origen.

Implementación

CAPA DE ACTIONS

→ Clase: *DiagramContextMoveNodeOperationAction*

→ Superclase: *NMStateAction*

→ Método a extender: *actionPerformed()*

```
public void actionPerformed(ActionEvent event) {
        super.actionPerformed(event);
        MoveNodeOperationRefactoring moveNodeOperationRefactoring = new
MoveNodeOperationRefactoring();
        Project project = Application.getInstance().getProject();
        moveNodeOperationRefactoring.execute(project);
}
```

CAPA DE MODELO

→ Clase: *MoveNodeOperationRefactoring*.

→ Superclase: *NavigationModelRefactoring*

→ Método a extender: *refactor()*

```
public void refactor(Element sourceElement, Element destElement) {
        Class clazz = (Class) sourceElement;
        List<Operation> operationsList = clazz.getOwnedOperation();
        if (operationsList.size() > 0) {
                String[] operations = new String[operationsList.size()];
                int i = 0;
                for (Operation operation : operationsList) {
                        if (!(operation.getName().equals(""))) {
                                operations[i] = operation.getName();
                                i++;}
                }
                String eleccion = (String) JOptionPane.showInputDialog(null, "Choose an
operation:",   "Move Node Operation Refactoring", JOptionPane.QUESTION_MESSAGE, null,
operations, operations[0]);
                if (eleccion != null) {
                        i = 0;
                        int indice = 999;
                        boolean found = false;
                        while ((i < operations.length) && (!found)) {
                                if (eleccion.equalsIgnoreCase(operations[i])) {
                                        found = true;
                                        indice = i;
                                }
                                i++;
                        }
                        Operation operation = clazz.getOwnedOperation().get(indice);
                        clazz.getOwnedOperation().remove(indice);
                        clazz = (Class) destElement;
                        clazz.getOwnedOperation().add(operation);
                } else {
                        JOptionPane.showMessageDialog(null, "No operation selected", "Move
Node Operation Refactoring", 1);                    }
        } else {
                JOptionPane.showMessageDialog(null, "There is no operations in source
node",  "Move Node Operation Refactoring", 1);
        }
}
```

Ejemplo en MagicUWE4R sobre el caso de estudio

Tomemos como ejemplo el proceso de *login* de un usuario. En un principio, teníamos en la pantalla de inicio o home el botón de Login que al clickearlo nos mostraba un pop-up con los campos usuario y contraseña. Luego de estudiar el comportamiento de los usuarios en el sitio Web, se pudo determinar que el usuario en la mayoría de los casos quería autenticarse en esta pantalla, por lo que se convertía tedioso el tener que navegar por varios nodos hasta llegar a realizar su operación. Por ende se decidió mover los campos directamente a la pantalla de inicio, o sea que la operación pasó de un nodo a otro para acortar el camino de navegación.

Procedimiento**:**

PASO 1: Seleccionamos los nodos a refactorizar

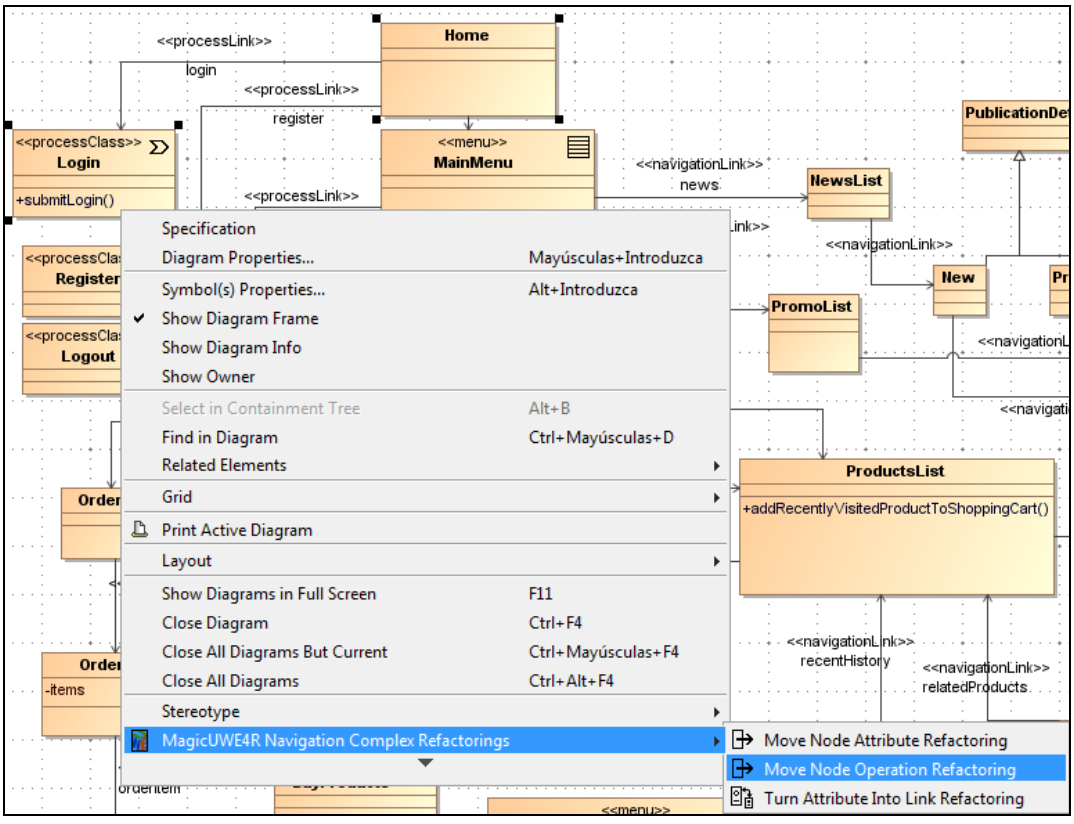


Figura 28. Move Node Operation - Paso 1

PASO 2: Elegimos la operación a mover

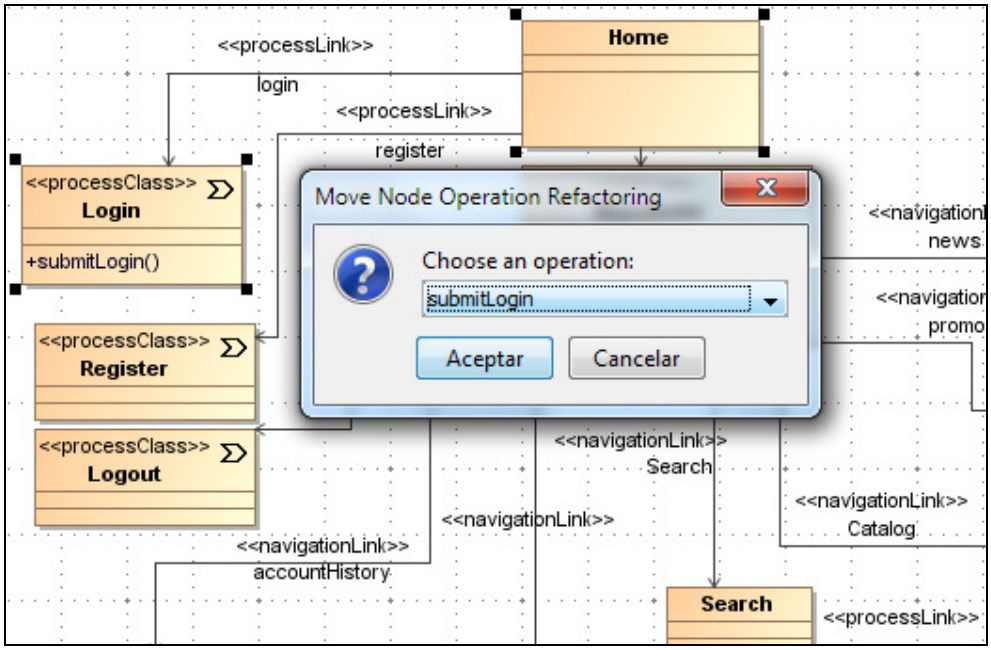


Figura 29. Move Node Operation - Paso 2

RESULTADO FINAL

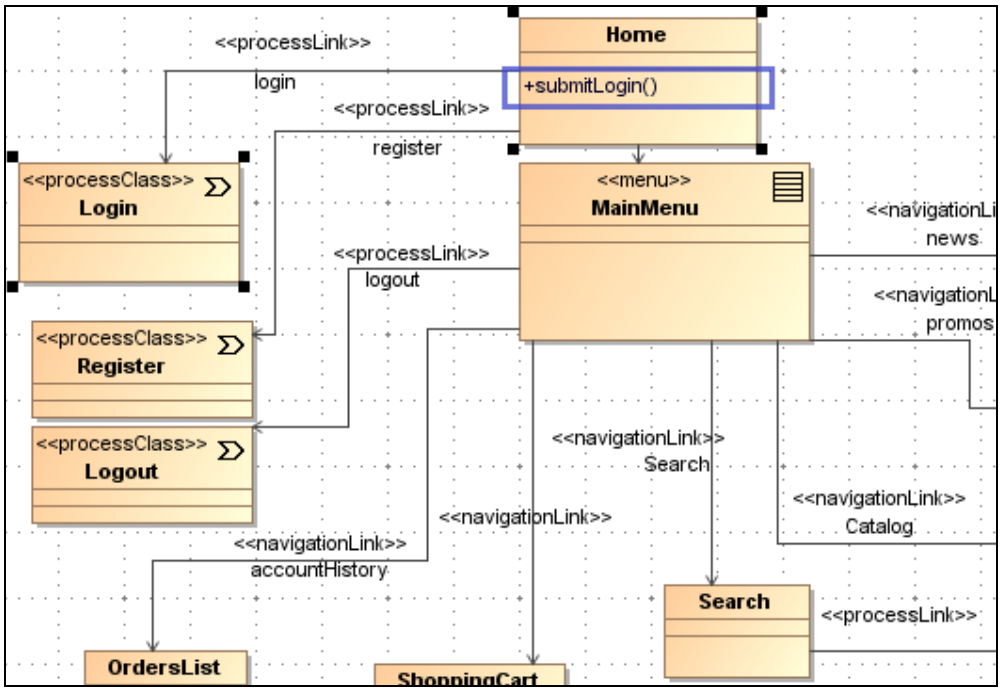


Figura 30. Move Node Operation - Resultado Final

5.5 Rename Node

Motivación

Se le otorga un nuevo nombre a un nodo determinado. Opcionalmente puede ser empleado en *Split Node Class* para renombrar el nodo refactorizado.

Mecanismo

Refactoring atómico.

Precondiciones:

a) El nodo a renombrar debe existir en el diagrama navegacional

b) El nuevo nombre no debe estar presente en nodos existentes.

Ejecución:

1. Seleccionar el nodo al cual queremos renombrar
2. Darle un nuevo nombre al nodo.

Implementación

CAPA DE ACTIONS

→ Clase: *DiagramContextRenameNodeAction*

→ Superclase: *NMStateAction*

→ Método a extender: *actionPerformed()*

```
public void actionPerformed(ActionEvent event) {
        super.actionPerformed(event);
        RenameNodeRefactoring renameNodeRef = new RenameNodeRefactoring();
        Project project = Application.getInstance().getProject();
        renameNodeRef.execute(project);
}
```

CAPA DE MODELO

→ Clase: *RenameNodeRefactoring*.

→ Superclase: *NavigationModelRefactoring*

→ Método a extender: *refactor()*

```
public void refactor(Element element) {
        String str = JOptionPane.showInputDialog(null, "New name: ", "Rename Node
Refactoring", 1);
        if (str != null) {
                Class cl = (Class) element;
                cl.setName(str);
        }
}
```

Ejemplo en MagicUWE4R sobre el caso de estudio

Como vimos en diagramas anteriores, el menú principal cuenta con la función de búsqueda. Supongamos que en un principio esta funcionalidad es básica, y sólo nos ofrecía un campo para ingresar las palabras claves a buscar. Luego, se quiso refinar esta opción, otorgando más opciones de búsqueda como ordenamiento, categorías, fecha, etc. Es decir, se pasa de un *Search* a un *AdvancedSearch*. Para este caso, por ejemplo, se aplica el Rename Node.

