MagicUWE4R: Una herramienta de refactoring en el modelado de aplicaciones Web

MIGUEL DJEBAILE

Fac. de Informática, Universidad Nacional de La Plata
50 y 120, La Plata, CP 1900, Buenos Aires, Argentina
migueldje@gmail.com

Resumen. MagicUWE4R incluye 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 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. A su vez, se pone énfasis en desarrollar los refactorings empleando patrones de diseño, y en unidades atómicas, de manera que puedan componerse, para que el motor de refactoring 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.

Keywords: Refactoring, Model Driven Development, Patrones de diseño, Metodologías ágiles, UWE

1 Introducción

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 muchos casos, 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 este caso, 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** 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."

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.

Por otro lado, las metodologías de desarrollo de aplicaciones Web existentes [1], como por Ej. UWE, UWA, OOHDM, WebML, son "model-driven", es decir metodologías que siguen el desarrollo dirigido por modelos (Model Driven Development - **MDD**). 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.

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**.

Dada la importancia en la tendencia actual de permanente cambio y adaptabilidad en las aplicaciones Web, es deseable poder contar con una herramienta de software que permita plasmar de manera automática los refactorings.

Lo que propone este paper es utilizar refactoring, una práctica que se asocia a las metodologías ágiles, dentro del proceso de desarrollo de una metodología dirigida por modelos como son las metodologías Web. En particular nos centraremos en refactorings sobre los modelos de la metodología UWE.

UWE no da soporte a refactoring, por lo que fue necesario implementar una herramienta, para así poder emplear metodologías Web, Model - Driven, y soportando refactoring de modelos.

2 Trabajos Relacionados

2.1 Model Driven Architecture (MDA)

Model-Driven Architecture (MDA) es una iniciativa de OMG. Es una implementación de MDD (Model Driven Development) [5]. MDD no define tecnologías, herramientas, procesos o secuencia de pasos a seguir. La implementación MDA se ocupa de ello mediante el uso de un lenguaje de modelado UML y meta-pasos a seguir en el desarrollo de sistemas. MDA se basa en la construcción y transformación de modelos.

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

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

2.2 Refactoring

Martin Fowler define al término Refactoring como 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 [3].

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, más difícil de ver el diseño en el código, más difícil es preservarlo y más rápidamente decae. 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 para hacer la misma cosa, 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 cosas sobre el diseño que antes no se observaban, por lo tanto se pueden detectar errores más claramente.
- Ayuda a desarrollar código más rápidamente, ya que los buenos diseños ayudan a desarrollar más rápidamente código al no tener que perder demasiado tiempo detectando y depurando errores.

2.3 Refactoring de modelos

Se define refactoring de modelo como el 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. Es una propuesta transformacional para el desarrollo de software iterativo. 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

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

- se basa en la aplicación de reglas para el refactoring de modelos,

- 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,

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 UWE

UWE (UML-Based Web Engineering) [2] 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. Los sistemas adaptativos y la sistematización son dos aspectos sobre los que se enfoca UWE.

UWE hace un uso exclusivo de estándares reconocidos como UML y el lenguaje de especificación de restricciones asociado OCL. Para simplificar la captura de las necesidades de las aplicaciones Web, UWE propone un proceso dividido en tres pasos o actividades:

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 esta relacionado internamente. Es decir, cómo se enlazan los elementos de navegación. Para ello, se emplean unidades de navegación llamadas “nodos” conectadas por enlaces de navegación.

Modelo de Presentación: Representa las vistas del interfaz del usuario mediante modelos estándares de interacción UML.

3 Arquitectura base

3.1 Caso de estudio: MagicDraw Open API

MagicDraw es una completa aplicación para el modelado 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 y base de datos orientadas a objetos (OO).

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.

3.2 Extensibilidad: Plugins

La forma en que MagicDraw nos ofrece extensibilidad es mediante su Java 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.

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.

3.3 MagicUWE

Hasta ahora 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 [7].

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.

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 lograr que el diseño de aplicaciones Web con UWE y MagicDraw sea un proceso simple.

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

4 Arquitectura de MagicUWE4R

4.1 Arquitectura

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

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

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 refactoring en el código

Como se mencionó previamente, 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 con otros refactoring más complejos. Es decir, poder tener la posibilidad de crear un nuevo refactoring, más complejo, a partir de la composición de refactorings más sencillos ya existentes.

Veamos un ejemplo para clarificar el concepto: existe un refactoring complejo, *Split Node Class*, que se aboca a desacoplar un nodo repleto de información o saturado de funcionalidades ajena a él.

Este refactoring sigue los siguientes cuatro 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 la clase del nodo origen y el nuevo nodo para permitir al usuario poder acceder la información original y su conjunto de operaciones.
5. En caso opcional, renombrar el nodo origen

Podemos ver que en el punto 2 y 3 podemos hacer uso de refactorings atómicos (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*

Vemos 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 template method como 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.

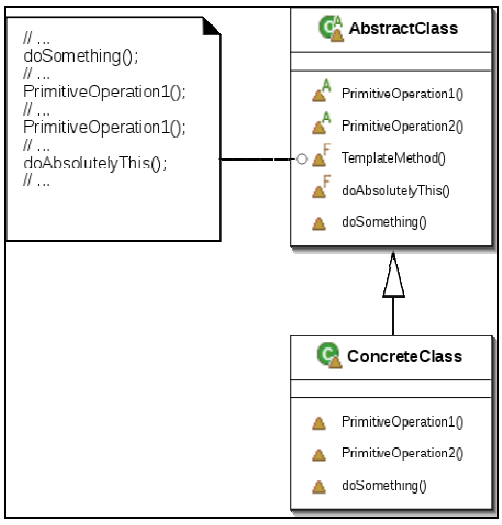

Figura 1. Estructura del patrón Template Method

Como refleja la Figura 1, 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.

