Manejando la evolución de aplicaciones web con adaptación

Magíster en Ingeniería de Software

Facultad de Informática

Universidad Nacional de La Plata

Autor: **Pablo Federico Solarz**
Director: **Dr. Gustavo Rossi**

noviembre de 2003

Índice temático

Apéndices

1 Introducción

1.1 Aplicaciones y evolución

La complejidad es una característica inherente al software desde sus orígenes, e igualmente característico es que dicha complejidad crece en consonancia con el universo de roles que desempeña en la actualidad. Permanentemente aparecen nuevas ideas, metodologías y herramientas para hacerle frente, que atacan diferentes aspectos del ciclo de vida de desarrollo de software, o más aún, los enfoques del ciclo mismo.

Gran parte de la complejidad tiene que ver con la necesidad de estar preparado para adecuarse permanentemente a nuevos requerimientos. Reusabilidad, encapsulamiento, *binding* dinámico, han sido respuestas con la que el paradigma de objetos ha respondido a esta problemática y junto a estas, la separación de responsabilidades de un conjunto de parte que cooperan entre si, tiene un lugar destacado como lo demuestra el *pattern* modelo vista controlador originado en Smalltalk separando y desacoplando tareas de interfase y del modelo de la aplicación.

Más allá de interfases y aplicaciones, la separación de responsabilidades y el desacoplamiento entre clases, desde un punto de vista más general, es abordada por una serie de *design patterns* así como su contraparte necesaria, el aspecto cooperativo entre las mismas. En este último aspecto, el pattern Observer aborda la problemática de un conjunto indeterminado de objetos observadores que requieren ser notificados y actualizados respecto de cambios de estado de otro objeto determinado. El pattern permite la evolución de los observadores, incluso el agregado o eliminación de algunos de ellos sin interferir en el objeto observado, lo que permite disminuir el impacto que cambios de una parte del software, tales como vistas u observadores pueden requerir sobre otra, tales como el modelo o elementos observados.

Como cualquier disciplina, el software, y particularmente su diseño, evoluciona a medida que se sistematizan aspectos del mismo. Los design patterns constituyen un hito sustancial en la sistematización de soluciones a problemas que se repiten en el diseño de diversas aplicaciones. Sobre este hito se desarrolla otro, el refactoring o reestructuración de programas orientados a objeto, que tiene por objetivo introducir en forma automática mejoras, en especial design patterns en el código, sin modificar su comportamiento. La reestructuración es un paso previo a la introducción de nuevas prestaciones del software existente.

1.2 Aplicaciones en la web

Internet ha ocupado un lugar destacado en los últimos años, por su crecimiento, y por el interés industrial, comercial y científico despertado, entre los impulsores de desarrollos y búsquedas de soluciones a tradicionales y nuevas problemáticas del software. Organizaciones de distinto tipo, como instituciones científicas, técnicas y gubernamentales, empresas comerciales, industriales y de servicios migran a sistemas de información basados en la web. Dentro de esta diversidad, las aplicaciones más populares son las relacionadas con comercio electrónico y subastas. Otras de gran importancia tienen que ver con actividades colaborativas, integración de información y conocimiento originalmente distribuido.

Todas estas, se acercan a las aplicaciones complejas tradicionales y distan considerablemente de las características de los sitios web iniciales en cuanto a que van mucho más allá de la observación de información estática, y de la realización de transacciones simples como las de ingreso de datos desde formularios. Pero por otra parte adoptan las características “omnipresentes” de internet, las potencialidades de navegación y presentación de diferente tipo de información hipermedial de la web.

Se han desarrollado herramientas y fundamentalmente metodologías de diseño de aplicaciones web que abordan la tarea de unir ambos paradigmas.

Esta síntesis de ambos tipos de aplicaciones, denominada aplicaciones web, donde confluyen algún paradigma tradicional, especialmente el orientado a objetos, con la navegación hipermedia, no es de todas maneras trivial, especialmente cuando se incorpora el ingrediente de instanciaciones relacionadas con la navegación y de usuarios con perfiles de características diversas.

Hay métodos de diseño de aplicaciones web que hacen eco de la idea de separar responsabilidades y establecer un esquema de colaboración al estilo del pattern Observer. Particularmente OOHDM (Object Oriented Hipermedia Design Method) considera las aplicaciones Web como vistas navegables del modelo del dominio de la aplicación, permitiendo la existencia, por un lado de un modelo conceptual orientado a las abstracciones que surgen del dominio, y por el otro, de un modelo orientado a proveer objetos que permitan distintas vistas navegables del dominio según el observador, llamado modelo navegacional, potenciando así la reusabilidad. Las definiciones de estos objetos navegacionales, están basadas en las clases, sus atributos, métodos y asociaciones del modelo conceptual.

1.3 Adaptación

Sin embargo, se plantea un problema inverso: los observadores tienen una dependencia elevada de los elementos observados, ya que la observación se basa en

referencias a estos. Esta situación plantea entonces potenciales problemas de mantenimiento cuando lo que se modifica es el elemento observado y existen gran cantidad de observadores, cuyas referencias al elemento observado deben adecuarse.

La propuesta del presente trabajo tiene por objetivo minimizar el mantenimiento del conjunto de vistas que componen el modelo navegacional (o modelo observador) ante cambios en el modelo conceptual (o modelo observado). Se compone de dos métodos que se pueden utilizar en forma independiente o complementaria. Ambos se basan en el concepto OOHDM de que los objetos navegacionales en cualquier vista, están definidos sobre elementos del esquema de clases del modelo conceptual, o sea, en sus atributos y métodos visibles externamente, y hacen referencia a la firma de los mismos, y no a la definición de los métodos.

Un método propone la utilización de definiciones de miembros de las clases del modelo navegacional en forma genérica, es decir a partir de grupos de miembros de clases del modelo conceptual que tienen alguna característica en común. Este agrupamiento se implementa haciendo uso de los pares *Tag/Value* del modelo de extensión UML, con lo que el miembro navegacional se define referenciando al *Tag* y no al atributo u operación conceptual. Así, cambios en estos últimos, como agregados o eliminaciones, no afectan a la clase observadora a la vez que las instancias se mantienen actualizadas. El uso de este método de definición es opcional.

El otro método actúa en tiempo de diseño, redefiniendo los componentes del modelo navegacional ante cambios del modelo conceptual. Las modificaciones en el esquema de clases del modelo conceptual son analizadas para determinar si el elemento modificado es parte de la definición de uno o más nodos (clases del modelo navegacional) pertenecientes a alguna vista, en cuyo caso, un mapeo mediado por condicionantes que tienen que ver con el contexto del elemento nodal, determina el cambio correspondiente a producir en la definición de cada nodo, que es entonces lo que se adapta a la nueva versión del esquema de clases.

Para ejemplificar el uso de ambos métodos se analiza su aplicación en respuesta a cambios de diferente grado de complejidad en el modelo conceptual, desde renombrar una variable hasta la evolución por etapas de un esquema de clases, utilizándose aquellas modificaciones ya formalizadas como *operaciones de refactorización o refactoring*.

La sistematización de los cambios en el modelo conceptual es un requisito indispensable en este segundo método, ya que se mapean cambios del modelo conceptual al navegacional. Ante cada operación de cambio en el modelo conceptual hay predefinida una operación de mapeo al modelo navegacional que detecta en el conjunto de clases navegacionales la presencia de referencias al modelo conceptual modificado. De esta

manera el método es extensible respecto de nuevas operaciones de *refactoring,* siendo necesario definir la correspondiente operación de mapeo.

Hay dos aspectos de la mantenibilidad que son los objetivos del método propuesto:

1. Eliminar inconsistencias en las vistas cuando hay variaciones en el modelo conceptual.
2. Seguir desde el modelo de navegación, evoluciones del modelo conceptual.

Respecto del segundo punto, cabe señalar que el diseño del modelo de navegación está basado en la relación de los diferentes perfiles de usuario respecto de la aplicación. Si las modificaciones en la aplicación requieren de una nueva tarea de análisis de perfiles de usuario respecto a su punto de vista de la o parte de la misma, entonces escapa al método propuesto.

En el resto del trabajo se utilizan indistintamente los términos *operaciones de refactoring* u *operaciones de refactorización*.

Las figuras que muestran las clases que operan las transformaciones se muestran con marco doble, a diferencia del efecto de la operación en sí, cuyos esquemas se muestran sin marco.

1.4 Otras contribuciones del presente trabajo

- Junto al modelo de adaptación de vistas (u observadores) señalado, se presenta un modelo de actualización de las asociaciones del esquema de clases del modelo conceptual, en correspondencia con las modificaciones producidas en las definiciones de las clases. Esto permite que el esquema se mantenga en correspondencia con el código cada vez que una operación de *refactoring* modifica este último, aunque la motivación principal se encuentra en que la definición de las vistas navegacionales se basa en dichas asociaciones.
- La definición de asociación usada incorpora a la definición UML (Unified Modeling Languaje) la referencia por la cual una clase está asociada con otra, lo que es condición necesaria para mantener las definiciones de las clases navegacionales en correspondencia con el código con el que se definen las clases del modelo conceptual.
- El modelo de adaptación incluye una reformulación de las operaciones de refactoring, al encapsular cada una en una clase que además, se compone con aquellas que tienen la responsabilidad de actualizar las asociaciones del modelo conceptual y las definiciones de las clases navegacionales.
- Se reformula la definición de nodos y vínculos (*links*) del modelo OOHDM, cambiando el lenguaje de consulta de objetos por la cadena (o path) de asociaciones de clases del modelo conceptual. Esta reformulación, permite expresar las definiciones de las clases navegacionales completamente en términos del

metamodelo UML o alternativamente mediante la sintaxis con la que cada lenguaje orientado a objetos representa las asociaciones.