Procedimiento:

PASO 1: Seleccionamos el nodo a renombrar

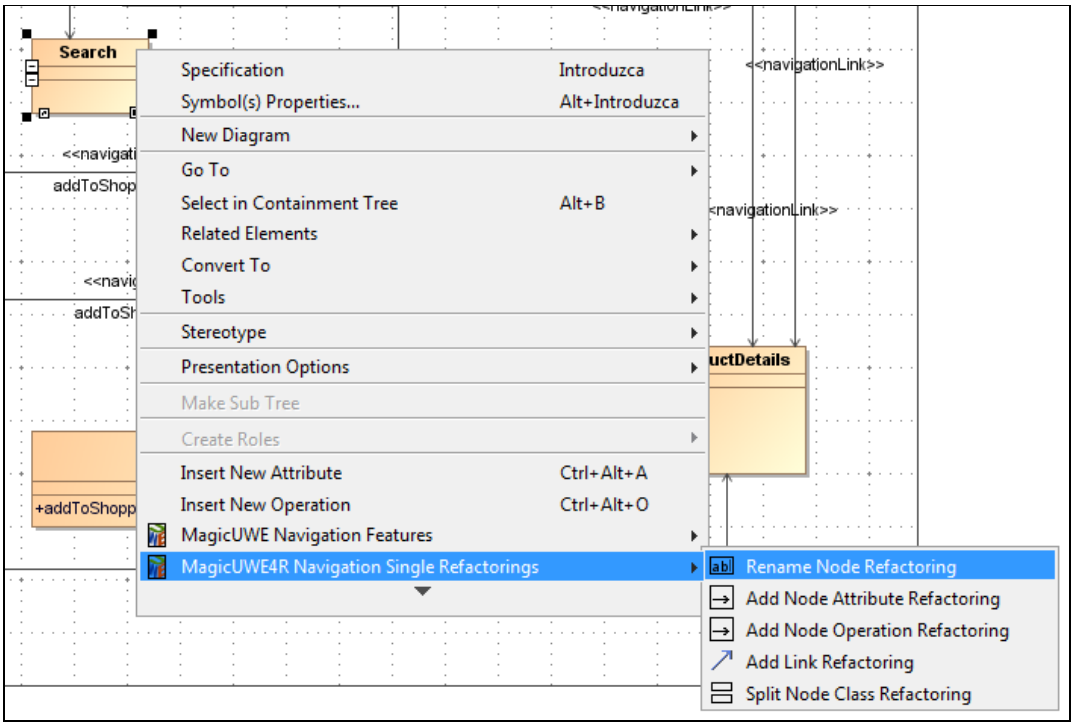


Figura 31. Rename Node - Paso 1

PASO 2: Tipeamos el nuevo identificador del nodo

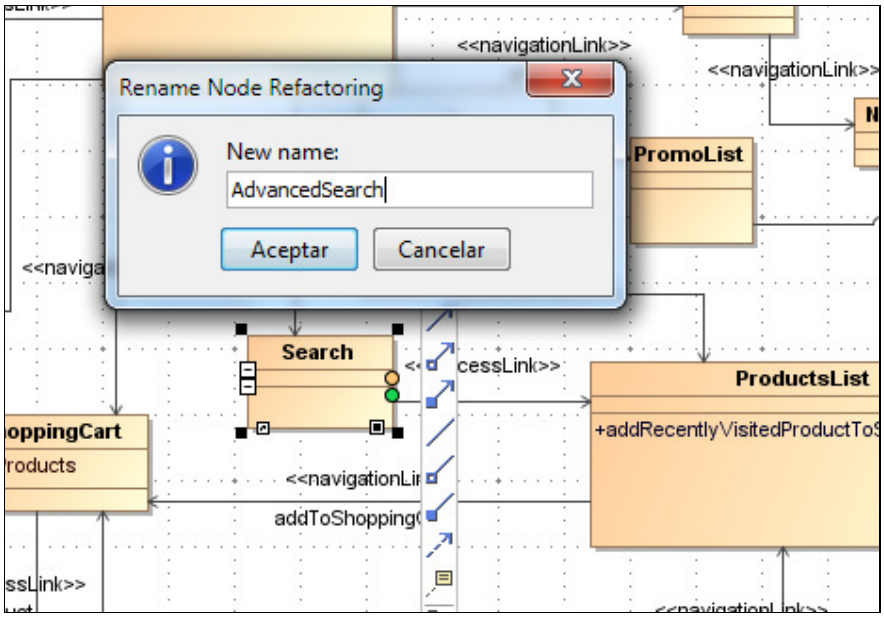


Figura 32. Rename Node - Paso 2

RESULTADO FINAL

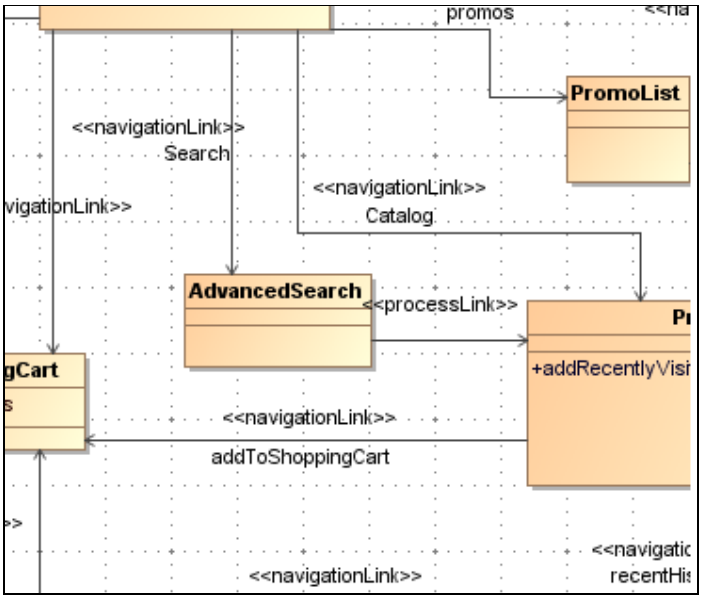


Figura 33. Rename Node - Resultado Final

5.6 Add Link

Motivación

Permite al usuario navegar entre dos nodos. Este refactoring atómico permite al usuario navegar hacia un nuevo nodo que ha sido agregado recientemente en el diagrama de navegación. Otra aplicación para este refactoring es cuando es necesario proveer al usuario un camino nuevo de navegación (acortar el camino) entre nodos distintos que ya son alcanzables desde otro nodo.

A su vez es empleado para componer refactorings como *Split Node Class* y *Turn Attribute Into Link*.

Mecanismo

Refactoring atómico.

Precondiciones:

a) Los nodos a conectar deben pertenecer al diagrama de navegación.
b) No existe un link entre el nodo origen y destino con la misma semántica del que se quiere agregar, lo cual crearía un link redundante.

Ejecución:

1. Seleccionar el nodo origen.
2. Seleccionar el destino del enlace de navegación. El link *to-target* será creado automáticamente entre ambos nodos.

Implementación

CAPA DE ACTIONS

→ Clase: *DiagramContextAddLinkAction*

→ Superclase: *DrawPathDiagramAction*

→ Método a extender: *createElement()*

```
public Element createElement() {
        AddLinkRefactoring addLinkRefactoring = new AddLinkRefactoring();
        Project project = Application.getInstance().getProject();
        return addLinkRefactoring.executeAndReturn(project);
}
```

CAPA DE MODELO

→ Clase: *AddLinkRefactoring*

→ Superclase: *NavigationModelRefactoring*

→ Método a extender: *refactorAndReturn ()*

```
public Element refactorAndReturn(Project project) {
        boolean isDirected = true;
        Association association =
project.getElementsFactory().createAssociationInstance();
        if (this.assocType.toString() != null) {
                MagicDrawElementOperations.addStereotypeToElement(association,
this.assocType.toString());
        }
        association.setName("to-target");
        List<Property> asocProperties = association.getMemberEnd();
        if (asocProperties.size() > 0) {
                if (isDirected) {
                        ModelHelper.setNavigable(asocProperties.get(0), true);
                } else {
                        for (Property property : asocProperties) {
                                ModelHelper.setNavigable(property, true);
                        }
                }
        }
        return association;
}
```

Ejemplo en MagicUWE4R sobre el caso de estudio

Decíamos que *Add Link* es empleado en el *Split Node Class*. Veamos el siguiente ejemplo: Tenemos un nodo ProductListWithDetails a desacoplar en los nodos ProductList y ProductDetails, luego debemos establecer una conexión entre ellos, de manera de establecer la navegabilidad de uno hacia el otro. Aquí entra en juego el presente refactoring.

Procedimiento:

PASO 1: Seleccionamos el nodo origen del enlace

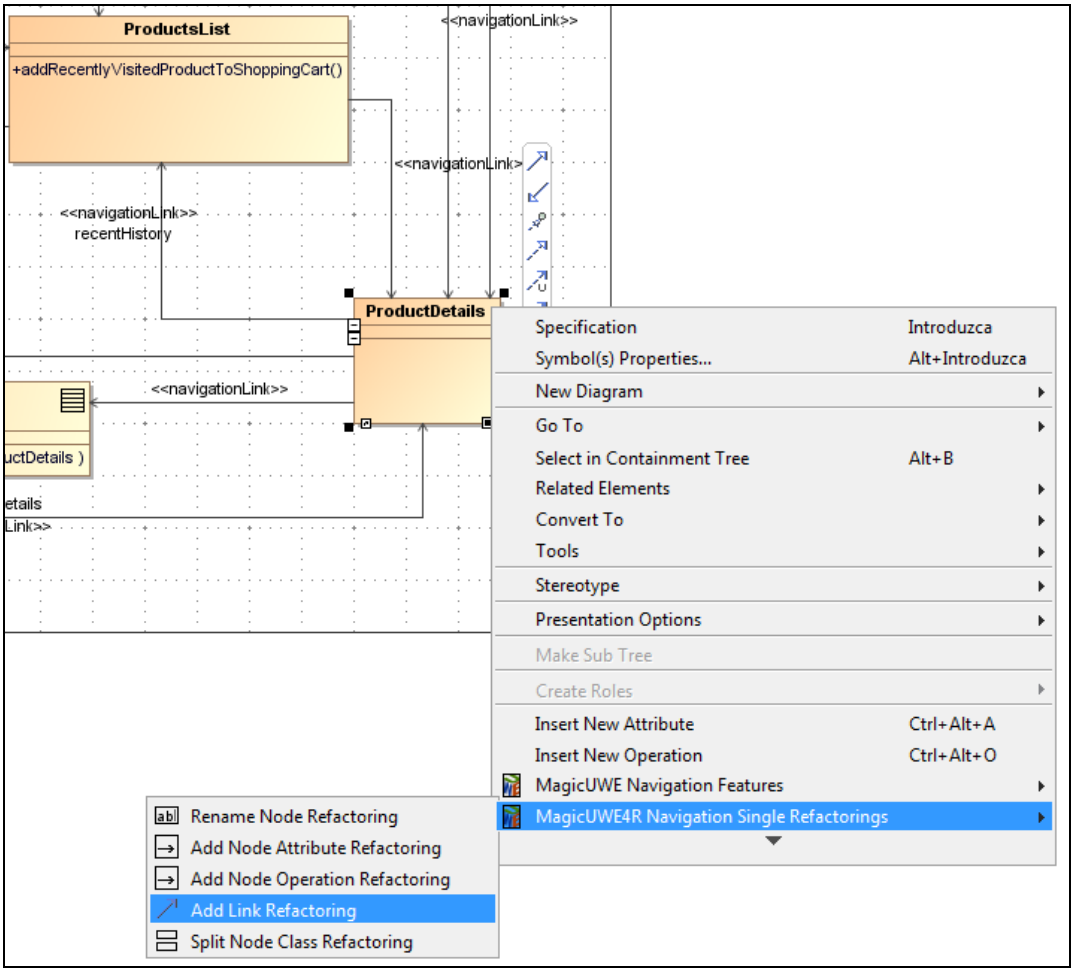


Figura 34. Add Link - Paso 1

PASO 2: Unimos nodo origen con destino

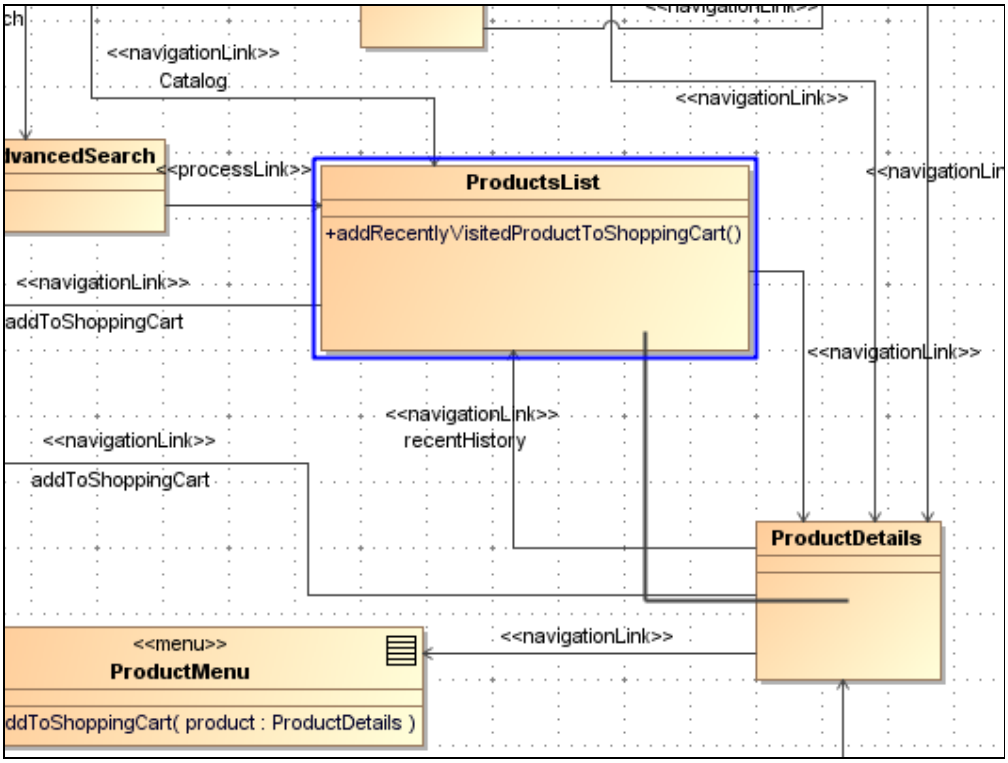


Figura 35. Add Link - Paso 2

RESULTADO FINAL

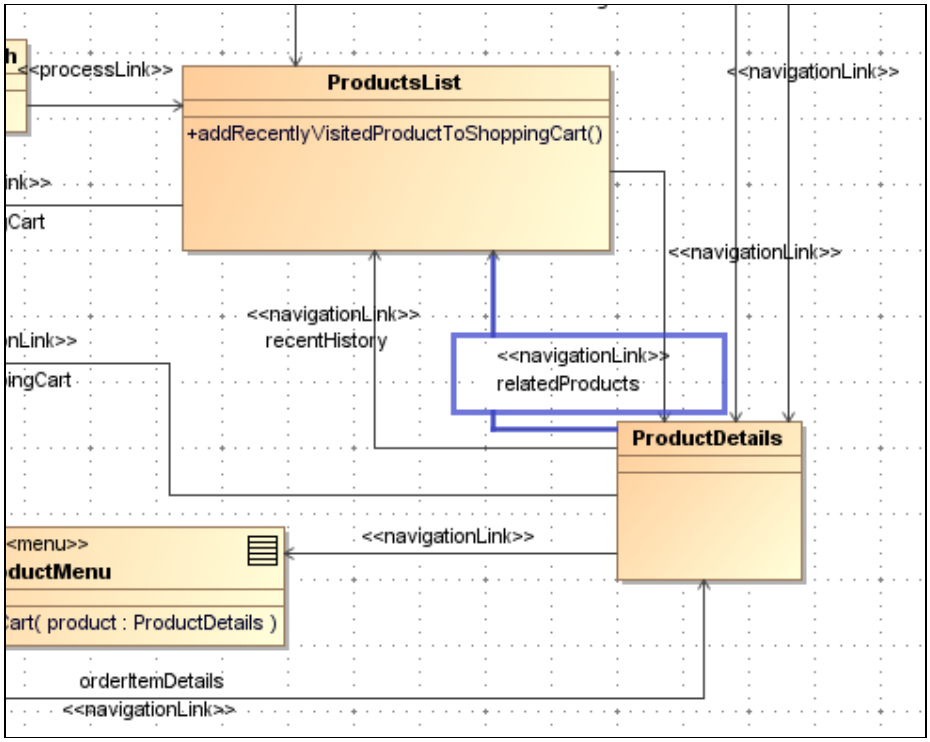


Figura 36. Add Link - Resultado Final

5.7 Split Node Class

Motivación

La motivación de este refactoring es desacoplar un nodo que está repleto de información, saturado de links y funcionalidades. La idea es extraer estos atributos y/u operaciones del nodo en cuestión a un nodo nuevo, estableciendo a su vez un enlace entre ambos.

Otra causa importante de su aplicación es permitir la evolución del diseño del modelo de navegación. Un nodo definido para mapear más de una clase del modelo conceptual puede convertirse extremadamente complejo y necesita ser particionado. Puede pasar lo inverso, que un nodo asociado con una clase del modelo conceptual necesite ser dividido para proveer vistas separadas, pero de mayor cohesión.

Mecanismo

Refactoring compuesto.

Precondiciones:

a) El nombre del nuevo nodo debe ser distinto a cualquiera de los ya existentes en el modelo de navegación.

Ejecución:

1. Agregar una nueva clase de nodo vacía.
2. Para cada atributo que se decida mover desde la clase del nodo original a la clase del nuevo nodo, usar el refactoring *Move Node Attribute* y agregar el atributo a la nueva clase de nodo.
3. Para cada operación que se decida mover desde la clase del nodo original a la nueva clase de nodo, usar el refactoring *Move Node Operation*
4. Usar el refactoring *Add link* para agregar un link entre la clase del nodo original y el nuevo nodo para permitir al usuario poder acceder a la información original y su conjunto de operaciones.
5. Opcionalmente renombrar el nodo original para que refleje el nuevo contenido haciendo uso del refactoring *Rename Node*.

Implementación

En el extracto de código que sigue podremos ver reflejada claramente la composición de refactorings (explicado anteriormente en la sección 4.4).

CAPA DE ACTIONS

→ Clase: *DiagramContextSplitNodeClassAction*

→ Superclase: *DrawPathDiagramAction*

→ Método a extender: *createElement()*

```
protected Element createElement() {
	SplitNodeClassRefactoring splitNodeClassRefactoring = new
SplitNodeClassRefactoring();
	Project project = Application.getInstance().getProject();
	return splitNodeClassRefactoring.executeAndReturn(project);
}
```

CAPA DE MODELO

→ Clase: *SplitNodeClassRefactoring*.

→ Superclase: *NavigationModelRefactoring*

→ Método a extender: *refactorAndReturn()*

```
public Element refactorAndReturn(Project project) {
	MoveNodeAttributeRefactoring moveNodeAttributeRefactoring = new
MoveNodeAttributeRefactoring();
	MoveNodeOperationRefactoring moveNodeOperationRefactoring = new
MoveNodeOperationRefactoring();
	DiagramPresentationElement activeDiagram = project.getActiveDiagram();
	List selected = activeDiagram.getContainer().getSelected();
	if (!selected.isEmpty()) {
		PresentationElement sourcePresentationElement = (PresentationElement)
selected.get(0);
		Element sourceElement = sourcePresentationElement.getElement();
		Class destClass =
MagicDrawElementOperations.createClass(UWEStereotypeClassNav.NAVIGATION_CLASS.toString()
);
		ModelElementsManager.getInstance().addElement(destClass,
project.getModel());
		PresentationElementsManager presentationManager =
PresentationElementsManager.getInstance();
		ShapeElement shape = presentationManager.createShapeElement(destClass,
activeDiagram);
		if (RefactoringUtils.hasAttributes(sourceElement)) {
			do {
				moveNodeAttributeRefactoring.refactor(sourceElement,
destClass);
			} while (RefactoringUtils.wantToMoveAttribute(sourceElement));
		}
		if (RefactoringUtils.hasOperations(sourceElement)) {
			do {
				moveNodeOperationRefactoring.refactor(sourceElement,
destClass);
			} while (RefactoringUtils.wantToMoveOperation(sourceElement));
		}
	}
	if (sourceElement != null && RefactoringUtils.wantToRenameNode()) {
		RenameNodeRefactoring renameNodeRefactoring = new
RenameNodeRefactoring();
		renameNodeRefactoring.refactor(sourceElement);
	}
	AddLinkRefactoring addLinkRefactoring = new AddLinkRefactoring();
	return addLinkRefactoring.refactorAndReturn(project);
}
```

Ejemplo en MagicUWE4R sobre el caso de estudio

Ya hemos hablado de este refactoring en ejemplos anteriores, así que continuaremos usando el mismo caso de la lista de productos y los detalles de cada uno de ellos.

Procedimiento:

PASO 1: Seleccionamos el nodo a desacoplar

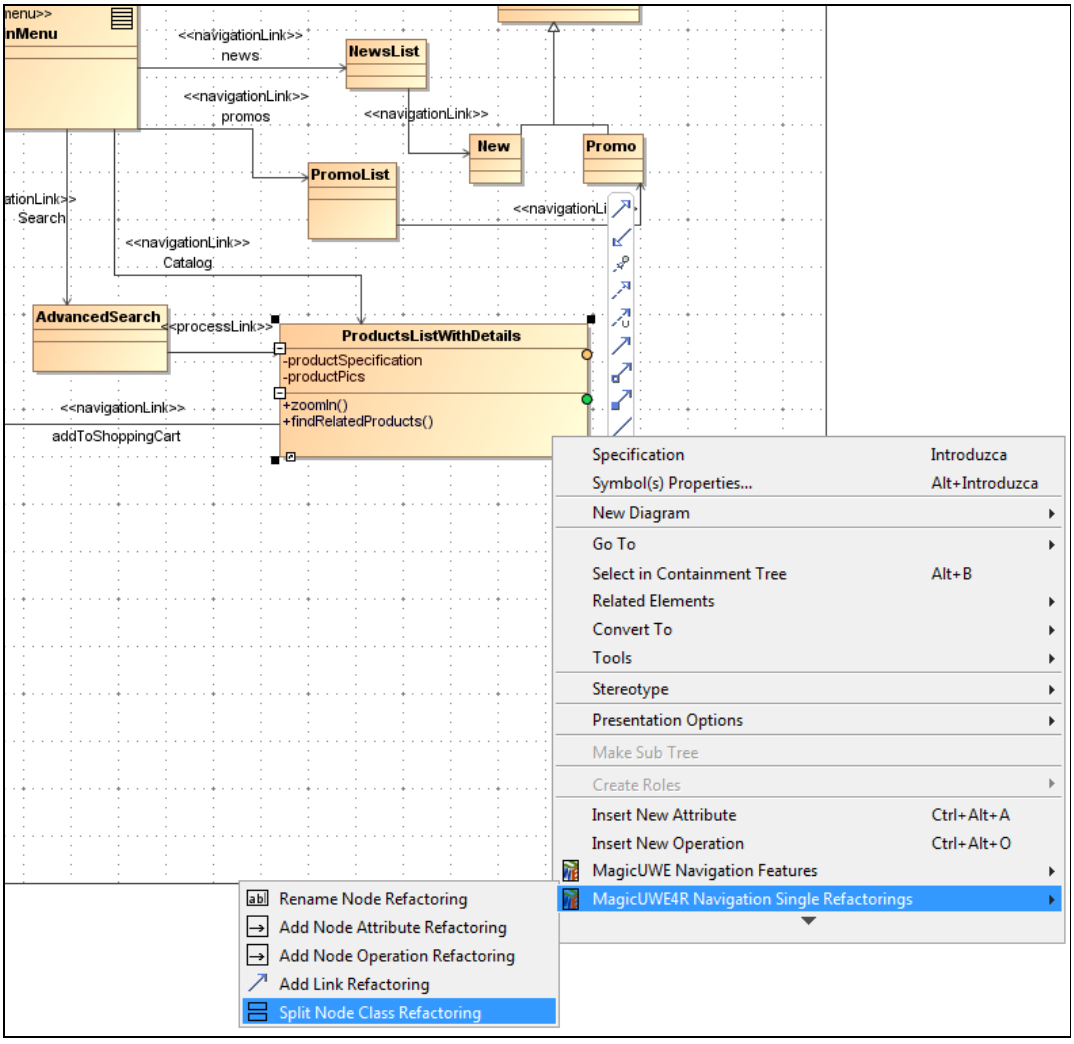


Figura 37. Split Node Class - Paso 1

PASO 2: Se nos ofrece elegir los atributos a mover, haciendo reuso del *Move Node Attribute*

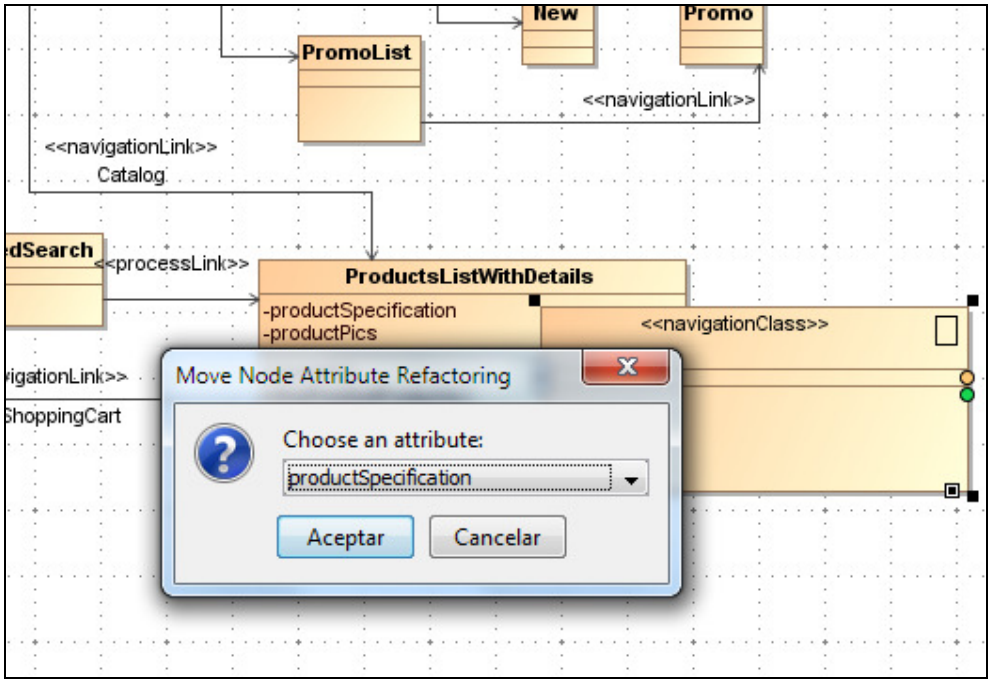


Figura 38. Split Node Class - Paso 2

PASO 3: Se nos da la posibilidad de mover 0 a N atributos

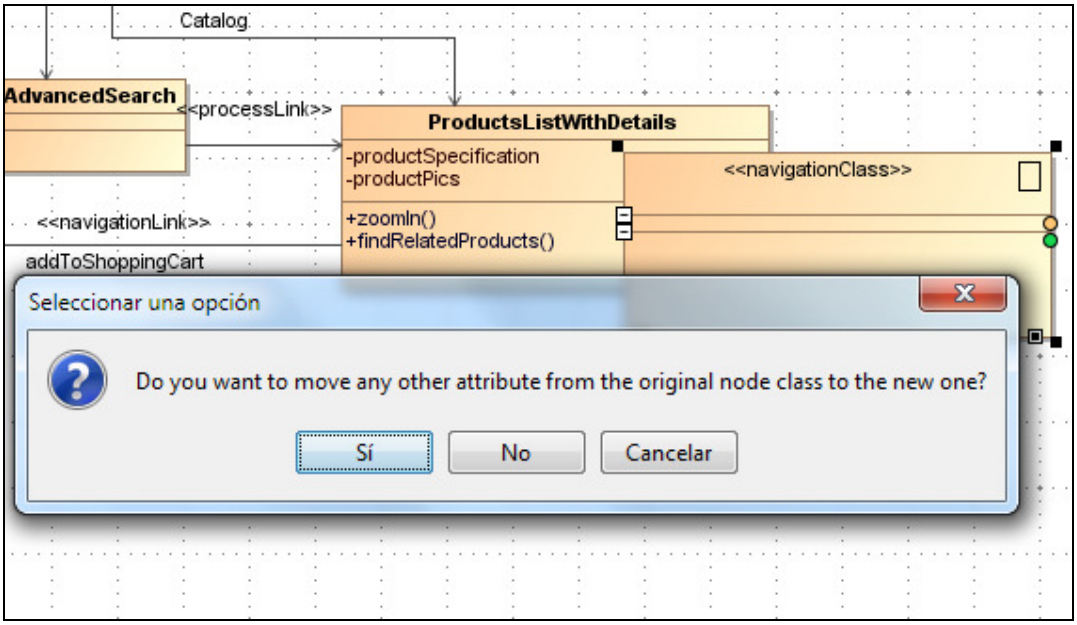


Figura 39. Split Node Class - Paso 3

PASO 4: Se nos ofrece elegir las operaciones a mover, haciendo reuso del *Move Node Operation*