Y una clase concreta, cuya función es implementar las operaciones primitivas para determinar el comportamiento específico de la clase concreta

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.

Veamos cómo se aplica esto a MagicUWE4R. 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:

- Se abre la sesión
- Se chequean las precondiciones
- Si las precondiciones son cumplidas, se hace el refactoring
- Se cierra la sesión.

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 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.

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). 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. Podemos ver su estructura representada en un diagrama de clases UML [6] en la Figura 2

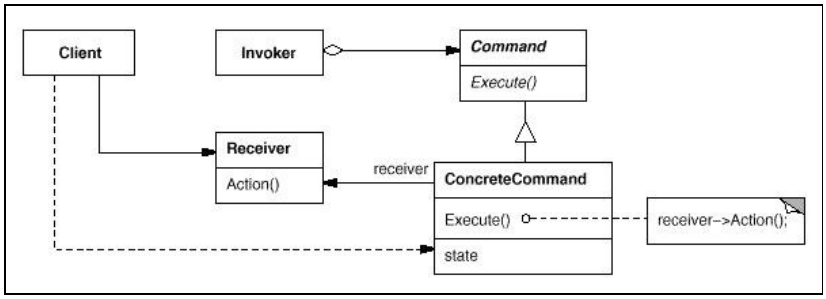

Figura 2. Estructura del patrón Command

El hecho de poder encapsular las acciones en objetos, 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.

Hablando ahora particularmente de MagicUWE4R, la arquitectura de Actions representa una implementación del patrón Command. 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 de Swing, 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.

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

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.

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 1. Métodos de la Clase Visitor

A su vez, encontramos en las clases como BaseElement, DiagramPresentationElement, PathConnector, PathElement, ShapeElement el método: public void **accept**(Visitor visitor) throws Exception

4.4 Extendiendo MagicUWE4R

Como dijimos anteriormente, nuestro plugin fue desarrollado en Java. Fue compilado con la JDK 1.6 y se empleó como IDE de desarrollo Eclipse.

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):

- Clase **magicUWE.core.PluginManager.java**: representa el Plugin Manager (ver sección 3.2). 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.

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

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

5 Refactorings en el modelo de navegación

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.
- Estructuras de acceso (índices).

5.1 Add Node Operation

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.

Implementación:

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

5.2 Move Node Attribute

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.

Implementación:

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.

5.3 Move Node Operation

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

Implementación:

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.

5.4 Rename Node

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

Implementación:

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

5.5 Add Link

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*.

Implementación:

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

5.6 Split Node Class

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/o métodos 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.

Implementación:

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 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*.

5.7 Turn Attribute Into Link

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.

Implementación:

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.

6 Refactorings en el modelo de presentación

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.

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 de aplicación lo soporten.
- Cambiar los efectos de la interfaz

A continuación se describen los refactorings del modelo de presentación implementados en MagicUWE4R.

6.1 Move Widget

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.
Implementación:

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.

6.2 Rename Page

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

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

6.3 Add Interface Anchor

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*
Implementación:

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

6.4 Split Page

Hay varios escenarios en los cuales es oportuno aplicar este refactoring: como consecuencia de aplicar el refactoring de modelo de navegación *Split Node* 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.
Implementación:

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 aplicar *Rename Page* de la página origen.

Conclusión

En el presente trabajo, introducimos una herramienta cuyo aporte más significativo fue la de incluir la práctica de refactoring dentro de una metodología de desarrollo de aplicaciones Web existente como UWE, que da soporte a la filosofía MDD. Es decir, hacemos uso de una serie de conceptos que se suelen manipular de manera separada, y los combinamos para obtener un avance significativo y que hace más abarcativa la integración de los mismos.

Otro gran valor que creemos aporta el trabajo es el haberle puesto hincapié al aspecto interno del proyecto, a la construcción del mismo, haciendo foco en buenas prácticas de diseño. De esta manera, mediante la aplicación de patrones de diseño aceptados y ampliamente empleados por la comunidad de objetos, se logró construir una herramienta cuyo motor de refactorings es por demás adaptable, extensible y escalable.

Referencias bibliográficas

1. D. Schwabe y G. Rossi. “An Object Oriented Approach to Web-Based Application Design”. *Theory and Practice of Object Systems* 4(4), Wiley and Sons, 1998.
2. N. Koch and A. Kraus. “The expressive power of UML-based web engineering”. In Proc. of 2nd Int. Workshop on Web Oriented Software Technology (IWWOST02) at ECOOP02, Malaga, Spain, 2002.
3. M. Fowler. “*Refactoring Improving the Design of Existing Code”*. Addison Wesley. 1999.
4. A. Garrido, G. Rossi y D. Distante. “Systematic Improvement of Web Application Design”. *Journal of Web Engineering*, 8 (4), 2009.
5. S. J. Mellor, A. N. Clark y T. Futagami. “Model-Driven Development”. *IEEE Software*, Septiembre-Octubre 2003.
6. UML Resource Page, OMG, http://www.uml.org/
7. MagicUWE. Resource Page: http://uwe.pst.ifi.lmu.de/toolMagicUWE.html