1.5 Estructura del presente trabajo

El resto de la presentación se reparte de la manera que sigue:

En la segunda parte se hace una breve introducción acerca de modelos de diseño de las aplicaciones para la web centrando la atención en los componentes del modelo OOHDM

En la tercera parte se propone una reformulación de algunos componentes OOHDM e incorporan otros, los que serán elementos básicos del modelo de clases navegacionales.

En la cuarta parte se trata el tema de la evolución de SW, desde la base sistematizada provista por algunos autores de operaciones de refactoring. En este capítulo se plantea una nueva óptica de estas operaciones desde una modelo orientado a objetos que permitirá luego, extender su alcance hacia el modelo navegacional.

En la quinta parte se resumen distintos abordajes de la adaptación de SW, surgidos a la vez de distintas motivaciones. Se introduce además el concepto y estrategia de adaptación usado en el presente trabajo.

El modelo de adaptación de las definiciones de las clases navegacionales, ante modificaciones de las clases que componen el modelo conceptual, se desarrolla en la sexta parte.

En la parte séptima se evoluciona un esquema conceptual, y en base al modelo del capítulo anterior se mapean los cambios producidos en el modelo navegacional.

Finalmente, en las conclusiones se discute acerca del alcance del método, así como posibles derivaciones del mismo.

2 Modelos de diseño de aplicaciones para la web

2.1 Caracterización de aplicaciones en la web

Las aplicaciones en la web han evolucionado desde la navegabilidad mediante vínculos de hipertexto, de solo lectura, pasando luego a incorporar presentaciones de formularios para carga de datos por parte de los clientes. Paralelamente se fueron agregando características hipermediales en forma masiva a medida que la tecnología lo permitía. Actualmente conviven las anteriores con aplicaciones de negocios donde convergen diferentes perfiles de usuario con roles activos y la realización de complejas transacciones.

La creciente demanda por parte de todo tipo de organizaciones respecto de este tipo de aplicaciones se basa en características tales como ubicuidad, servicios, facilidades de acceso a la información, formatos de información disponibles, estándares abiertos, etc. Esta demanda ha impulsado la aparición en el mercado de herramientas y métodos que potencien el desarrollo de este tipo de aplicaciones.

Por otra parte, la complejidad señalada anteriormente también ha impulsado, desde diferentes líneas, abordajes sistemáticos y metodológicas tanto en aspectos de análisis como de diseño, que independientemente del uso posterior de herramientas y plataformas permiten expresar el desarrollo de aplicaciones web en términos de modelos abstractos, pero a la vez representativos de los elementos componentes de estas aplicaciones.

Un método de uso extendido para abordar la complejidad de las aplicaciones en general, e incorporado al diseño de aplicaciones para la web, es la separación de responsabilidades, que recibió un gran impulso desde el desarrollo de la tecnología OO y la aparición del MVC en Smalltalk, separando y desacoplando las tareas de interfase y las de la aplicación.

En cuanto a las aplicaciones para la web (o simplemente aplicaciones web), una orientación desarrollada por varios autores es separar lo concerniente a la lógica del dominio de la lógica de navegación, y esta a su vez de los aspectos de presentación [GOM 01], [CER 00], [BAR 00], [CON 99], [SCH 98]. Esta separación permite sacar provecho tanto de las tecnologías de software tradicionales, particularmente la orientada a objetos y sus mecanismos de abstracción, como de las orientadas a hipermedia.

2.2 OOHDM

Una generalización altamente difundida de la idea de MVC es el pattern Observer [GAM 95], donde una cantidad independiente de observadores son notificados cuando el sujeto observado cambia de estado, siendo esta, la notificación, la única relación entre ambos.

Parte de las metodologías de diseño desarrolladas en base a la separación de responsabilidades, ha abordado además, haciendo uso de la idea del pattern Observer, la multiplicidad de perfiles de usuario. A partir de un único dominio de aplicación y un modelo único de dominio, diferentes usuarios tienen diferentes observaciones acuerdo a la característica de observador que representan.

En particular, OOHDM [ROS 96], [SCH 98] considera las aplicaciones web como vistas navegacionales u observadores de un modelo representativo del dominio de la aplicación o modelo conceptual. El conjunto de vistas navegacionales conforman el modelo navegacional de la aplicación como se muestra en la figura 1. Cada vista aborda un aspecto particular de la aplicación, en general, diseñada desde la óptica de un perfil de usuario determinado.

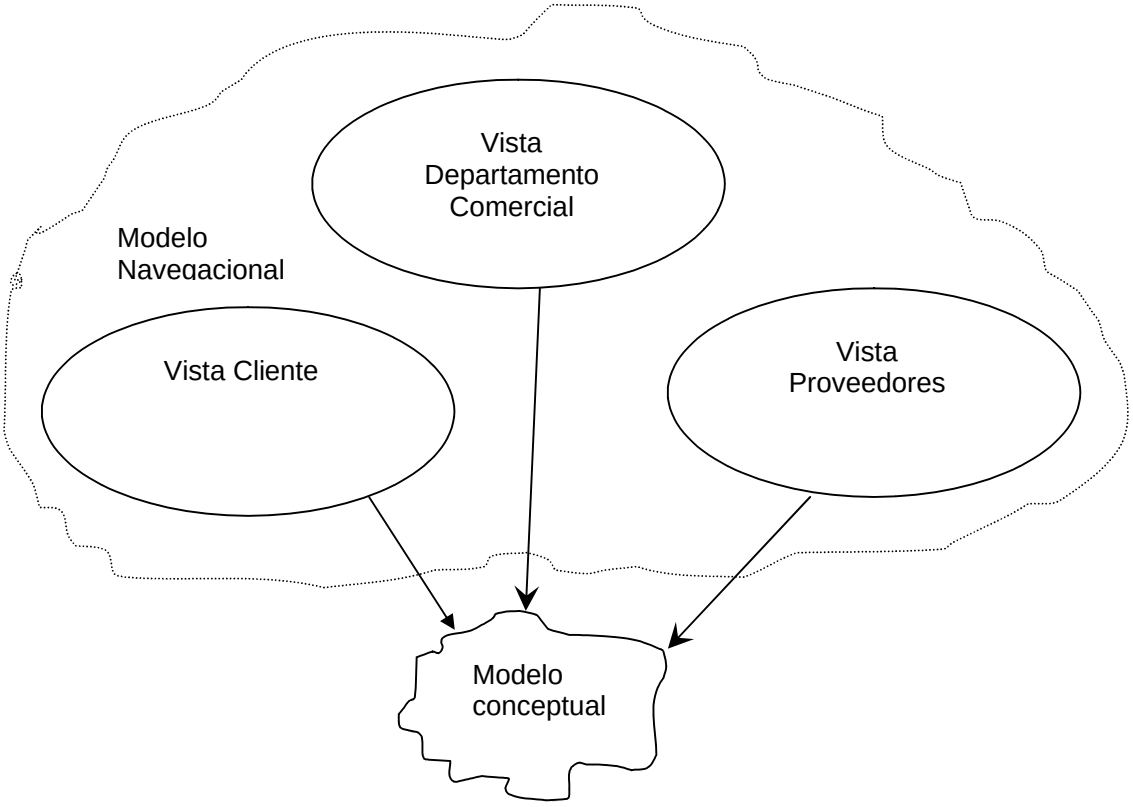


Figura 1: vistas navegacionales de un modelo conceptual.

Ambos modelos, el navegacional y el conceptual, se desarrollan en base al paradigma de objetos y se representan haciendo uso del lenguaje de modelado UML (Unified Modeling Language). Los aspectos propios de interfase de usuario son abordados en otra etapa, con herramientas adecuadas al efecto. Este último aspecto escapa al objeto del presente trabajo.

El modelo conceptual, aborda la semántica del dominio de la aplicación, sin preocuparse por caracterizaciones acerca del perfil de usuario. Está construido en base a elementos conocidos de modelado OO tales como clases, relaciones y mecanismos de abstracción tales como generalización/especialización y agregación. En la figura 2 se muestran un par de clases representando parte de un esquema conceptual.

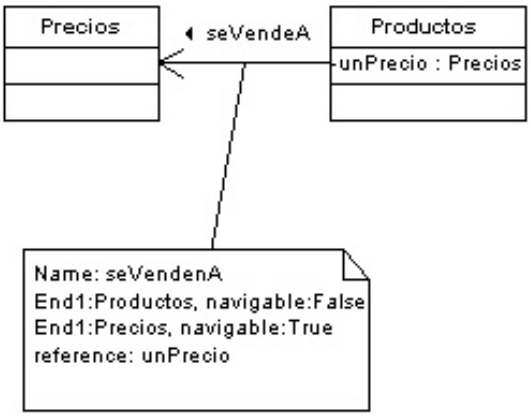


Figura 2: Clases y asociación conceptual.

El modelo navegacional es desarrollado según cada perfil de usuario, como un conjunto de observadores del modelo conceptual. Está compuesto de clases navegacionales denominadas nodos, con atributos y operaciones que caracterizan sus estados posibles y su comportamiento ante mensajes, tal como ocurre con las clases conceptuales.

Un nodo se define como observador de una clase del modelo conceptual desde donde, asociaciones mediante, puede observar atributos de otras clases conceptuales relacionadas con la primera. La representación estereotípica generalizada es similar a una clase conceptual, con la particularidad de una barra vertical a la derecha del nombre. En este trabajo se lo representa con <<*nodo*>> sobre el nombre del nodo.