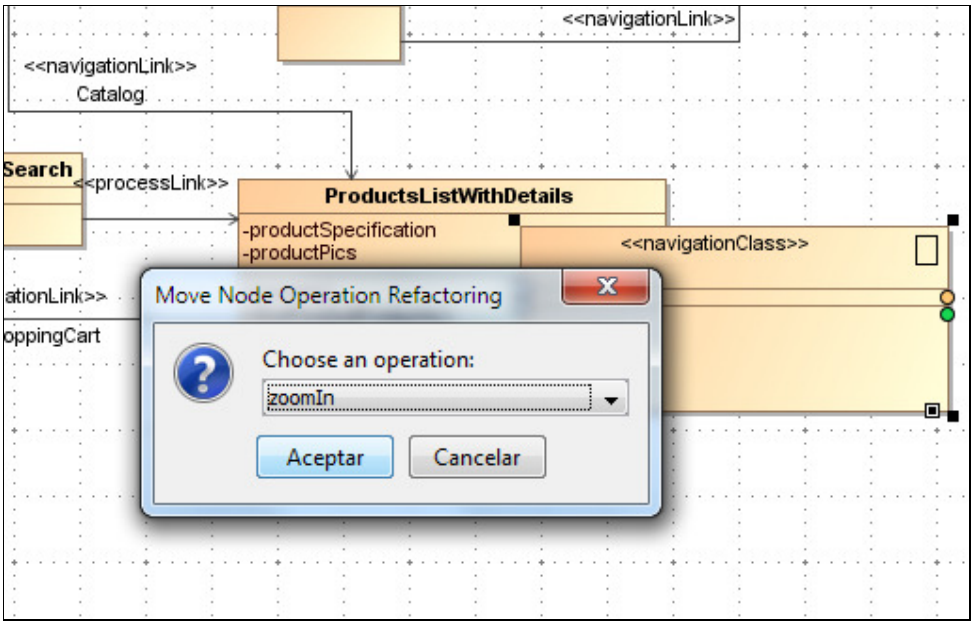


Figura 40. Split Node Class - Paso 4

PASO 5: Se nos da la posibilidad de mover 0 a N operaciones

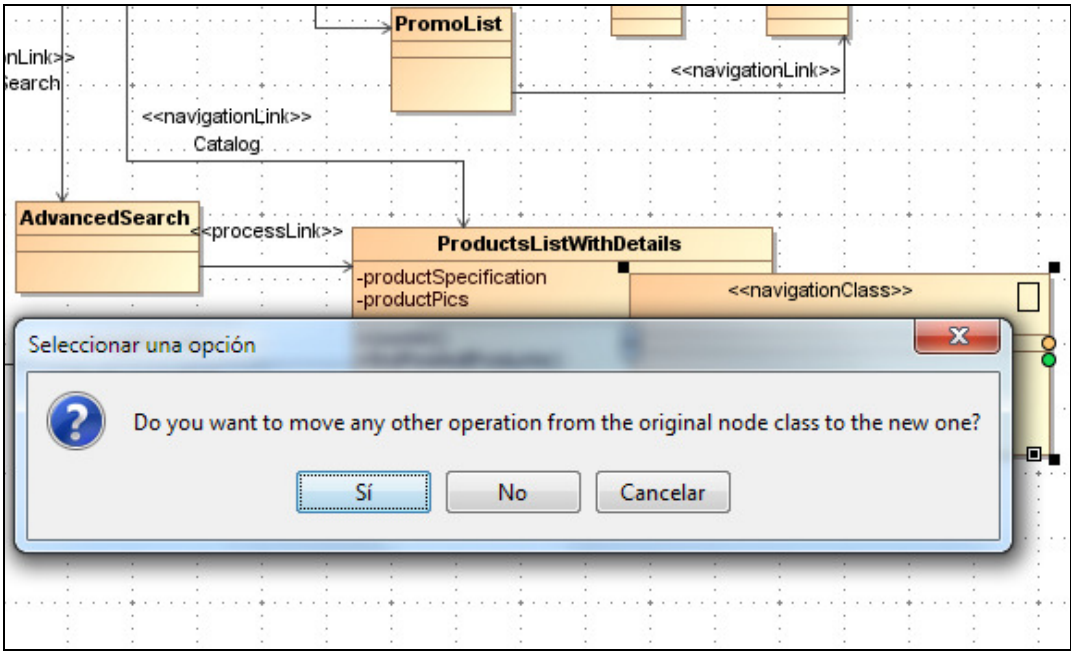


Figura 41. Split Node Class - Paso 5

PASO 6: Se nos ofrece la posibilidad de renombrar el nodo refactorizado haciendo reuso de *Rename Node*

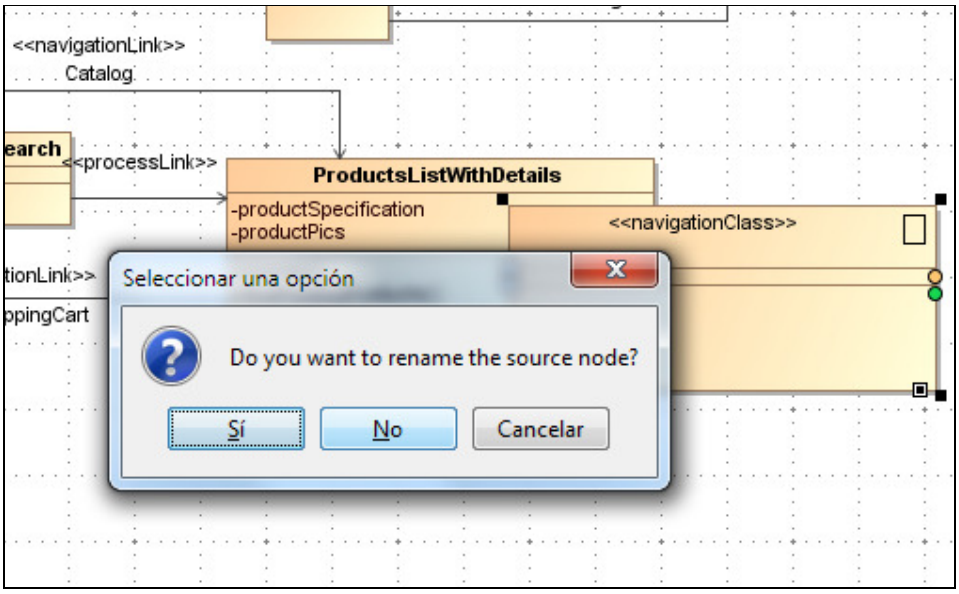


Figura 42. Split Node Class - Paso 6

RESULTADO FINAL

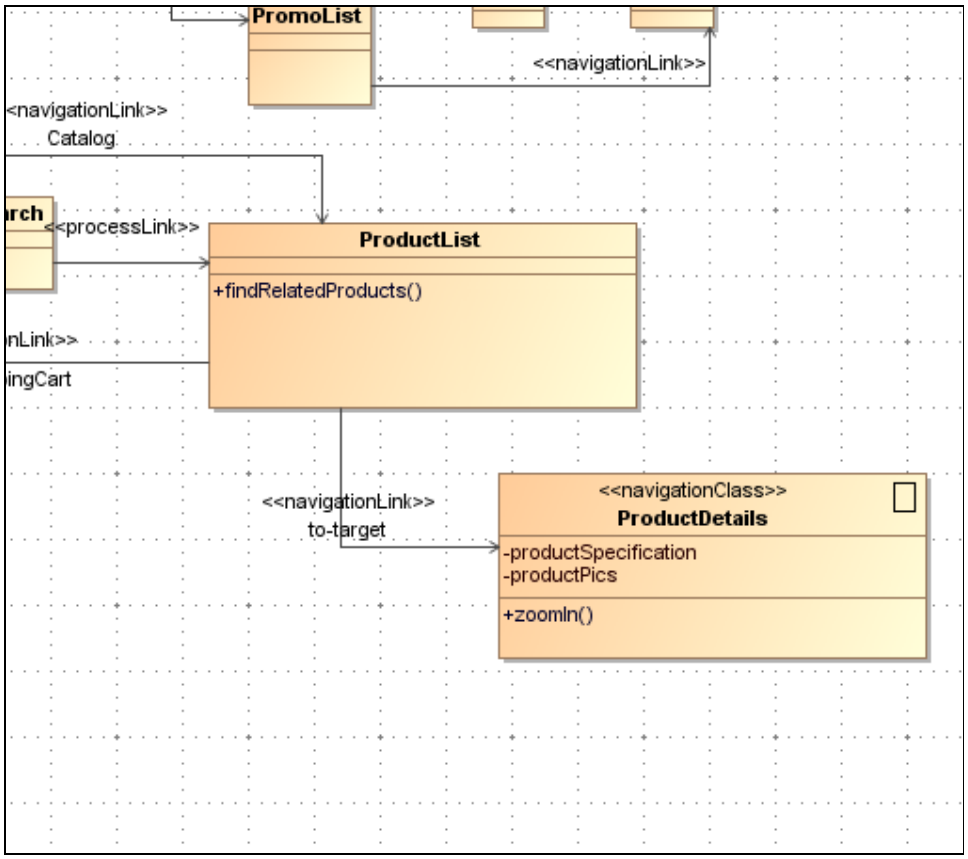


Figura 43. Split Node Class - Resultado Final

5.8 Turn Attribute Into Link

Motivación

Este refactoring ataca la necesidad de transformar un atributo que se muestra en la pantalla, en un camino de navegación hacía más detalles sobre lo que ese atributo representa [31].

Mecanismo

Refactoring compuesto.

Precondiciones:

a) El link con nombre del atributo seleccionado no existe previamente entre nodo origen y nodo destino.

Ejecución:

1. Seleccionar el atributo en el nodo de origen que deseamos convertir en el enlace.
2. Usar el refactoring *Add Link* para agregar un nuevo link, con nombre del atributo seleccionado previamente, desde el nodo de origen al destino
3. En el nodo origen, reemplazar la definición del atributo por la definición de un link o ancla.

Implementación

CAPA DE ACTIONS

→ Clase: *DiagramContextTurnAttributeIntoLinkAction*

→ Superclase: *DrawPathDiagramAction*

→ Método a extender: *createElement()*

CAPA DE MODELO

→ Clase: *TurnAttributeIntoLinkRefactoring*.

→ Superclase: *NavigationModelRefactoring*

→ Método a extender: *refactorAndReturn()*

Ejemplo en MagicUWE4R sobre el caso de estudio

El usuario puede visualizar los ítems que posee pendientes a adquirir en el carrito de compras. El usuario sólo puede visualizar el nombre de los productos que ha comprado, pero es incapaz de entrar a los detalles de los productos desde el carrito.
Al aplicar el refactoring transformamos el atributo myProducts en el nodo del ShoppingCart, que solo contenía el nombre de los productos, para que se transforme en un link hacia los detalles de cada uno de los mismos

Procedimiento:

PASO 1: Seleccionamos nodo origen y nodo destino

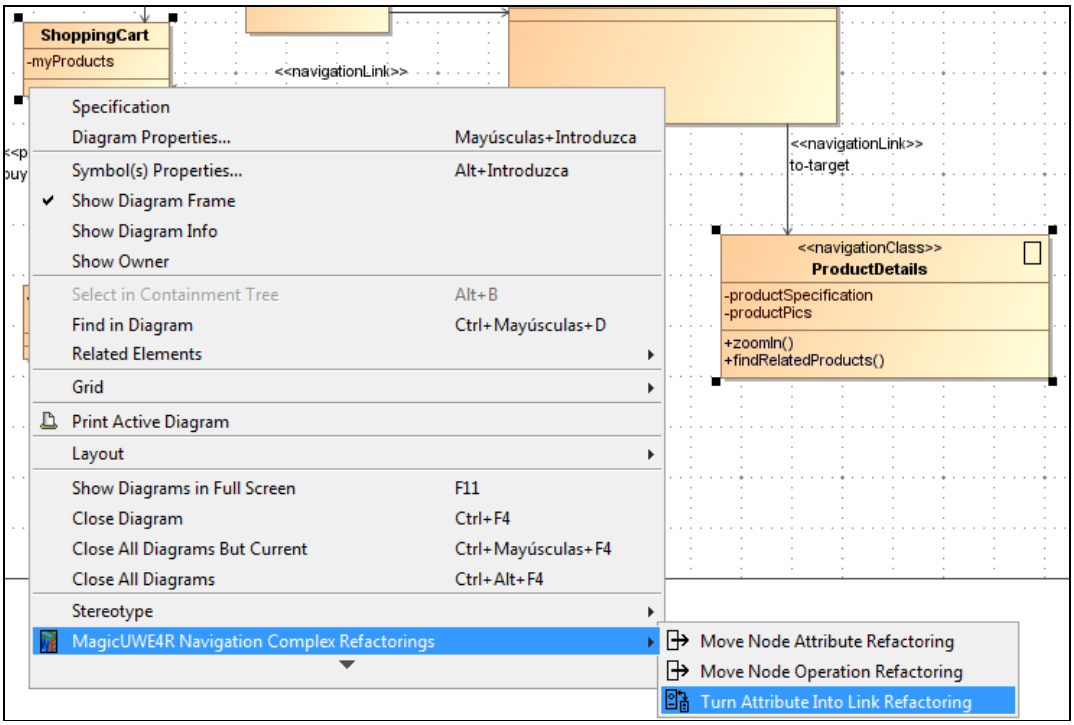


Figura 44. Turn Attribute Into Link - Paso 1

PASO 2: Seleccionamos el atributo a convertir en link

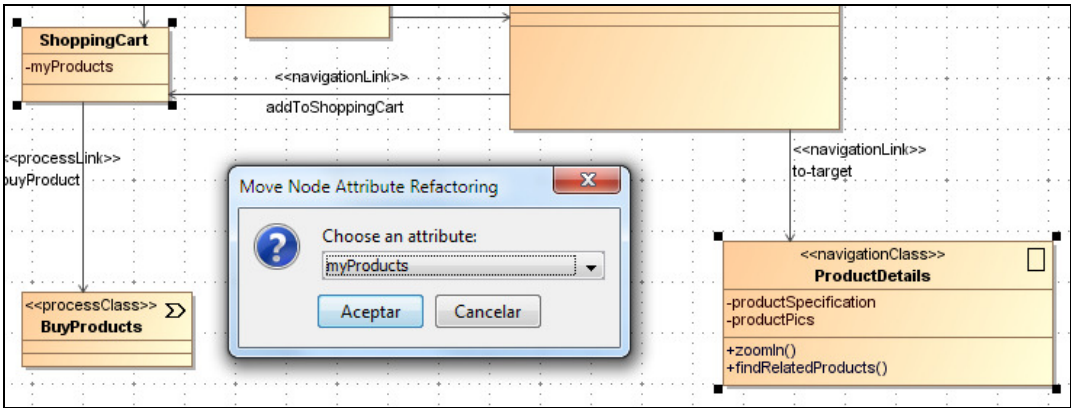


Figura 45. Turn Attribute Into Link - Paso 2

PASO 3: Establecemos el enlace entre nodo origen y destino

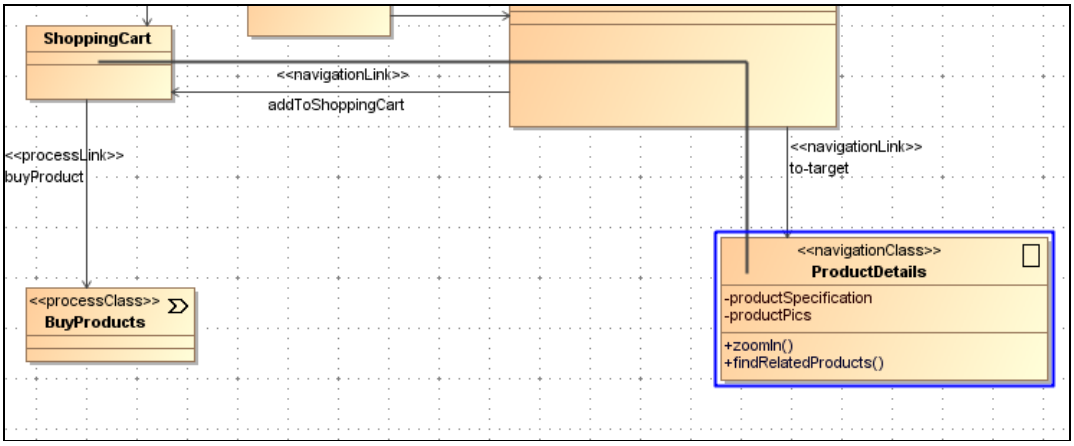


Figura 46. Turn Attribute Into Link - Paso 3

RESULTADO FINAL

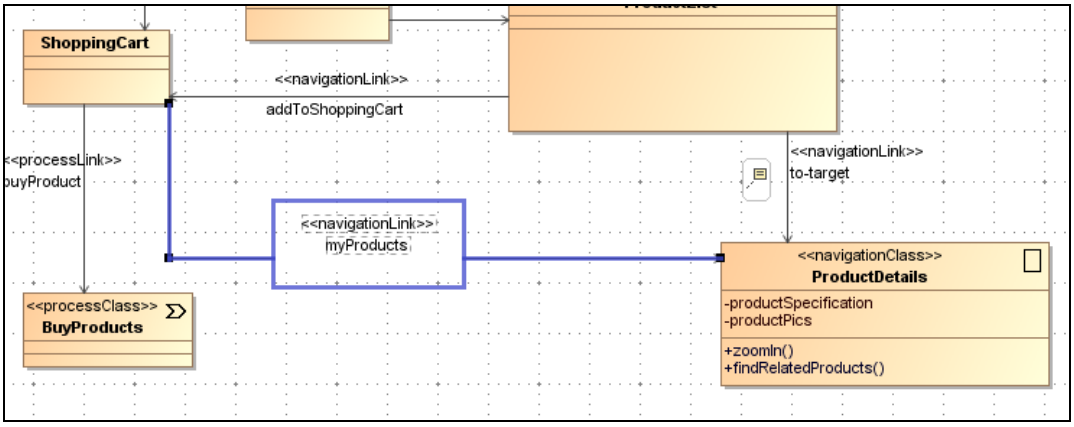


Figura 47. Turn Attribute Into Link - Resultado Final

CAPÍTULO 6
Refactorings en el Modelo de Presentación

En este capítulo veremos los distintos refactorings del modelo de presentación implementados en MagicUWE4R. Para cada uno de ellos, se hará una breve descripción del mismo, y se ilustrará su funcionalidad en MagicDraw.

En los diagramas de presentación se definen diversos elementos que nos permiten modelar la interfaz gráfica de una aplicación Web.

Los elementos que serán afectados por nuestros refactoring son:

- El tipo de los objetos de interfaz (widgets) que componen la página
- La composición de los widgets en cada página.
- Las transformaciones de interfaz que se suceden luego de la interacción del usuario.

Los refactorings del modelo de presentación pueden:

→ Cambiar el tipo de los widgets por otro, pero conservando la funcionalidad subyacente.

→ Cambiar la disposición de los widgets en las páginas.

→ Agregar información u operaciones a una página, siempre y cuando el modelo de navegación y conceptual lo soporten.

→ Cambiar los efectos de la interfaz

A continuación se describen los refactorings del modelo de presentación implementados en MagicUWE4R. Veremos que el causante de gran parte de ellos es la aplicación de un refactoring sobre el modelo de navegación.

Es importante aclarar que este grupo de refactorings es un subconjunto sobre el catálogo que se describe en [25].

6.1 Move Widget

Motivación

Refactoring que permite mover un widget de presentación de una página a otra. Este refactoring atómico es empleado usualmente al aplicar un *Split Page*, donde algunos widgets necesitan ser movidos de la página original a la nueva.

Mecanismo

Refactoring atómico.

Precondiciones:

a) El widget no existe en la página destino

Ejecución:

1. Se seleccionan página origen y página destino.
2. Se elige el widget a trasladar.
3. Se copia la definición del widget a la página destino.
4. Se elimina el widget de la página origen.

Implementación

CAPA DE ACTIONS

→ Clase: *DiagramContextMoveWidgetAction*

→ Superclase: *DrawShapeDiagramAction*

→ Método a extender: *createElement()*

CAPA DE MODELO

→ Clase: *MoveWidgetRefactoring*.

→ Superclase: *PresentationModelRefactoring*

→ Método a extender: *refactorAndReturn()*

Ejemplo en MagicUWE4R sobre el caso de estudio

Tomemos el ejemplo empleado en el refactoring navegacional *Move Node Operation*, basado en el proceso de login de un usuario, donde se pasó de tener un pop-up con los campos usuario y contraseña, a tener embebidos estos campos dentro del home. Para ello, en el modelo de presentación, es necesario mover widgets de una página a otra.

Procedimiento:

PASO 1: Seleccionamos páginas origen y destino

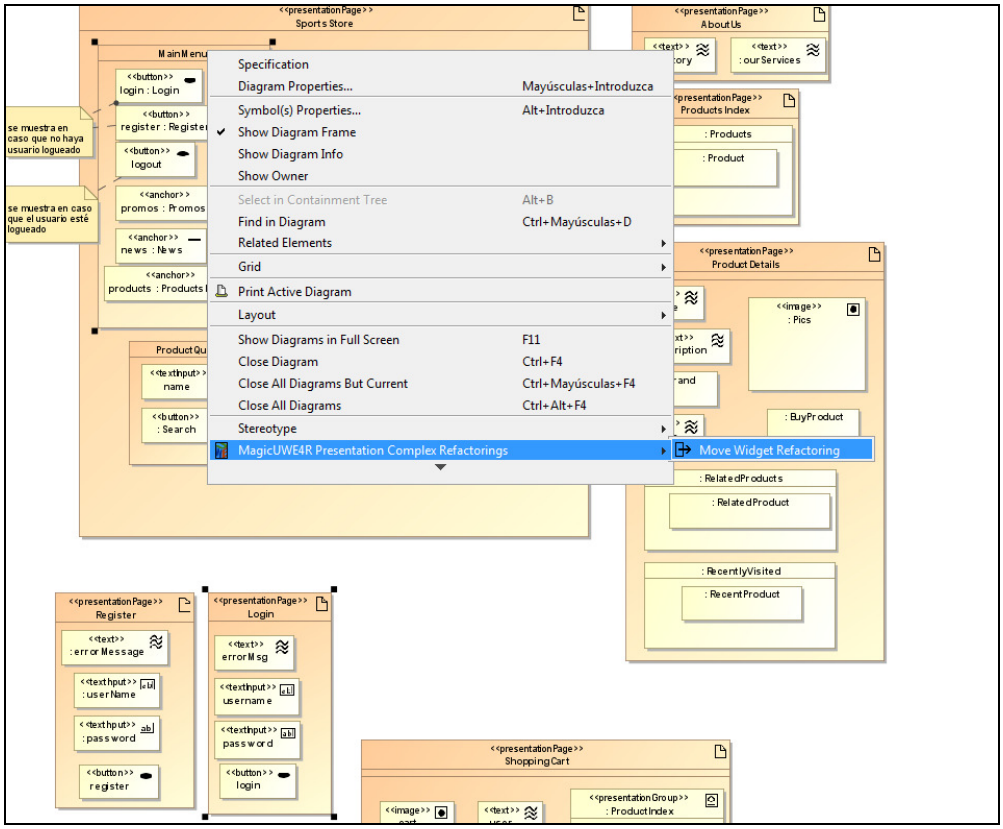


Figura 48. Move Widget - Paso 1

PASO 2: Seleccionamos el widget a mover

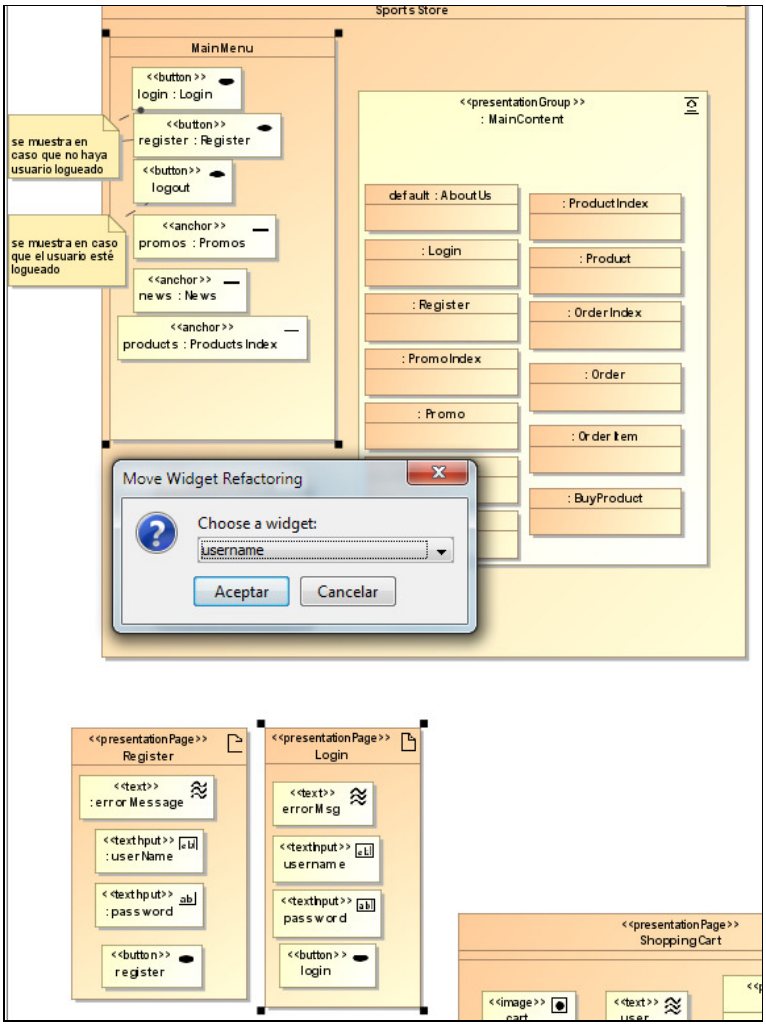


Figura 49. Move Widget - Paso 2

PASO 3: Indicamos la nueva página del widget

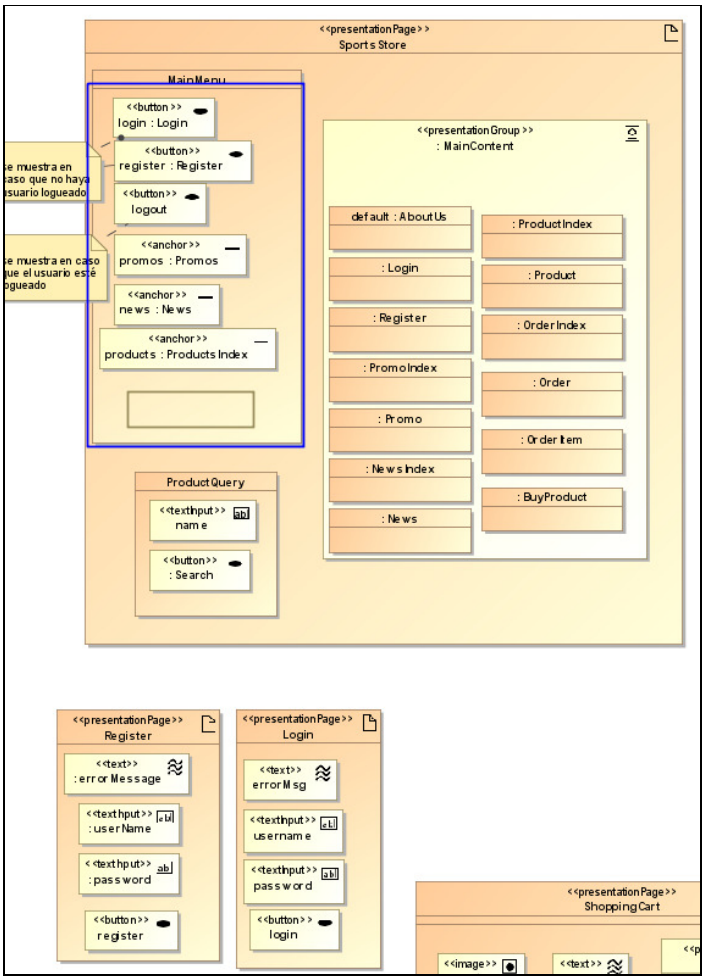


Figura 50. Move Widget - Paso 3

RESULTADO FINAL

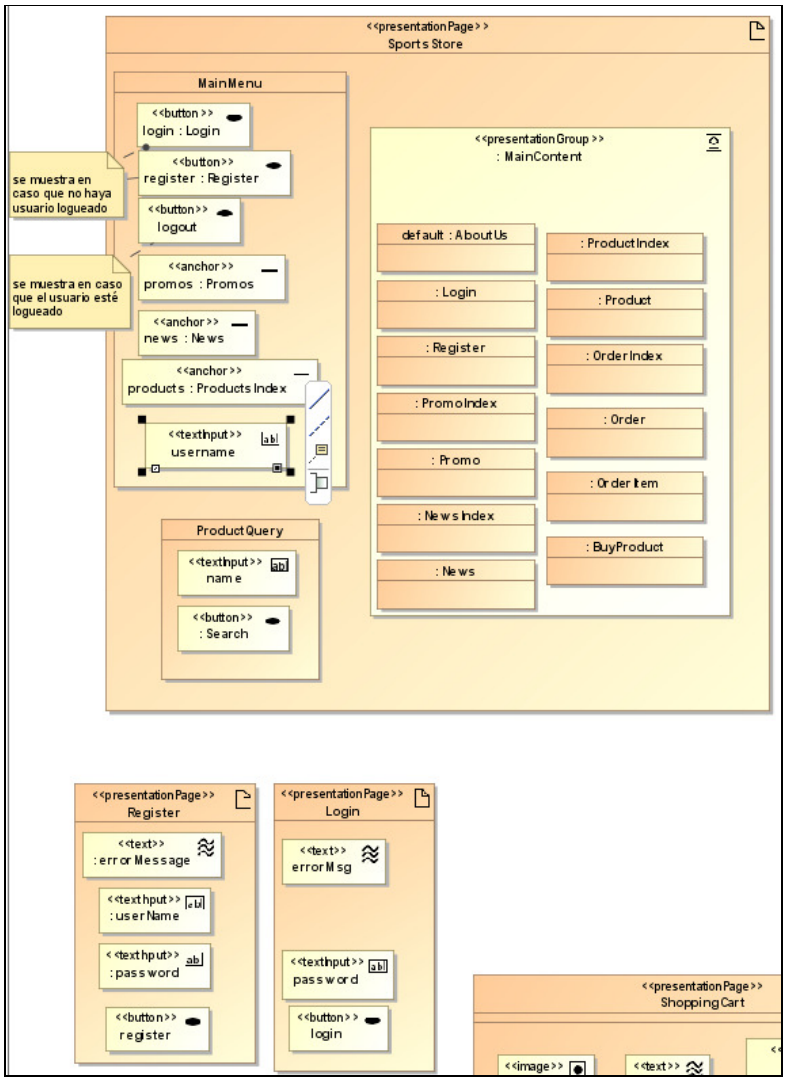


Figura 51. Move Widget - Resultado Final

6.2 Rename Page

Motivación

Se le otorga un nuevo nombre a una página determinada. Opcionalmente puede ser empleado en *Split Page* para renombrar la página refactorizada.

Mecanismo

Refactoring atómico.

Precondiciones:

a) La página a renombrar debe existir en el diagrama de presentación
b) El nuevo nombre no debe estar presente en páginas existentes.

Ejecución:

1. Seleccionar la página que queremos renombrar
2. Darle un nuevo nombre.

Implementación

CAPA DE ACTIONS

→ Clase: *DiagramContextRenamePageAction*

→ Superclase: *NMStateAction*

→ Método a extender: *actionPerformed()*

CAPA DE MODELO

→ Clase: *RenamePageRefactoring*.

→ Superclase: *PresentationModelRefactoring*

→ Método a extender: *refactor()*

Ejemplo en MagicUWE4R sobre el caso de estudio

Tomemos el mismo ejemplo que el correspondiente al refactoring *Rename Node* del modelo de navegación, donde desacoplábamos el nodo ProductListWithDetails, produciendo un nuevo nodo ProductDetails y renombrando el existente a ProductList.

Haciendo una analogía sobre el modelo de presentación, podemos suponer que tenemos una página ProductIndexWithDetails que, mediante el refactoring Split Page, es desacoplada, creando una página de nombre ProductDetails, y renombrando la existente a ProductIndex. En este último paso es donde se hace uso del *Rename Page*

Procedimiento:

PASO 1: Seleccionamos la página a renombrar

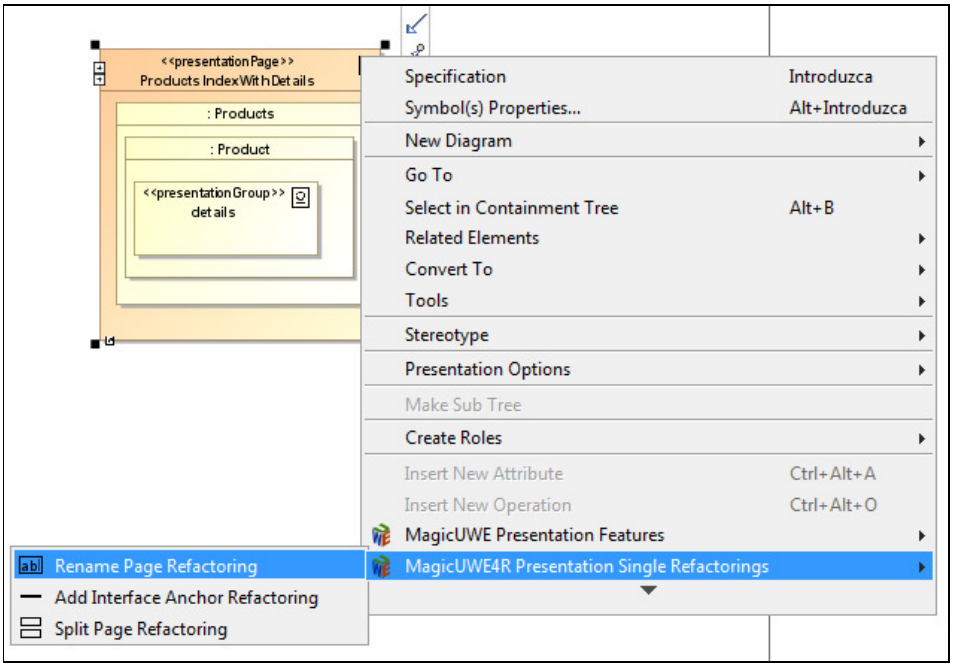


Figura 52. Rename Page - Paso 1

PASO 2: Definimos el nuevo nombre de la página

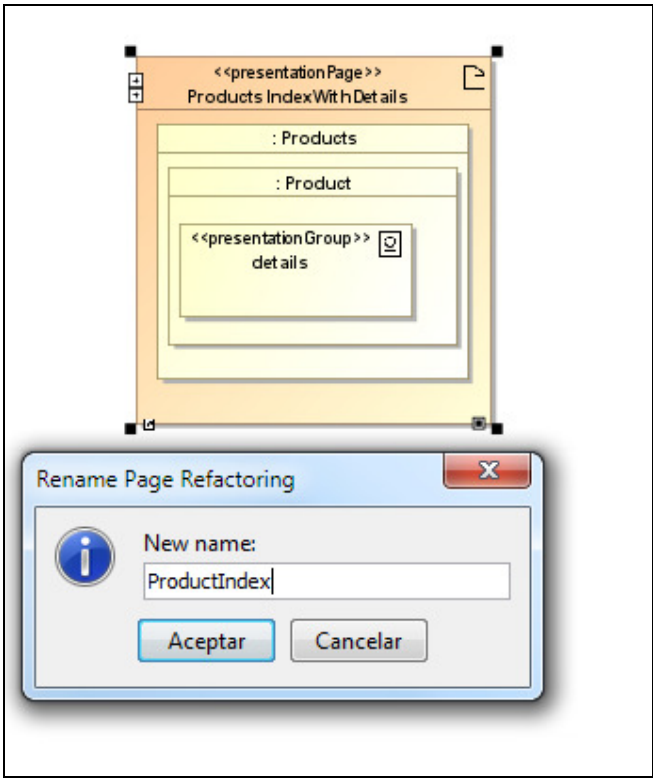


Figura 53. Rename Page - Paso 2

RESULTADO FINAL

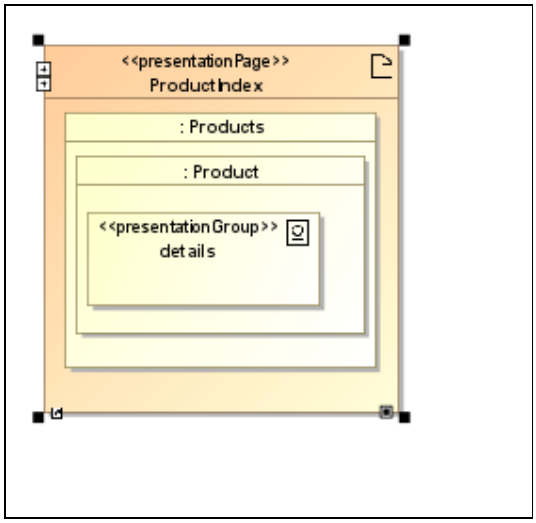


Figura 54. Rename Page - Resultado Final

6.3 Add Interface Anchor

Motivación

Este refactoring es consecuencia del *Add link* o *Turn Attribute Into Link* en el modelo de navegación, o bien cuando se necesita hacer visible un link de un nodo. También forma parte de la composición del *Split Page*

Mecanismo

Refactoring atómico.

Precondiciones:

a) El widget de tipo *anchor* (o ancla) con el nombre dado no existe en la página destino

Ejecución:

1. Identificar la página a la cual se debe agregar el widget de tipo *anchor*.
2. Agregar el widget con el estereotipo *anchor*.

Implementación

CAPA DE ACTIONS

→ Clase: *DiagramContextAddInterfaceAnchorAction*

→ Superclase: *DrawShapeDiagramAction*

→ Método a extender: *createElement()*

CAPA DE MODELO

→ Clase: *AddInterfaceAnchorRefactoring*.

→ Superclase: *PresentationModelRefactoring*

→ Método a extender: *refactorAndReturn()*

Ejemplo en MagicUWE4R sobre el caso de estudio

Como resultado del refactoring *Turn Attribute Into Link* en el modelo de navegación, tenemos en el de presentación un escenario similar. El usuario, desde la página del carrito de compras, sólo puede visualizar el nombre de los productos que ha comprado, pero es incapaz de entrar a los detalles de los productos desde el carrito.