La navegación entre nodos se realiza recorriendo los vínculos (*links*), llamados también relaciones navegacionales. Estos provienen, en general, de asociaciones presentes en el esquema conceptual, aunque son vínculos solo aquellos que permiten navegar entre nodos.

Complementariamente, un ancla o un índice debe ser atributo de la clase navegacional origen, desde donde es posible recorrer el vínculo para desembocar en el nodo destino [ROS 96], [SCH 98]. Los elementos del modelo navegacional siguen la semántica UML, con algunas particularidades particularidades. [SCH 99] es un trabajo dedicado particularmente a la nomenclatura OOHDM.

En la figura 3 se muestra un nodo, denominado ListaPrecios definido en base a las clases de la figura 2. La clase Productos es la clase a partir de la cual el nodo se convierte en observador del modelo conceptual, particularmente desde alguna de sus instancias y se denomina clase base.

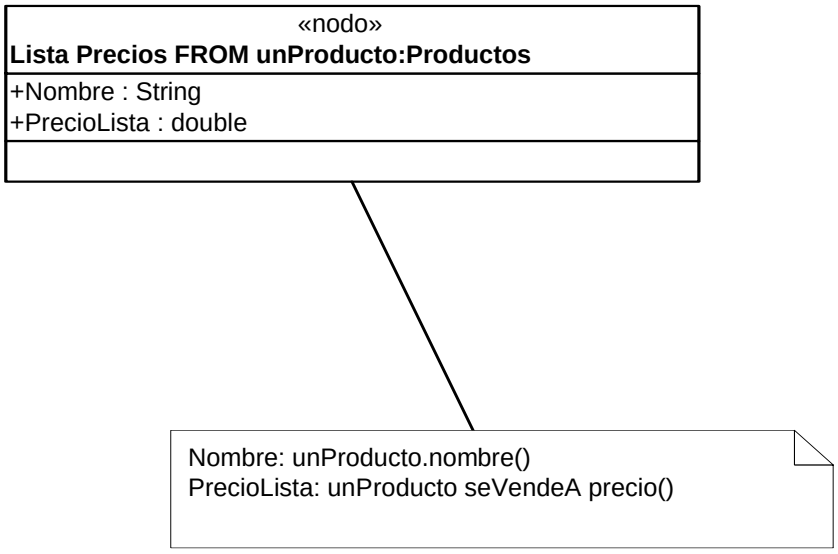


Figura 3: Clase navegacional.

Cada perfil de usuario que se define se modela con los elementos señalados (nodos, vínculos, anclas, etc.), conformando un esquema observador del modelo conceptual característico a dicho perfil. Aunque a la vez, diferentes perfiles pueden compartir (reusar) elementos con los esquemas de otros perfiles del modelo navegacional.

En la figura 4, se muestran algunas clases de un modelo de aplicación comercial y algunas pertenecientes a vistas de un par de perfiles de observadores diferentes, el de Clientes y el de gerencia comercial.

La clase observada por cada nodo está asociada con este por una flecha con línea segmentada. Miembros de otras clases son observados por los mismos nodos a través de las asociaciones conceptuales, como se verá más adelante.

Cada vista tiene nodos que son exclusivos, como N_Entrada para la vista clientes y EvaluacionPedidos, PoliticaPrecios y PedidosAProveedores de la vista Gerencia Comercial; así como hay nodos compartidos por diferentes vistas, como el nodo ListaPrecios.

El nodo N_Entrada es un observador del modelo conceptual desde la clase Clientes, a partir de la cual también puede "ver" miembros de Pedidos, CuentaCorrienteClientes, Productos, etc. según como estén definidos sus atributos y limitado a las asociaciones surgidas desde Clientes. El nodo ListaPrecios observa parte del modelo conceptual desde la clase Productos, la cual está asociada a Precios, lo que permite definir el atributo nodal PrecioLista como se mostró en la figura 3. También desde el nodo N_Entrada se puede navegar al nodo ListaPrecios a través del *link* de la figura y a otros posibles dentro de la Vista Clientes. Lo mismo ocurre con las otras vistas.

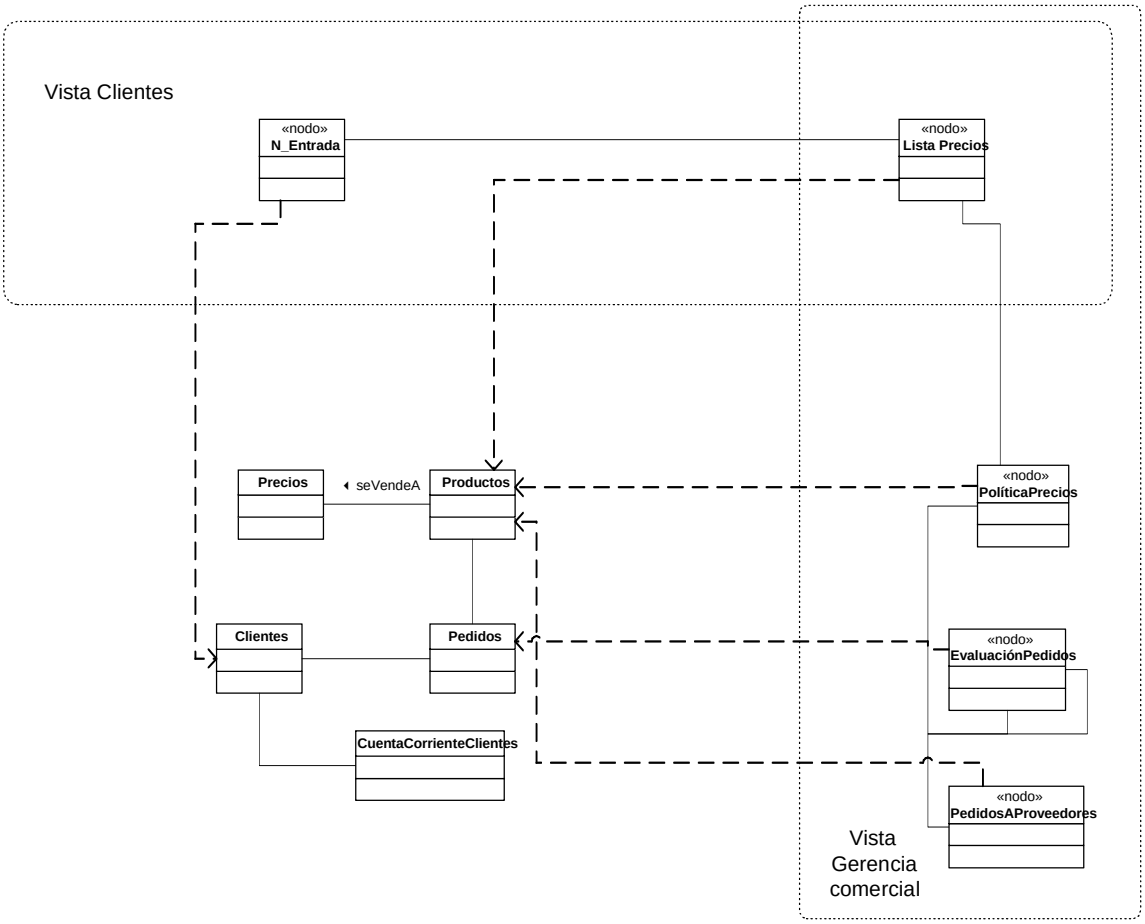


Figura 4: vistas navegacionales del modelo conceptual.

Las vistas representan el total de la observación posible que un tipo de observador puede realizar sobre el modelo conceptual desde el modelo navegacional. El diseño de la navegabilidad, que representa el recorrido de ítems de información dentro de cada vista, se realiza conformando contextos navegacionales, que son conjuntos de objetos navegacionales (tales como nodos, links u otros contextos navegacionales). conceptualmente relacionados. Como se verá en apartados posteriores, la relación directa entre el modelo conceptual y el navegacional, que es objeto del método que se presenta más adelante, está dada en las definiciones de los nodos. El resto de las construcciones navegacionales se basa en estos.

3 Elementos de representación del modelo navegacional

3.1 Representación de componentes de clases navegacionales

Los esquemas que conforman los modelos, conceptual por una parte y navegacional por otra, en cualquiera de sus vistas, están conformados por clases asociadas entre sí. Desde el punto de vista de diseño, se prescinde de la implementación de las clases del modelo conceptual, pero si se hace referencia a la implementación de las clases navegacionales en cuanto sus aspectos de observadores tanto de clases y asociaciones del modelo conceptual.

Los atributos nodales en OOHDM, se definen haciendo referencia a miembros de las clases conceptuales. Cada nodo se construye observando atributos y/o métodos seleccionados desde una clase que funciona como centro de observación, denominada clase base, desde donde puede observar otras clases, con la restricción de que estén asociadas a la clase base directamente o mediante otras clases. Un ejemplo de esto se encuentra en el nodo de la figura 3 respecto del esquema de la figura 2.

Partiendo de ese criterio, en [ROS 96], [SCH 98] se propone una definición de atributos nodales basada en lenguajes de consulta de objetos, particularmente haciendo uso del método propuesto en [KIM 90]. En la figura 5 se muestran las definiciones del nodo de la figura 3 usando este método.