Al aplicar el refactoring *Add Interface Anchor*, agregamos un link para cada producto del ShoppingCart, que solo contenía el nombre de los productos, para que se establezca un enlace hacia los detalles de cada uno de los mismos

Procedimiento:

PASO 1: Seleccionamos la página

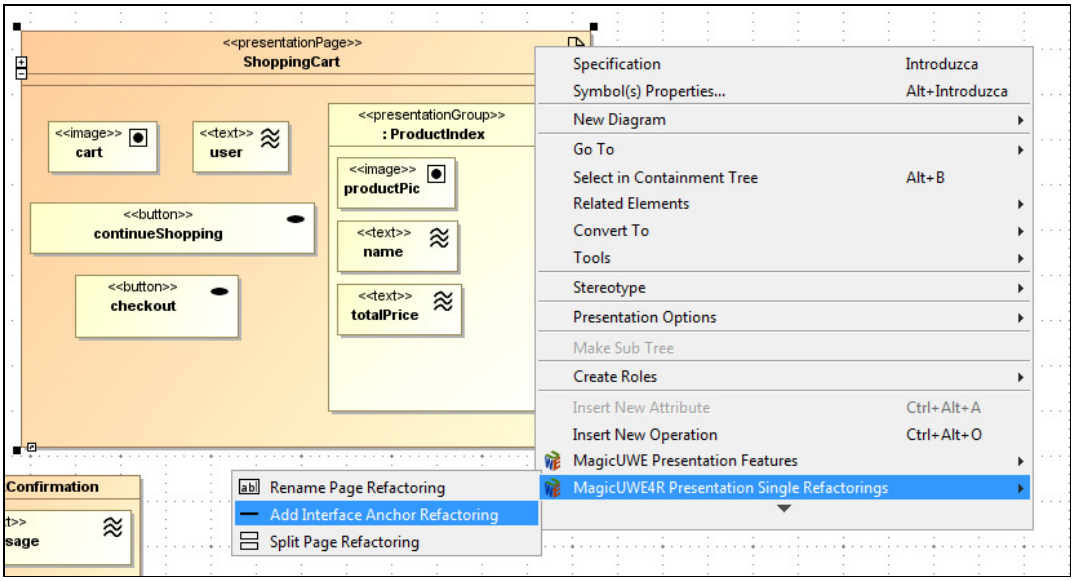


Figura 55. Add Interface Anchor

RESULTADO FINAL

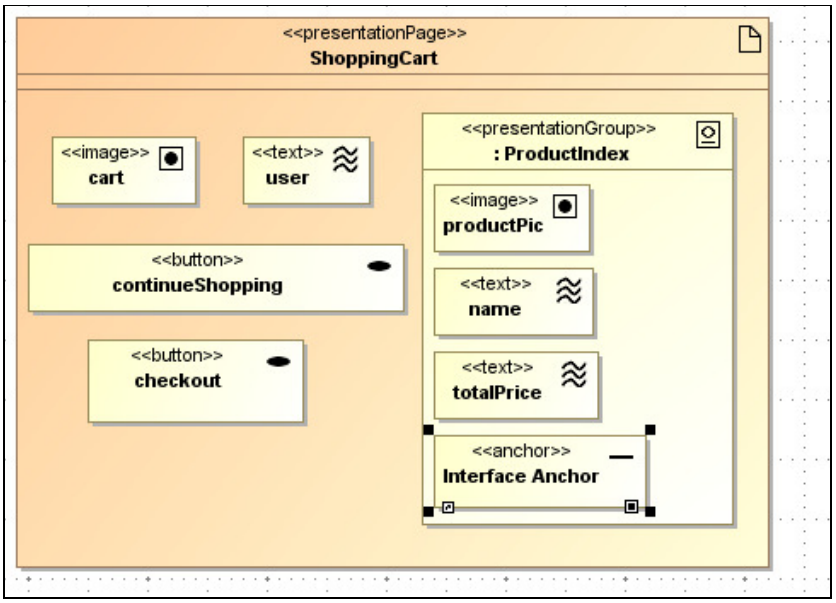


Figura 56. Add Interface Anchor - Resultado Final

6.4 Split Page

Motivación

Hay varios escenarios en los cuales es oportuno aplicar este refactoring: como consecuencia de aplicar el refactoring de modelo de navegación *Split Node Class* o debido a que una página se ha vuelto demasiado colmada de información y mostrar toda la información en la misma página hace que el usuario se pierda (por ejemplo, debido a que debe realizar demasiado scrolling).

Lo que propone este refactoring es partir o dividir la página en una o más páginas o secciones. Mejora la usabilidad, navegabilidad e interacción con el usuario.

Mecanismo

Refactoring compuesto.

Precondiciones:

a) El nombre de la nueva página debe ser distinto a cualquiera de las ya existentes en el modelo de presentación.

Ejecución:

1. Obtener la página a dividir.
2. Usar el refactoring *Move Widget* para mover los widgets seleccionados desde la página original a la nueva.
3. Usar el refactoring *Add Interface Anchor* en la página origen para enlazar las dos páginas y permitir al usuario acceder al contenido y las operaciones disponibles en la página original.
4. Verificar si es necesario cambiar el nombre de la página original y si es así cambiarlo mediante *Rename Page*.

Implementación

CAPA DE ACTIONS

→ Clase: *DiagramContextSplitPageAction*

→ Superclase: *DrawShapeDiagramAction*

→ Método a extender: *createElement()*

CAPA DE MODELO

→ Clase: *SplitPageRefactoring*.

→ Superclase: *PresentationModelRefactoring*

→ Método a extender: *refactorAndReturn()*

Ejemplo en MagicUWE4R sobre el caso de estudio

Como consecuencia de la aplicación del refactoring *Split Node Class* en el modelo de navegación, nos encontramos en un potencial caso de aplicar el presente refactoring. Tomando el ejemplo mencionado en ocasiones anteriores, nos encontramos con la página en donde vemos el listado de productos junto a los detalles de cada uno de ellos. Luego de haber aplicado el *Split Node Class* en el modelo de navegación, desacoplando el nodo ProductListWithDetails, debemos actuar en el modelo de presentación, de manera de descomprimir la página que contenía esta información, de la siguiente manera:

Procedimiento:

PASO 1: Seleccionamos la página a desacoplar

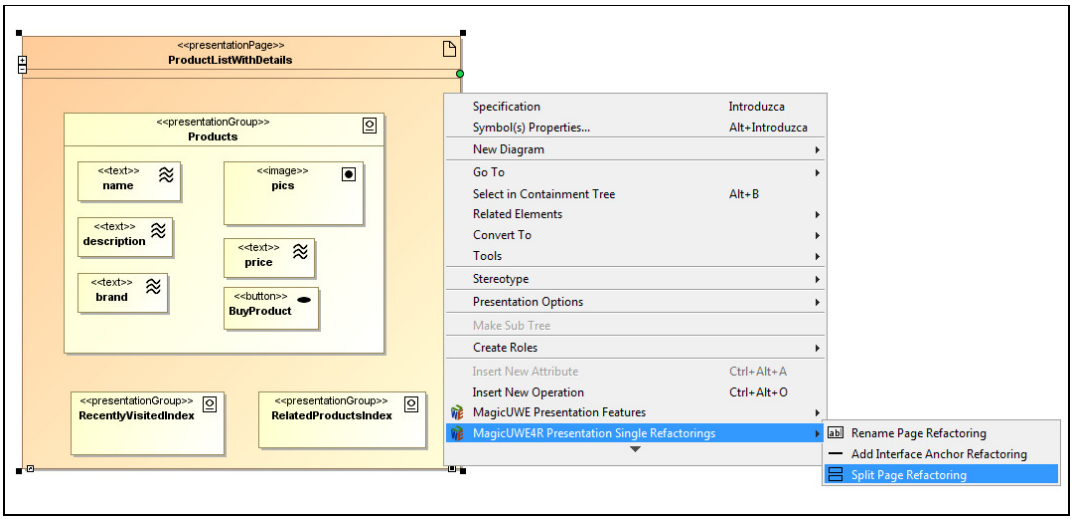


Figura 57. Split Page - Paso 1

PASO 2: Haciendo uso del Move Widget

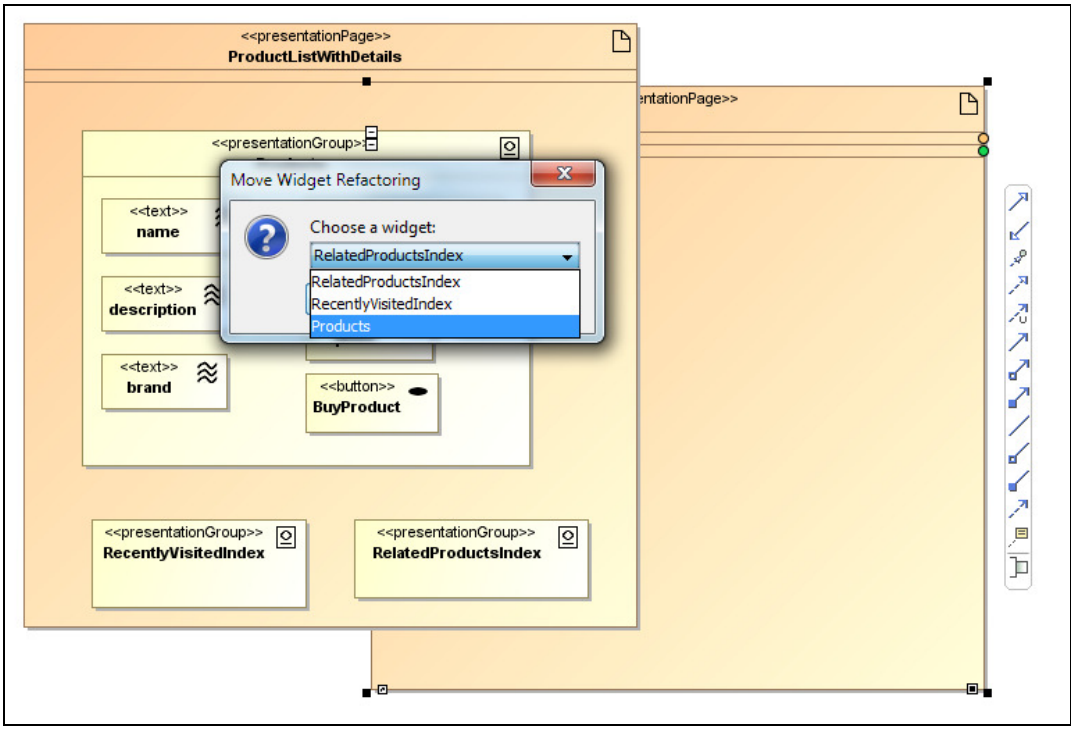


Figura 58. Split Page - Paso 2

PASO 3: Haciendo uso del Rename Page

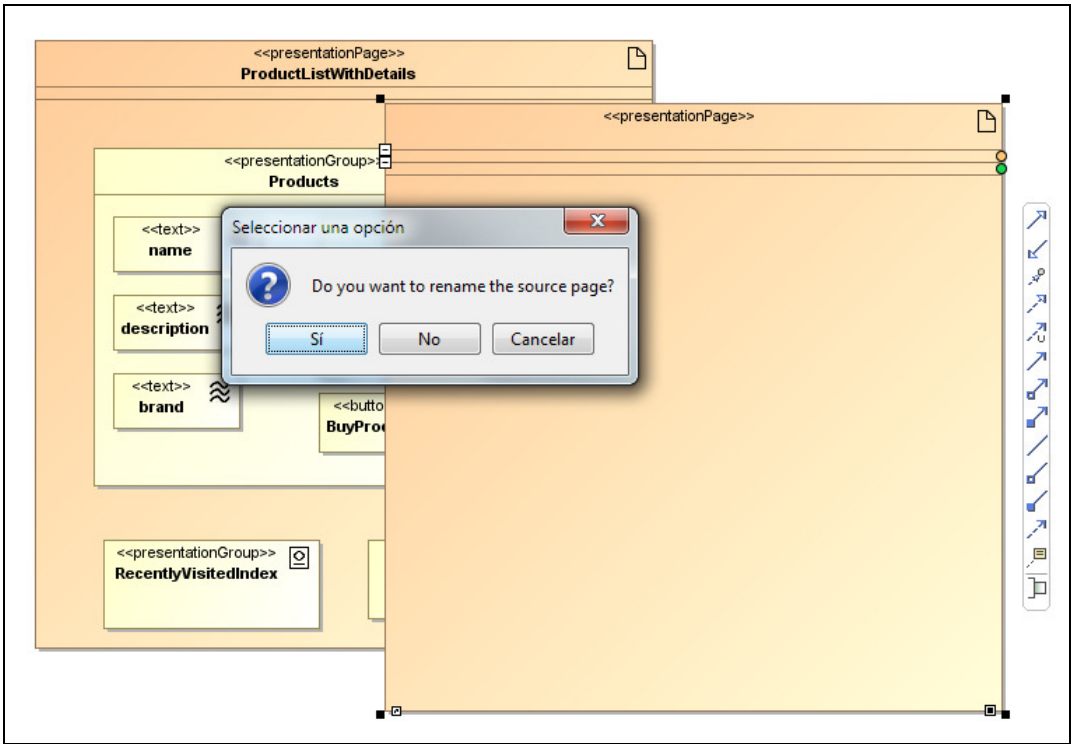


Figura 59. Split Page - Paso 3

RESULTADO FINAL

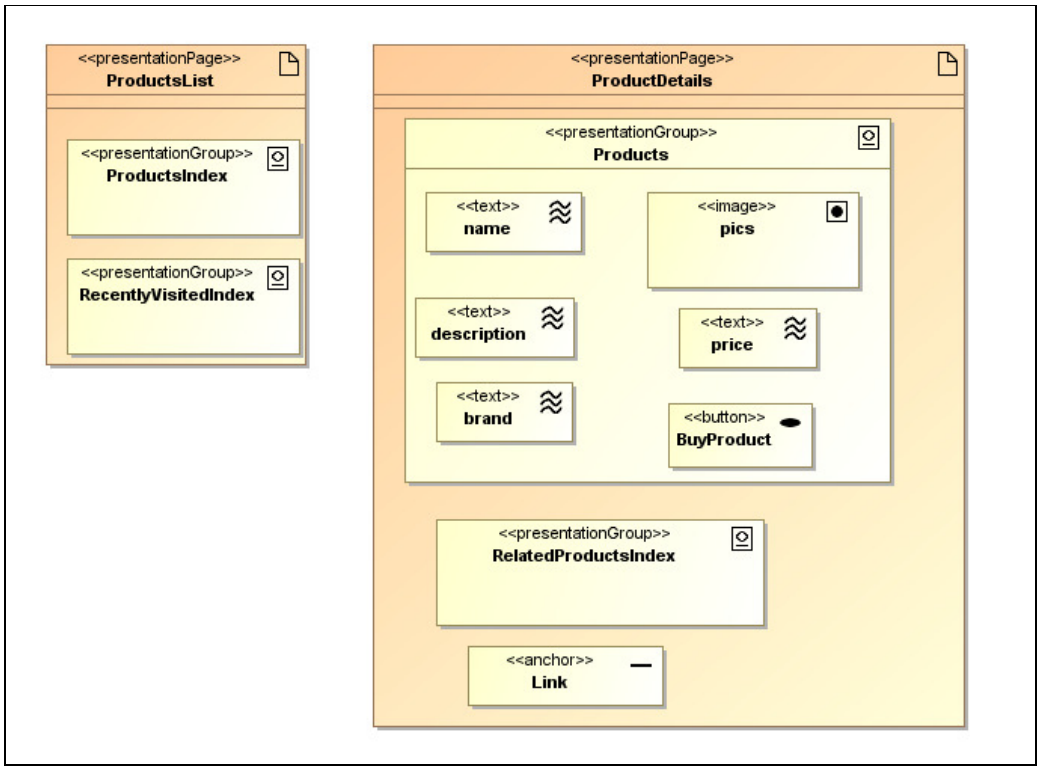


Figura 60. Split Page - Resultado Final

CAPÍTULO 7
Conclusiones y Trabajos Futuros

El presente capítulo se enfoca en aportar una pequeña reseña de lo que fue esta tesis, presentando la conclusión final, enumerando las contribuciones y limitaciones, y definiendo trabajos futuros que pueden desprenderse.