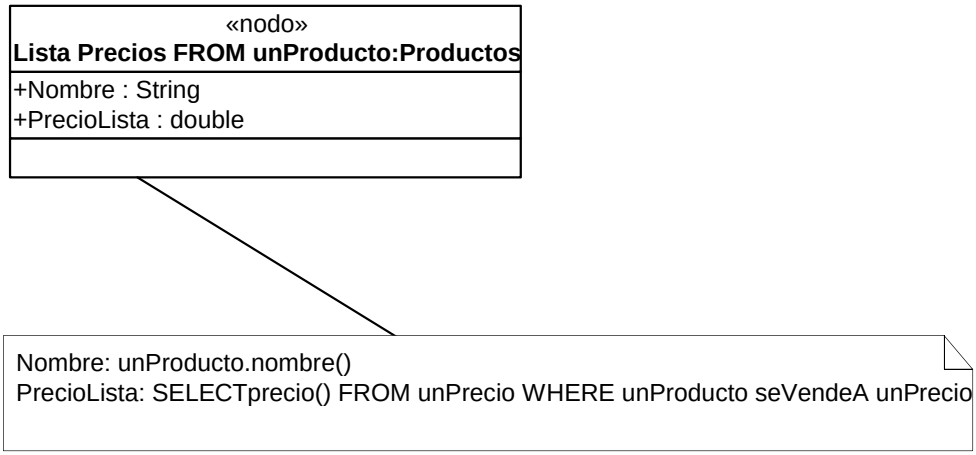
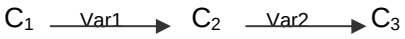


Figura 5: Clase navegacional. Con la definición original de atributos en OOHDM.

La definición propuesta en el trabajo actual es similar en cuanto a las definiciones de los atributos cuyos contenidos provienen de la clase base. Para los atributos cuya definición depende de clases asociadas, la observación se realiza mediante los miembros por los que una clase está asociada a otra. La propuesta tiene similitud con la de [KIF 92] conocida como “path expressions” para bases de datos orientadas a objetos, donde se efectúan consultas a atributos a través de un “database path” definido como una secuencia finita de objetos asociados, con la que se reemplaza el join relacional. Y mayor similitud con la de [PAL 95], donde hay una definición de “path expressions” para un esquema de clases representado a través de grafos, que consiste de:



Donde las flechas indican que clases están asociadas y la navegabilidad de la asociación, Var1 y Var2 indican la variable mediante la cual la clase de la izquierda puede navegar en la de la derecha.

Como se verá próximamente, en la definición nodal se utiliza el concepto de “path expressions” aunque ajustado a la definición de las asociaciones UML.

La definición de un nodo es de la manera que se muestra a continuación (en el apéndice A se detallan todas las definiciones del modelo):

Nodo N FROM obj:Clase de obj [INHERIT FROM: nodoPadre]

.

Ai: obj.mr

Aj: obj asociacion1 ms

Ai, Aj son atributos nodales

obj es una variable hace referencia a la instancia en particular que observa el nodo **N**.

mr es el miembro de **obj** que provee contenido al atributo **Ai**.

asociacion1 es la asociación mediante la cual **obj** está relacionada con el objeto del cual **ms** es miembro. El contenido del atributo **Aj** es provisto por **ms.**

Una forma más general de definición de atributos es:

Aj: obj asociacion1 asociacion2 ...asociacionN mt

Donde **asociacion1 asociacion2 ...asociacionN** es una cadena de asociaciones (y también un “path expression”).

Las definiciones de atributos están basadas en el esquema de clases, concebido desde el lenguaje UML y a diferencia de la propuesta de [KIF 92] se hace referencia explícita a las asociaciones.

Por otra parte las asociaciones tales como **asociacion1** están definidas como en UML con los siguientes atributos (entre otros):

Name: String
End1: Class ; isNavigable: boolean
End2: Class ; isNavigable: boolean

a los cuales se agrega (o sea no está contenido en la especificación UML) las referencias por las cuales un objeto puede navegar en otro:

Reference: <expresión>

Donde el resultado de <**expresión**> es la referencia a una instancia de la clase asociada navegable.

Este agregado se debe a que es la referencia quien provee entidad concreta a la asociación, es decir, a la variable que en definitiva la implementa. Como se verá más adelante, permite que la asociación sea sensible a cambios que la afecten desde el código.

Además, en la cadena de asociaciones **asociacion1 asociacion2 ...asociacionN**, debe cumplirse que End1 de la asociación iésima es igual a End2 de la asociación iésima-1, y la navegabilidad es de izquierda a derecha.

Para el ejemplo de la figura 5, la definición de atributos es la que sigue:

Nodo ListaPrecios FROM unProducto:Producto
.
Nombre: unProducto.nombre()
PrecioLista: unProducto seVendeA precio()

Donde la asociación seVendeA define como:

Name: "seVendeA"
End1: Producto ; isNavigable: false
End2: Precio ; isNavigable: true
Reference: unPrecio

Como se muestra en el comentario de la figura 2.

También los vínculos se representan de manera diferente a la de los trabajos originales de OOHDM:

Link: H

End1: Node1, Navigable = [True|False]

End2: Node2, Navigable = [True|False]

Label: lb

Ci:<..>

Cf:<..>

Donde se especifican los nodos relacionados por el vínculo, las navegaciones posibles y un rótulo para representarlo. **Ci** y **Cf** permiten establecer condiciones iniciales y finales de navegación (por ejemplo si un nodo está o no en condiciones de ser activado, o que ocurre en un nodo una vez abandonado).

3.2 Definición alternativa

El modelo de definiciones nodales anterior, se basa en los elementos de diseño del esquema de clases UML y tiene por objetivo mantener una correspondencia biunívoca entre ese esquema y la representación de las clases en algún lenguaje. Una posible implementación sintáctica de los atributos nodales, alternativa, más simple que la anterior, es evitando la referencia a las asociaciones conceptuales UML. De esta manera los “path expressions” se componen de las referencias que permiten la navegabilidad de un objeto a otro. Este modelo de definiciones alternativo, referido a la figura 5:

Nodo ListaPrecios FROM unProducto:Producto

.

Nombre: unProducto.nombre()

PrecioLista: unProducto.unPrecio.precio()

Donde unPrecio es la referencia de la clase Producto a la clase que contiene los precios del mismo. En términos generales, este tipo de definición es expresable en términos de la sintaxis de cualquier lenguaje orientado a objetos. Sin embargo, en el resto del trabajo se hará referencia al modelo anterior, para mantener la correspondencia entre código y lenguaje de diseño.

3.3 Objetos navegacionales primarios y secundarios

La sintaxis propuesta muestra que las definiciones nodales hacen referencia directa al modelo conceptual, por lo que podemos conocer la repercusión que cada cambio en este, producirá en el primero. Los vínculos, por el contrario, en su definición hacen referencia a nodos (orígen y destino de navegación), por lo que la repercusión de los cambios en el modelo conceptual no es directa, sino mediada por los nodos.

A los objetos mencionados podemos agregar contextos navegacionales y estructuras de acceso, definidos en base a los anteriores, y permiten abordar la representación de las distintas vistas y las posibles rutas de navegación como esquemas navegacionales. Un contexto navegacional es un conjunto de objetos navegacionales, relacionables bajo algún criterio, usualmente explorados en forma secuencial. Las estructuras de acceso indican contextos accesibles desde el nodo actual.

Por otra parte, anclas e índices, que son también elementos del modelo de navegación, si bien son especificados como los atributos, en general su definición depende de contextos navegacionales

Los nodos se definen en base a clases y asociaciones del modelo conceptual, por eso tienen una relación primaria o directa con el modelo conceptual. Los objetos navegacionales denominados contexto navegacional, estructura de acceso, vínculos y, en general, las anclas, tienen una relación secundaria, o indirecta con el modelo conceptual.

En el resto del presente trabajo se refiere a la relación entre los modelos conceptual y los objetos primarios del navegacional, particularmente a la adaptación de estos últimos cuando se producen cambios en los objetos del primero. Ciertamente, las consecuencias de los cambios, son sufridas también por los objetos que mantienen relación secundaria, razón por la cual debieran ser adaptados, pero en relación a objetos navegacionales primarios ya redefinidos. En las conclusiones, hay dedicadas algunas líneas respecto a esta segunda adaptación.

3.4 Miembros genéricos

Un lenguaje de modelado contiene la especificación de los elementos (metamodelo) que luego se usan para desarrollar un modelo. UML posee elementos pertenecientes al corazón de su paquete fundacional como Clase, Atributo, Operación y Asociación, con los que se modelan gran parte de las aplicaciones orientadas a objeto actuales. Parte de su paquete fundacional, define además, mecanismos de extensión para poder proveer a los elementos del modelo otros aportes semánticos, agregar nuevos elementos no existentes en

el paquete fundamental, pero necesarios a una aplicación en particular. Concretamente los mecanismos de extensión incluyen Restricciones, Estereotipos y Valores etiquetados [UML 03].

Un estereotipo permite subclasificar clases agregándoles riqueza semántica por sobre la que ya tienen los elementos base definidos en UML. Lo hace por el significado del estereotipo en sí, además de poder poseer restricciones y valores agregados. Incluso puede introducir una nueva representación gráfica. Un ejemplo de estereotipo está dado por la definición de nodo presentada en el capítulo anterior.

Las restricciones y los valores etiquetados pueden ser agregados a cualquier elemento del modelo, independientemente de que estuvieran estereotipados o no, permitiéndonos particularizar su semántica.

Una clase puede tener solo un estereotipo, sin embargo, como los estereotipos admiten el mecanismo de clasificación y generalización, ese estereotipo puede heredar las propiedades de otros. Esto permite darles significados especiales a ciertas clases y nodos de los modelos conceptual y navegacional respectivamente.

Las restricciones y los valores etiquetados por el contrario, pueden agregarse.

Entre las posibilidades que nos permiten los valores etiquetados, en cuanto a agregar algún significado a elementos del modelo, está la de darle a cierto miembros la pertenencia a algún conjunto conceptual, permitiendo tratarlos desde un punto de vista más abstracto.

Frecuentemente los miembros de una clase, además de estar contenidos conceptualmente por la clase, pueden ser agrupables bajo algún subconcepto, sin que esto signifique modificación alguna para la clase en cuestión. Por ejemplo, en la clase Cliente de la figura 6, los miembros : direccion(), cp(), direccionLaboral(), fax() pueden ser agrupados bajo el concepto “Ubicación del Cliente”; direccion(), cp() bajo el concepto “Domicilio”; nombre(), fechaNacimiento(), sexo() bajo el concepto “Datos personales”. También conviene incluir en cada miembro genérico aquellas variables privadas a las que hacen referencia estos métodos (-direccion, -cp, etc.).

Para realizar este agrupamiento se rotulan los miembros en cuestión bajo un mismo *TaggedValue* (valor etiquetado), consistente de un par con el que, como primer elemento se señala que ese *TaggedValue* es un agrupamiento y se lo hace bajo el identificador “Miembro genérico”. El segundo miembro del par indica a que miembro genérico (o a que género) pertenece el miembro concreto en cuestión, por lo que contiene identificadores como: “UbicacionCliente”, “Domicilio”, “DatosPersonales”.

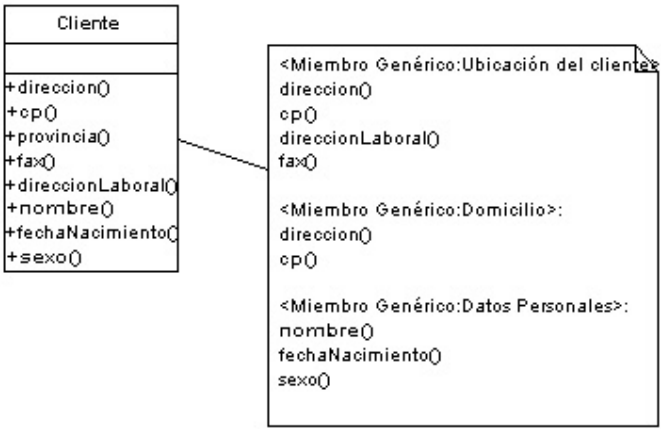


Figura 6: Clase con miembros rotulados.

Las definiciones de clases navegacionales haciendo uso de miembros genéricos tienen la forma siguiente:

Nodo N FROM obj:Clase de obj [INHERIT FROM: nodo padre]

.

Ai: obj.mr

[Am]. obj.<Miembro genérico: varmiembros>

Aj: obj asociacion1 ms

[An]. obj asociacion1 <Miembro genérico: varmiembros>

o más general:

Aj: obj asociacion1 asociacion2... asociacionN mt

[An]. obj asociacion1 asociacion2... asociacionM <Miembro genérico: varmiembros>

donde **[Am]**,**[An]**, son arreglos de atributos formados por los componentes del miembro genérico. Los nombres de los atributos son los nombres de los componentes, así como sus tipos.

En la figura 7 el Nodo N_Cliente es un nodo observador de la clase Cliente definida en la figura 6 basado en esta última definición.

En la figura 8 se muestra que agregar el miembro e-mail y eliminar Fax a la clase Cliente rotulados como miembros del género “Ubicación Cliente”, tiene por consecuencia la actualización automática del nodo N_Cliente, cosa que no ocurriría si se hace algún cambio al miembro nombre(), que está incorporado como miembro concreto, no perteneciente a ningún género, de la clase.

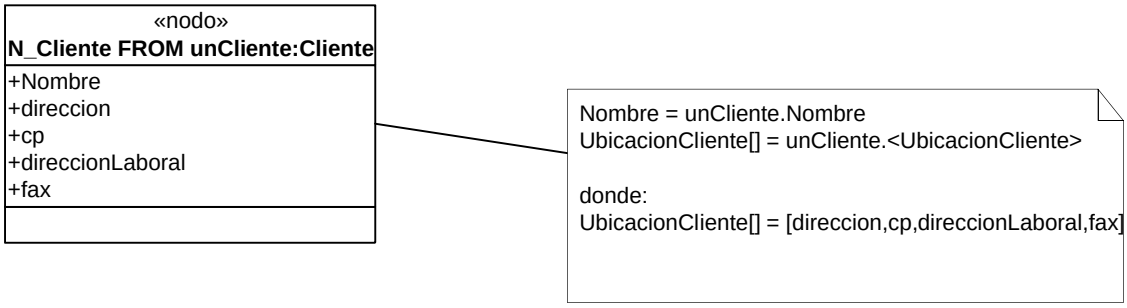


Figura 7: Nodo definido en base a una clase con miembros rotulados como miembros del género Ubicación Cliente.

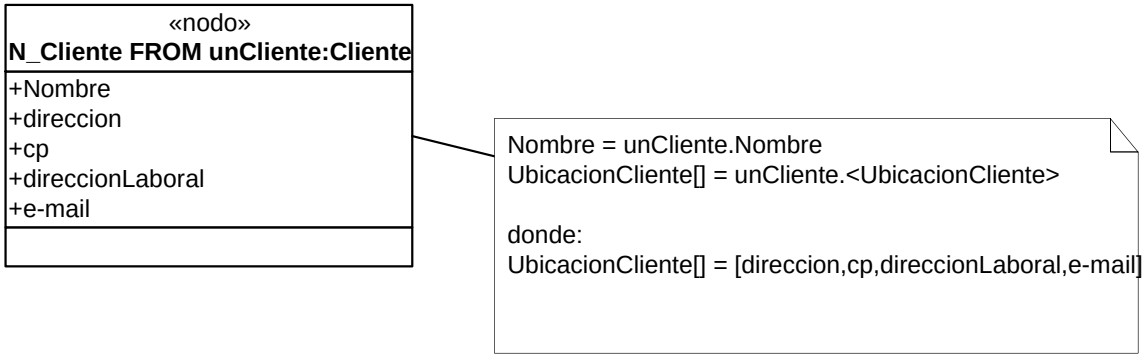


Figura 8: Nodo definido en base a una clase con miembros rotulados o genéricos, luego de un agregado y eliminación de miembros de la clase base.

Como se verá más adelante, el uso de miembros genéricos minimiza el impacto que algunas modificaciones en el modelo conceptual tienen sobre el modelo navegacional, particularmente agregados, eliminaciones y cambios de denominación de miembros.

4 Evolución de Software

4.1 Características del software actual

Una de las características sobresalientes del SW actual es la de estar sometido a una constante presión de cambio, originada en nuevos y a la vez más complejos requerimientos. La ingeniería de SW ha dado cuenta de esta situación con diferentes propuestas metodológicas y técnicas para llevar a cabo la evolución del SW, como respuesta a esa presión, que abarcan desde modelos de ciclo de vida de desarrollo, hasta herramientas de automatización de generación de SW.

Un aspecto central de la evolución del SW es la evolución de su diseño, el que puede tener los siguientes causales [TOK 99A]:

• Incremento de capacidades para soportar prestaciones nuevas o modificaciones a las ya existentes.

• Desarrollo de la reusabilidad de componentes para aplicarlos al SW actual o a otras aplicaciones.

• Reestructuración para soportar extensiones y mantenimiento.

y también reconoce razones humanas tales como:

• Experiencia adquirida o incorporada permite replantear aspectos del diseño.

• Nuevas perspectivas aportadas al proyecto.

• Experimentación en función del planteo de determinados objetivos.

La tecnología orientada a objetos concentra gran parte de los desarrollos actuales, debido, en parte, a su capacidad de respuesta a estos causales de cambio. Por una lado, esta tecnología permite abordar la complejidad mediante la abstracción, por otra herencia y polimorfismo potencian la reutilización de soluciones generadas con algún nivel de abstracción. Junto a estas, la separación de responsabilidades de un conjunto de parte que cooperan entre si, tiene un lugar destacado como lo demuestra el *pattern* modelo vista controlador originado en Smalltalk separando y desacoplando tareas de interfase y del modelo de la aplicación.

Más allá de interfases y aplicaciones, la separación de responsabilidades y el desacoplamiento entre clases, desde un punto de vista más general, es abordada por una

serie de *design patterns* así como su contraparte necesaria, el aspecto cooperativo entre las mismas. En este último aspecto, el pattern Observer aborda la problemática de un conjunto indeterminado de objetos observadores que requieren ser notificados y actualizados respecto de cambios de estado de otro objeto determinado. El pattern permite la evolución de los observadores, incluso el agregado o eliminación de algunos de ellos sin interferir en el objeto observado, lo que permite disminuir el impacto que cambios de una parte del software, tales como vistas u observadores pueden requerir sobre otra, tales como el modelo o elementos observados.

Como cualquier disciplina, el software, y particularmente su diseño, evoluciona a medida que se sistematizan aspectos del mismo, los *design patterns* son, además, en particular los desarrollados por [GAM 95] son una base sistemática de análisis de la evolución del SW, ya que representan soluciones a aspectos en los que el SW puede evolucionar, aunque por cierto, no exhaustivamente [STE A].

4.2 Reestructuración de software y operaciones de refactoring

Con el objetivo de automatizar la aplicación de patterns y otros cambios a aplicaciones OO, diversos autores [OPD 92] han aportado a la sistematización del proceso de introducción de los mismos a aplicaciones existentes, mediante operaciones de reestructuración conocidas como *operaciones de refactoring*.