7.1 Conclusiones

El desarrollo de software dirigido por modelos (MDA) es una técnica de desarrollo de software que está cobrando mucha fuerza en los últimos años. Mediante el uso de modelos y refactorizaciones de los mismos, se pretende combatir muchos de los problemas actuales del desarrollo de software. Y el elemento clave lo conforman los modelos. Si nos basamos en la metodología de desarrollo de aplicaciones Web llamada UWE, nos referimos a los modelos conceptual, navegacional y de presentación, todos definidos en UML.

Por otro lado, tenemos el concepto siempre presente en el desarrollo de software de refactoring. A pesar que desde su concepción se le fue asociado con el código fuente, ya tenemos aportes de investigación donde se aplica a los modelos.

Sin embargo, no se suele relacionar ambos conceptos. Como si desde la raíz ambos fueran totalmente opuestos, código fuente versus modelo, precisión versus abstracción.

La idea de esta tesis fue entonces conjugar refactoring con MDD, aplicado a una metodología Web como es UWE. Es que, para que la industria adopte nuevos procesos que en la teoría suenan enriquecedores o más productivos, es necesario contar con herramientas adecuadas.

La herramienta MagicUWE4R pasa a ser el puntapié inicial en la integración de estas dos potentes metodologías. Entregable como un plugin de la aplicación MagicDraw, se acopla perfectamente al gran potencial de esta aplicación.

Basada en un catálogo de refactorings genéricos, MagicUWE4R lo adaptó a la metodología UWE e implementa un subconjunto de ellos en esta primera versión. La forma en que se diseñó y desarrolló hace que posea la capacidad de ser fácilmente

extendida y mantenible por cualquier desarrollador que tenga interés en agregar nuevos refactorings o modificar los existentes. Esto a causa de principalmente dos puntos claves: el uso de patrones de diseño y la composición de refactorings.

7.2 Contribuciones

De esta tesis se obtienen una serie de aportes:

- ✓ Visión general del desarrollo de software dirigido por modelos (MDD).
- ✓ En base al estudio del modelado de aplicaciones Web, se adquiere conocimiento sobre la metodología de interés, UWE.
- ✓ Profundización del estudio de la técnica de refactoring, y aplicación de la misma al campo de MDD.
- ✓ A causa de la necesidad de extender la herramienta MagicUWE (plugin desarrollado en Java para el software de modelado UML MagicDraw), se realiza un relevamiento de la API de MagicDraw, para implementar nuestra herramienta de refactoring MagicUWE4R.
- ✓ A partir de la combinación de la técnica de refactoring y de la metodología MDD, se analiza, estudia y adapta un catálogo de diferentes refactorings a nivel de modelo que aplicará la herramienta MagicUWE4R.
- ✓ Se presenta un análisis y definición de la arquitectura e implementación de la primera herramienta de refactoring para el desarrollo basado en modelos de una aplicación Web, llamado MagicUWE4R. Algunos puntos centrales: composición de refactorings y aplicación de patrones de diseño.
- ✓ Como la herramienta MagicUWE se puede usar también para modelar aplicaciones móviles, la herramienta desarrollada MagicUWE4R tiene un alcance incluso mayor a las aplicaciones Web [32]

A su vez, se tuvo la posibilidad de participar en las 40º Jornadas Argentinas de Informática JAIIO (http://www.40jaiio.org.ar/est). El paper presentado ("*MagicUWE4R: Una herramienta de refactoring en el modelado de aplicaciones Web*") fue aprobado y publicado en el simposio EST, obteniendo la certificación correspondiente.

7.3 Limitaciones

Durante el transcurso del desarrollo de la tesis nos encontramos con un conjunto de limitaciones que tuvimos que manejar:

- La API de MagicDraw tiene muy poca documentación, y carece de una comunidad de desarrolladores en el mundo. Por lo que la investigación del código fue mi recurso primario a la hora de buscar soluciones. Con el paso del tiempo, pude lograr un conocimiento sólido de la arquitectura, con la posibilidad de ejecutar tareas de debugging, aumentando así la productividad de manera significativa. El foro de desarrolladores de MagicDraw [33] también palió esta falta.
- En el medio del proceso, nuevas versiones de MagicUWE fueron lanzadas, por lo que se tuvo que refactorizar el código en varias ocasiones, realizando chequeo de nuevos métodos, modificaciones en clases existentes, es decir, una adaptación completa del desarrollo hasta ese momento de lo que se había hecho.
- MagicDraw es una aplicación propietaria. Fue necesario conseguir una licencia.

7.4 Trabajos Futuros

La tesis fue pasando por varias etapas diferentes: desde una investigación inicial donde muchas ramas se abrían, cada una con un conjunto de ideas particulares, pasando por la codificación de la herramienta en Java, hasta la etapa de cierre, reflejada en el presente documento.

Pero algo común a todas era que durante cada una de ellas se presentaban nuevos enfoques, oportunidades de aportes interesantes, integración con nuevos temas, etc.

Dado que es necesario realizar un acotamiento a un tema particular, y ahondar en el mismo, hubo ciertos temas que quedaron fuera del alcance de la presente tesis, y que califican como potenciales trabajos futuros. Estos son:

→ *Sincronización de una capa a otra*. Es decir, que un refactoring aplicado a un modelo, por ejemplo el de navegación, sea reflejado en otro modelo, como por ejemplo el de presentación. Se ha realizado una investigación preliminar en el trabajo [34] aunque no existe ninguna herramienta de refactoring que haga sincronización real de refactorings de una capa a otra de una arquitectura de software, dada la complejidad que ello reviste.

→ *Construcción instantánea base de una aplicación Web a partir de la generación automática de código*. Construir una implementación base de manera automática a partir de los modelos diseñados y que los refactorings también se vean a nivel de código de manera automática. Por ejemplo, implementar un modelo de dominio de clases *POJO* [35] automáticamente basado en el modelo conceptual, o un conjunto de páginas web con las reglas de navegación (sobre un framework particular como Struts) basadas en el modelo de navegación y presentación. La investigación realizada indica que existe una herramienta llamada *UWE4JSF* [36], pero ésta cuenta con la desventaja que ante un cambio en el modelo, o ante un refactoring aplicado, la aplicación debe ser regenerada nuevamente, y es éste el aspecto a mejorar.

→ Implementación de otros refactorings que describe el catálogo referenciado en [25]

Bibliografía

[1] Refactoring community. http://www.refactoring.com/

[2] Martin Fowler. Refactoring: Improving the Design of Existing code. Addison-Wesley, 2000.

[3] Kent Beck. Extreme Programming explained: Embracing Change. Reading, Mass. Addison-Wesley, 1999.

[4] Nora Koch and Andreas Kraus. The expressive power of UML based web engineering. In Proc. of 2nd Int. Workshop on Web Oriented Software Technology (IWWOST02) at ECOOP02, pages 105/119, Malaga, Spain, 2002.

[5] D. Schwabe and G. Rossi. An object oriented approach to web-based application design. Theory and Practice of Object Systems, 4(4).Wiley and Sons. 1998.

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

[7] J. Nielsen. Designing Web Usability, New Riders, 1999.

[8] D. K. van Duyne, J.A. Landay, and J.I. Hong, The Design of Sites: Patterns for Creating Winning Web Sites, 2nd ed., Prentice Hall PTR, 2006.

[9] M. Boger, T. Sturm, P. Fragemann. Refactoring Browser for UML. Proceeding NODe '02 Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World

[10] MagicDraw UML. https://www.magicdraw.com/

[11] UML® Resource Page, OMG. http://www.uml.org

[12] MagicUWE, UWE Plugin for MagicDraw.
http://uwe.pst.ifi.lmu.de/toolMagicUWE.html

[13] A. Garrido, G. Rossi, and D. Distante. Model Refactoring in Web Applications. Presented at 9th Int. Symp. on Web Site Evolution (WSE'07), 2007.

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

[15] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1

[16] J. Zhang, Y. Lin, J. Gray. Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In Volume II of Research and Practice in Software Engineering, pp. 199-218. 2005

[17] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addisson-Wesley, Reading Massachusetts, 1995.

[18] Alejandra Garrido, Gustavo Rossi, Damiano Distante. Refactoring for Usability in Web Applications. IEEE Software, vol. 28, no. 3, pp. 60-67, May/June 2011.

[19] Marianne Busch and N. Koch. MagicUWE - A CASE Tool Plugin for Modeling Web Applications. In Proc. 9th Int. Conf. Web Engineering (ICWE'09), LNCS, volume 5648, pages 505-508. ©Springer, Berlin, 2009.

[20] Object Constraint Language.
http://en.wikipedia.org/wiki/Object_Constraint_Language

[21] A. Kraus, Alexander Knapp and N. Koch. Model-Driven Generation of Web Applications in UWE. In Proc. MDWE 2007 - 3rd International Workshop on Model-Driven Web Engineering,CEUR-WS, Vol 261, 2007.

[22] MagicDraw Open API User Guide
http://www.magicdraw.com/files/manuals/MagicDraw%20OpenAPI%20UserGuide.pdf

[23] How to develop your MagicDraw plugin.
http://mdwhatever.free.fr/index.php/2010/03/how-to-develop-your-magicdraw-plugin/

[24] D. Roberts. Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana Champaign, 1999

[25] A. Garrido, G. Rossi y D. Distante. Systematic Improvement of Web Application Design. Journal of Web Engineering, 2009.

[26] Christopher Alexander, Sara Ishikawa, Murray Silverstein. A Pattern Language: Towns, Buildings, Construction. 1977

[27] Inversion of Control. http://martinfowler.com/bliki/InversionOfControl.html

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

[29] Java Development Kit.
http://www.oracle.com/technetwork/java/javase/downloads/index.html

[30] Jordi Cabot and Cristina Gomez. A catalogue of refactorings for navigation models. In Proc. Of the 8th. Int. Conference on Web Engineering: ICWE'08, Yorktown Heights, New York, 2008.

[31] G. Rossi, Mario Matias Urbieta, Jeronimo Ginzburg, D. Distante and A. Garrido. Refactoring to Rich Internet Applications. A Model-Driven approach. In Proc. of the 8th. Int. Conference on Web Engineering: ICWE'08, Yorktown Heights, New York, 2008.

[32] Challiol, Cecilia. Desarrollo Dirigido por Modelos de Aplicaciones de Hipermedia Móvil. Tesis Doctoral. Facultad de Informática, Universidad Nacional de La Plata.2011.

[33] No Magic Community Forum. https://community.nomagic.com

[34] Daniel Ruiz-González, Nora Koch, Christian Kroiss, José-Raúl Romero, and Antonio Vallecillo. Viewpoint Synchronization of UWE Models. In Proc. MDWE 2009 - 5rd International Workshop on Model-Driven Web Engineering, CEUR-WS, Vol 455, June 2009.

[35] Martin Fowler on POJOs. http://www.martinfowler.com/bliki/POJO.html

[36] Christian Kroiss and Nora Koch. UWE4JSF - A Model-Driven Generation Approach for Web Applications. In Proc. 9th Int. Conf. Web Engineering (ICWE'09), LNCS, volume 5648, pages 493-496. ©Springer, Berlin, 2009.

[37] G. Rossi, Daniel Schwabe, and A. Garrido, Design reuse in hypermedia applications development. In proceedings of Hypertext ’97. Southampton, UK, 1997.

[38] D.Van Duyne, J. Landay, and J. Hong. The Design of Sites. Addison-Wesley, 2003.

[39] Christian Kroiß and N. Koch. UWE Metamodel and Profile: User Guide and Reference. LMU Technical Report, 2008.

[40] Mark Grand. Patterns in Java, Volume 1 (Second Edition), John Wiley & Sons, 2002.