Una *operación de refactoring* es una transformación de un programa que preserva su comportamiento y que normalmente reformula una parte del diseño de la aplicación [OPD 92]. La sistematización y eventualmete las automatizaciones que a partir de aquella se generen, permiten minimizar los errores propios de tareas manuales, minimizar tiempos y costos del cambio y mantener una adecuada configuración del software.

En la figura 9 se muestra, como ejemplo de aplicación de una operación de refactoring, los diseños previo y posterior a la aplicación de la operación substitute [TOK 99A], que cambia la dependencia respecto de una clase, hacia su clase base.

Hay operaciones de refactoring que requieren de llamadas a otras operaciones de refactoring. En este caso son operaciones compuestas. Por el contrario, si una operación no depende de otras operaciones, se denomina primitiva o simple.

La construcción del sistema de operaciones de refactoring a partir de un conjunto de operaciones simples y un conjunto de operaciones compuestas dependiente de las anteriores, y a la vez abierto a incorporaciones nuevas, es coincidente en diversos autores. Sin embargo, hay una gran dispersión en la definición de ambos conjuntos. Si bien el análisis comparativo escapa a los objetivos del presente trabajo, en el apéndice B se muestra un conjunto de operaciones de refactoring. En B.2.1, las operaciones primitivas de

Opdyke [OPD 92]. En B.2.2 las operaciones primitivas de Ó Cinneide [CIN 00], en base a las cuales construye operaciones compuestas que denomina *minitransformaciones* y *transformaciones* cuya aplicación constituyen la introducción de un design pattern. En B.2.3 se muestran las operaciones según Tokuda [TOK 99A], quien las clasifica como operaciones de transformación del esquema de clases, operaciones de introducción de patterns al programa, e introducción de clases y miembros a partir de la identificación de Hots Spots [TOK 99B].

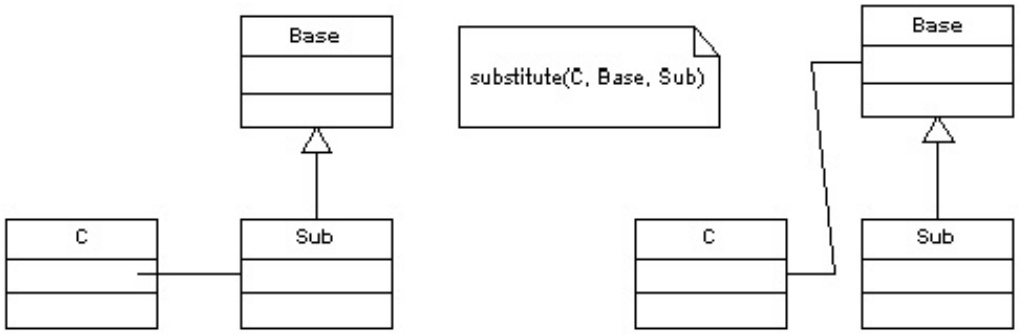


Figura 9: Aplicación de la operación de refactoring *substitute*.

Una visión extrema de la clasificación de operaciones refactoring permite descomponer las anteriores en operaciones atómicas, esto es reducidas a agregar o eliminar elementos tales como clases o miembros. Un ejemplo de esto es rename_variable (apéndice B.2), que puede ser descompuesto en remove_variable y add_variable. Algunos ejemplos de esta descomposición en términos de las operaciones add_[variable | method | class] y remove_[variable | method | class] se muestran en el apéndice B.2.4.

4.3 Efectos de la aplicación de operaciones de refactoring

Para refactoring, una modificación cualquiera en un programa sea por el método que fuera, una vez finalizada, debe garantizar que el resto no se vea afectado, esto quiere decir que la porción modificada debe ser “semánticamente equivalente” a la anterior. En términos del programa significa la *preservación del comportamiento* del mismo, una misma entrada produce la misma salida antes y después de la aplicación de la o las operaciones.

Complementariamente, entonces, las operaciones de refactoring, deben ser las encargadas de analizar si están dadas las condiciones para su ejecución, y en caso negativo abortar la misma. Esto requiere que, previamente se chequeen condiciones para la aplicación de cualquier modificación.

En las operaciones simples de creación o agregado de entidades (variables, métodos o clases), donde esas entidades carecen de referencias en el resto del programa, las condiciones a chequear se limitan a que la entidad agregada no entre en conflicto con otra existente en cuanto a su denominación y alcance. En el caso de eliminación de entidades, el concepto de preservación del comportamiento, requiere que la entidad eliminada no sea referenciada desde otra parte del programa, por lo que la ejecución de la operación requiere que no haya referencias a la variable previamente a la eliminación.

Generalizando lo anterior, la aplicación de cada operación lleva consigo un conjunto de potenciales violaciones a la integridad del programa, y por lo tanto, un conjunto de condiciones que deben chequearse previamente a su ejecución, llamadas condiciones iniciales.

Las operaciones además, generan valores, que en conjunto son las condiciones finales, que no tienen efecto directo sobre el programa, pero si son de utilidad para procesos posteriores en operaciones compuestas, donde la realización de una operación depende no solo de condiciones establecidas al inicio de una operación compuesta, sino de resultados intermedios producidos por operaciones componentes. En la figura 10 se muestra una operación compuesta de dos operaciones, se ve que si una condición inicial de una operación intermedia no se cumple, toda la operación vuelve a cero.

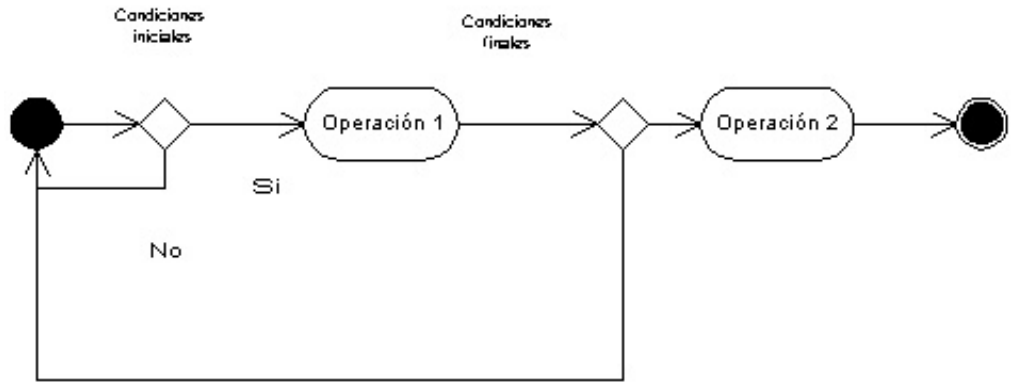


Figura 10: Ejecución de una operación compuesta

En el presente trabajo, se hace uso de la sistematización de modificaciones de programas orientados a objetos que proveen las operaciones de refactoring. Se distinguen las operaciones según su complejidad, sin predefinir un esquema acotado de niveles de la misma.

4.4 Las operaciones de refactoring en un marco orientado a objetos

Como se señaló en el capítulo anterior, el modelo navegacional observador del conceptual, está basado tanto en las clases y sus interfaces como en las asociaciones explícitamente definidas en el esquema de clases. A diferencia de esto, las operaciones de refactoring modifican las clases del modelo conceptual, e implícitamente las asociaciones entre ellas al crear, eliminar o cambiar las referencias. Sin embargo, no hacen referencia explícita a las asociaciones, y por lo tanto, estas no son modificadas. En realidad, las operaciones de refactoring, si bien permiten modificar estructuras complejas referidas en el esquema, no se refieren al esquema de clases sino al código fuente que lo implementa (hay algunas líneas de trabajo en refactoring actuales que se orientan a refactorizar desde grafos, ver apéndice B).

Por otra parte, las operaciones compuestas requieren del uso de una gran cantidad de variables, y del uso de parámetros para comunicar las operaciones componentes.

Además, fuera del alcance de las modificaciones realizadas por las operaciones de refactoring, también están las clases y relaciones navegacionales.

El modelo que se propone más adelante, incorpora operaciones para modificar las asociaciones conceptuales explícitamente y las clases navegacionales, en estrecho vínculo con las modificaciones realizadas mediante las operaciones de refactoring.

Es un modelo de refactoring orientado a objetos que aborda la complejidad generada por estas incorporaciones con encapsulamiento y con agregación. Cada operación de refactoring es ahora parte de un objeto específico cuyas responsabilidades son: llevar a cabo la modificación, chequear las condiciones iniciales para que se realice, y como se verá en el capítulo siguiente, actualizar las asociaciones del esquema de clases explícitamente y las clases navegacionales.

En el marco del modelo propuesto, el nombre de la operación de refactoring es ahora el nombre de la clase, que a su vez implementa métodos declarados en la clase abstracta Refactoring. Los parámetros de la operación original son ahora parámetros del constructor. La operación rehacer() es la encargada de ejecutar los cambios, llamando si es necesario a funciones de apoyo existente en esa clase o en otra relacionada. En la figura 11 se muestra la clase AddMethod según el modelo propuesto.

A partir de los parámetros obtenidos por la instancia de AddMethod, la operación rehacer agrega el método denominado *nombreMetodo* de tipo *tipo* a la clase *c,* cuya lista de parámetros, con sus respectivos tipos se obtiene de *lp*. El cuerpo del método, opcional,

agregado se obtiene de *metodo*. Si no hay conflicto con el nombre del método, los tipos tanto del método como de sus parámetros son correctos, la operación se realiza.

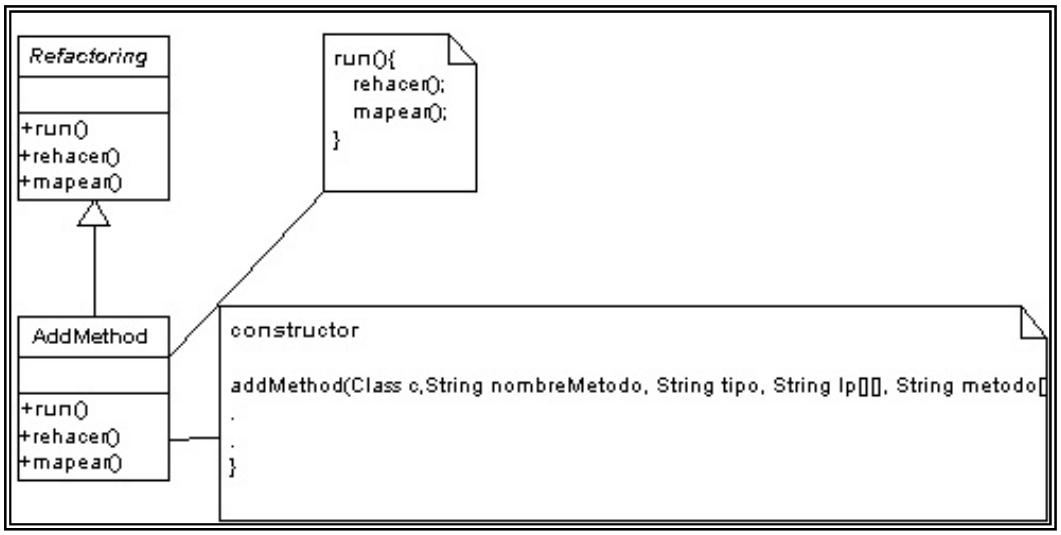


Figura 11: Clases refactoring

Las operaciones compuestas originan, en este nuevo modelo, objetos compuestos. Una operación rehacer(), de un objeto compuesto, llama a su vez las operaciones rehacer() de los componentes. El objeto compuesto chequea las condiciones de realización de la operación compuesta, y coexiste con los chequeos de cada clase componente. Si alguna condición no se cumple, se deshace toda la operación compuesta, volviendo el esquema al estado original tal como ocurre con el diagrama de la figura 10.

Un ejemplo de operación compuesta es la operación Substitute, cuyas consecuencias se muestran en la figura 9. Esta operación substituye la referencia existente en la clase C, respecto de la clase Sub por una referencia hacia la clase Base.

En términos de operaciones que componen Substitute hay varias alternativas de realización. Una de ellas es la eliminación de las referencias, mediante operaciones remove(), a la clase Sub, y su reemplazo por una nueva referencia, mediante operaciones add(), esta vez a la clase Base. Otra alternativa, que se desarrolla en la figura 12, es la de cambiar el tipo, en la declaración e instanciación, de la referencia a la clase Sub, por el tipo de la clase Base haciendo uso de *change_type.run(..).* Una vez realizado el cambio, funcionalmente la substitución está realizada, sin embargo en un esquema de clases de tipo UML, la operación es incompleta ya que no actualiza la caracterización de la asociación. Esto es, la asociación inicial tiene la forma:

Name: CConSub

End1: C ; isNavigable: false

End2: Sub ; isNavigable: true

Con el agregado del presente trabajo (tratado anteriormente):

Reference: x

Luego de la operación, la asociación debiera ser:

Name: CConBase

End1: C ; isNavigable: false

End2: Base ; isNavigable: true

Reference: x //La operación no cambió el nombre de la referencia, solo su tipo

Para realizar esa transformación es que se agregan a la operación Substitute.rehacer(), las 2 operaciones siguientes:

1)

removeAssociation(String nombreAsociación);

en el ejemplo en particular referencia = “CConSub”.

2)

addAssociation(String End1, boolean navegabilidad1, String End2, boolean navegabilidad2, Object referencia, String nombre);

en el ejemplo End1 = C.toString, navegabilidad1 = false, End2 = Base.toString, navegabilidad2 = true, referencia = x, nombre = CconBase)

Las clases de refactoring permiten modificar el esquema de clases conceptual estático (bajo la definición UML), aún con sus asociaciones.

En el apéndice A, se encuentran las especificaciones de otras clases de refactoring con sus respectivas operaciones.

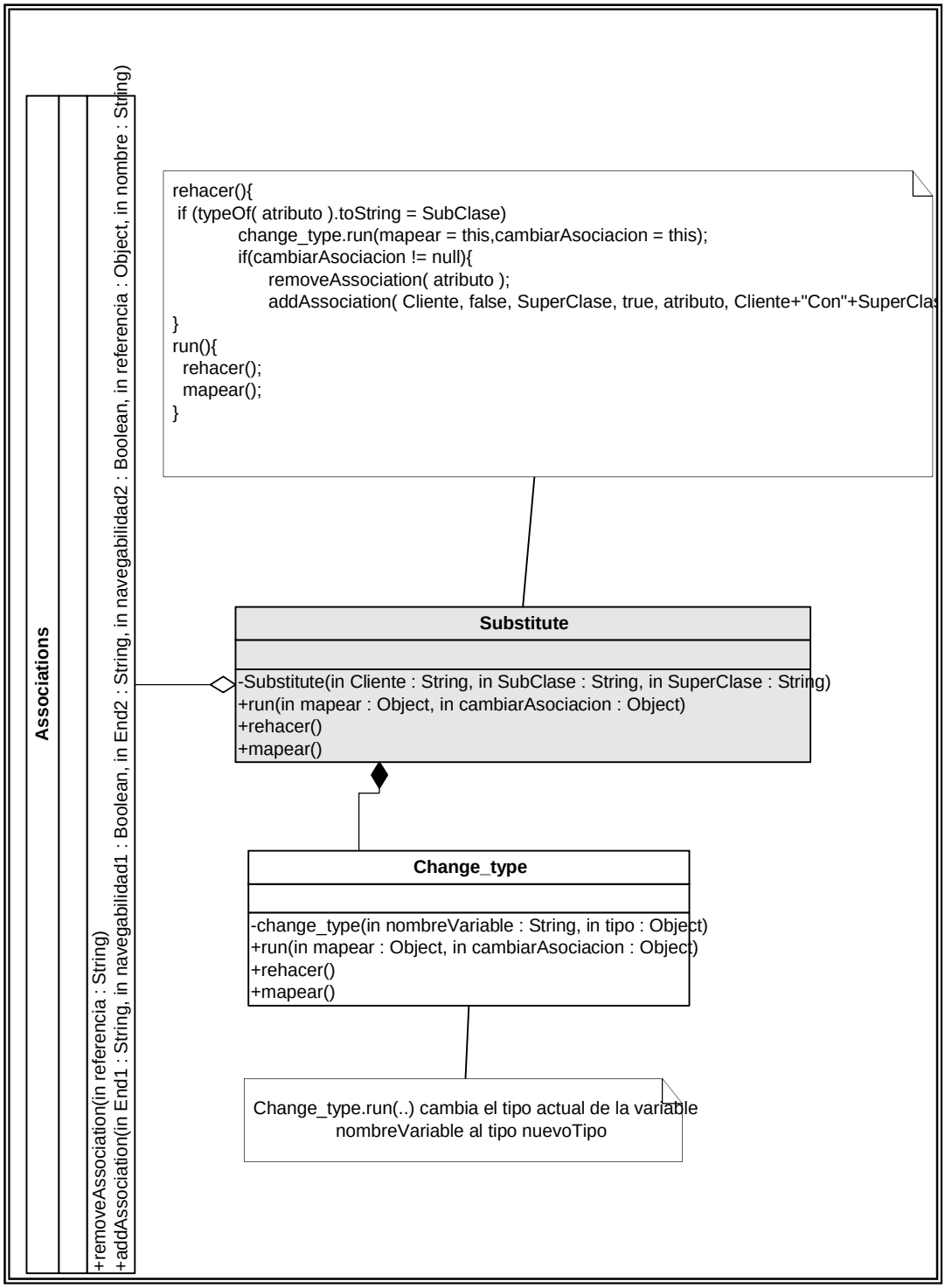


Figura 12: Clase Substitute y sus componentes.

5 Adaptación de software

5.1 Concepto de adaptación y sistemas adaptativos

Del diccionario de la Real Academia Española [RAE 01], Adaptar: 1. Acomodar, ajustar algo a otra cosa. 2.Hacer que un objeto o mecanismo desempeñe funciones distintas de aquellas para las que fue construido. 3. Modificar una obra científica, literaria, musical, etc., para que pueda difundirse entre público distinto de aquel al cual iba destinada o darle una forma diferente de la original. 4. Dicho de una persona: Acomodarse, avenirse a diversas circunstancias, condiciones, etc.5. *Biol.* Dicho de un ser vivo: Acomodarse a las condiciones de su entorno.

Además de la anterior, gran parte de las definiciones de la palabra *Adaptar* hace referencia a la relación de sistemas biológicos a su contexto, particularmente a su capacidad de modificarse para sobrevivir en el mismo.

La definición puede extenderse también a los sistemas artificiales, en cuanto a su capacidad acomodarse a su entorno y de desempeñar funciones nuevas. Entendiendo por entorno todo aquello que no es parte de lo que se considera “el sistema” en un momento dado y que de alguna manera interactúa con el.

También del diccionario de la Real Academia Española, Adaptable: Capaz de ser adaptado. Adaptación: Acción y efecto de adaptar o adaptarse.

Siguiendo la definición de Widrow [WID 85], se puede decir que un sistema es adaptable si es modificable respecto de las exigencias del entorno, de manera tal de preservar o mejorar sus capacidades originales.

A diferencia de la evolución, en la que se habla del cambio de un sistema en sí, la adaptación ejerce la evolución del sistema en respuesta a cambios en su contexto.

Un sistema y su contexto se relacionan entre sí por con un protocolo estímulo / respuesta determinado. En base a esta relación, la especificación de que es contexto y que es sistema es en algunos casos arbitraria. Si bien en algunos casos hay interfases claras que determinan ambos, también es posible en muchos casos, determinar arbitrariamente un conjunto de elementos de un sistema que cumplen una función determinada como (sub) sistema, en relación al resto cumpliendo el rol de contexto.

Aplicado al SW, el concepto de adaptación también admite la arbitrariedad de la relación contexto / sistema. Cambios en los requerimientos, modificaciones en los protocolos de comunicación con otros sistemas, cambios en el hardware o en sistemas operativos, etc., requieren de la adaptación del “sistema”, aunque cambios en partes de este necesariamente

conllevan la necesidad de adaptación de otras partes del mismo con las que mantiene un vínculo estímulo / respuesta, trasladando los límites de la relación contexto / sistema hacia adentro de lo originalmente definido como “sistema”.

Otro aspecto importante a considerar es quién “adapta” al sistema ante cambios en su contexto. En un sentido estricto, la adaptación se refiere a una capacidad propia del sistema de tener cierta lectura de los cambios en el entorno y de generar los cambios adecuados. Por ejemplo, muchos sistemas biológicos tienen la capacidad de autoadaptarse, y cuando un sistema artificial tiene esta propiedad, se lo denomina sistema adaptativo [WID 85]. Sin embargo, y en un sentido más amplio, la adaptación de un sistema puede provenir de una entidad externa a la relación contexto / sistema, un ejemplo de esto puede observarse en la adaptación de vistas materializadas que se muestra en párrafos siguientes.

En cuanto a su aplicación concreta, el concepto de adaptación en SW se ha aplicado a diferentes situaciones, algunas de las cuales se describen brevemente en los próximos párrafos. Distinguiéndose en las mismas:

- La adaptación en tiempo de diseño, que da respuestas permanentes a cambios permanentes en el entorno.
- La adaptación en tiempo de ejecución, que, por el contrario es el cambio de comportamiento como respuesta a estímulos desde las interfases.

5.2 Aplicaciones en software

En los párrafos siguientes se muestran algunos usos del concepto de adaptación, que dista de ser exhaustivo. Tiene por objetivo mostrar la variedad de uso del concepto de adaptación en software, aunque en realidad el uso del término es mucho más extendido ya que en general, cuando un diseño flexible, fácil de modificar, se considera adaptable. También ocurre con el concepto en tiempo de ejecución, donde en el paradigma de objetos la adaptabilidad está emparentada con el “dinamic binding”.

5.2.1 Personalización

Un tipo de adaptación es la *personalización* de interfases, estados y/o comportamiento, de acuerdo al perfil de usuario o a las acciones de este sobre el SW. La característica de este tipo de adaptación es la de modificar la percepción que los usuarios tienen acerca del SW, a través de un gran variedad de mecanismos, que van desde la restricción de acceso a tareas o información, encontrados en la mayoría de las aplicaciones de negocios, hasta los diseños para variar la topología, estructura y comportamientos de la navegación en aplicaciones hipermedia, como es el caso de algunas portales y aplicaciones

e-commerce [ROS 01][ROS 02]. Es un tipo de adaptación de SW de gran difusión en la actualidad.

Si bien en tiempo de diseño se da respuesta a la caracterización de los perfiles de usuario y se preveen conjuntos de acciones posibles de su parte, la adaptación se realiza en tiempo de ejecución, al concretarse el perfil interviniente y sus acciones a través de las interfases.

5.2.2 El pattern Context

Con el objetivo de manejar dinámicamente la evolución del SW orientado a objetos Seiter [SEI 96] propone la adaptación dinámica de una parte del SW respecto de otra haciendo uso de los patterns Traversal y Context [PAL 95]. El pattern Traversal permite recorrer una estructura de objetos sin conocimiento acerca de la misma (lo que lo hace más flexible que el pattern Iterator) a partir de un objeto determinado. El pattern Context define un tipo de relación en base a la cual el comportamiento de un objeto cambia de acuerdo a la presencia de diferentes objetos que forman su contexto. De esta manera, en tiempo de ejecución se pueden modificar las operaciones a realizar sobre un conjunto de objetos, desde otro conjunto de objetos de contexto. En la figura 13 se muestran las clases ContextM y ContextN, no pertenecientes a la jerarquía de la clase Base, pero que dinámicamente podrían alterar el comportamiento reemplazando las operaciones por omisión m() o n(), sobrepasando las limitaciones del pattern Stratergy generadas por la subclasificación. Sintéticamente podemos caracterizarlo como una adaptación dinámica, en tiempo de ejecución, de comportamiento, de un grupo de objetos en relación a otro grupo de objetos.

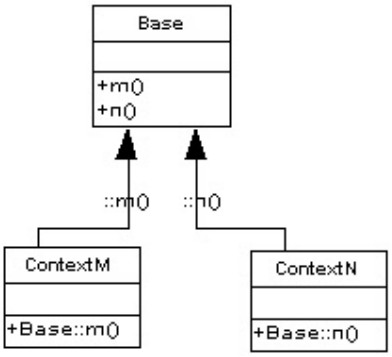


Figura 13: Adaptación de comportamiento con pattern Context

5.2.3 Adaptación desde información descriptiva

El método conocido como Adaptive Object Model (AOM) se caracteriza porque las entidades del dominio no son representadas por clases como ocurre típicamente en un diseño orientado a objetos, sino por descripciones (metadatos) que conforman un modelo de objetos dinámicamente modificable [YOD 02]. Estas descripciones están almacenadas usualmente en una bases de datos y al ser modificadas, el sistema refleja los cambios. De esta manera se evita modificar el código cuando cambian aspectos del dominio y además permite la existencia de interfases para que no programadores definan entidades nuevas. Su limitación principal está en que requiere que la arquitectura del sistema haya sido detalladamente ajustada.

El modelo está construido en base a patterns, como TypeObject que permite la creación de nuevas entidades dinámicamente, separando la entidad de su tipo. El comportamiento se define en los tipos mediante un pattern Strategy, con reglas que pueden ser evolucionadas dinámicamente.

5.2.4 Modificación de esquemas de vistas materializadas

La dinámica de modificaciones de esquema en bases de datos ha impulsado la necesidad de dar respuestas al mantenimiento de vistas materializadas. En muchos casos las fuentes de información son heterogéneas y distribuidas, y sus cambios requieren de la adecuación no solo del contenido [GUP 97], sino también de los esquemas de las vistas [XIN 99]

El framework EVE (Evolvable View Enviroment) [LEE 97] aborda el problema extendiendo las posibilidades de SQL como lenguaje de definición de vistas, para darle capacidad de adaptación a los cambios en los esquemas de diferentes fuentes de información, que son su contexto. Los atributos del esquema original son calificados como esenciales (inmodificables), intrascendentes (pueden ser modificados o eliminados) o reemplazables (pueden ser reemplazados por un equivalente, pero no eliminados).

La detección de los cambios en las fuentes de información se realiza desde un modelo descriptivo de fuentes de información (MIDS – Model for information source descriptions), que caracteriza las capacidades de las mismas, así como la relación entre ellas.

La adaptación se realiza en tiempo de diseño, es decir, las definiciones de las vistas se modifican al detectarse cambios en los modelos de fuentes de información, las que posteriormente que deben recompilarse. No hay intervención alguna de acciones en tiempo de ejecución.

5.3 Sistema y contexto

Los modelos de adaptación enunciados son solo una pequeña muestra de lo que, por un lado, se puede definir como contexto de un sistema de SW, y por el otro, de las soluciones que pueden darse como respuesta a cambios de ese contexto, dado el permanente cambio de los modelos de información y en algunos casos de los perfiles de usuario. Nuevos requerimientos amplían las responsabilidades de partes existentes e incorporan nuevas, lo que significa que el resto debe adecuarse a estos cambios.

La figura 14 muestra la relación genérica de un Sistema S y su Contexto C a través del conjunto de estímulos E y el conjunto de respuestas R. Si el contexto cambia de C a C', en general también cambia el conjunto de estímulos de E a E′ lo que conlleva la desadaptación del sistema S respecto de los nuevos estímulos, desconociendo algún subconjunto de E′ o generando respuestas erróneas ante los mismos.

Para mantener sus funcionalidades respecto del contexto, el sistema S tiene que migrar a S', generando una nueva relación contexto / sistema mediada por los estímulos E′ y las respuestas R′.

Como se señaló anteriormente, la migración del sistema S a S' puede realizarse desde el sistema mismo (sistema adaptativo) o desde una entidad externa a la relación contexto / sistema. Esta situación se representa en la figura 14 donde la entidad T genera S'. El origen de la adaptación (o migración de S a S') puede encontrarse en:

- Los nuevos estímulos del contexto, que es el modo de adaptación de los sistemas biológicos y en general de los sistemas de SW cuya adaptación se produce en tiempo de ejecución.
- Los modificadores del contexto. Es decir, el modificador del contexto es quien estimula la producción de cambios en el sistema mediante un modificador de sistema, ambos modificadores son, en un sentido abstracto, parte de la entidad externa T. A una modificación del contexto puede corresponder una modificación del sistema. Esta es la adaptación que se representa en la figura 14 mediante la flecha que une los bloques modificador de contexto y modificador de sistema. El modelo que se propone en el próximo capítulo es un método basado en este último esquema

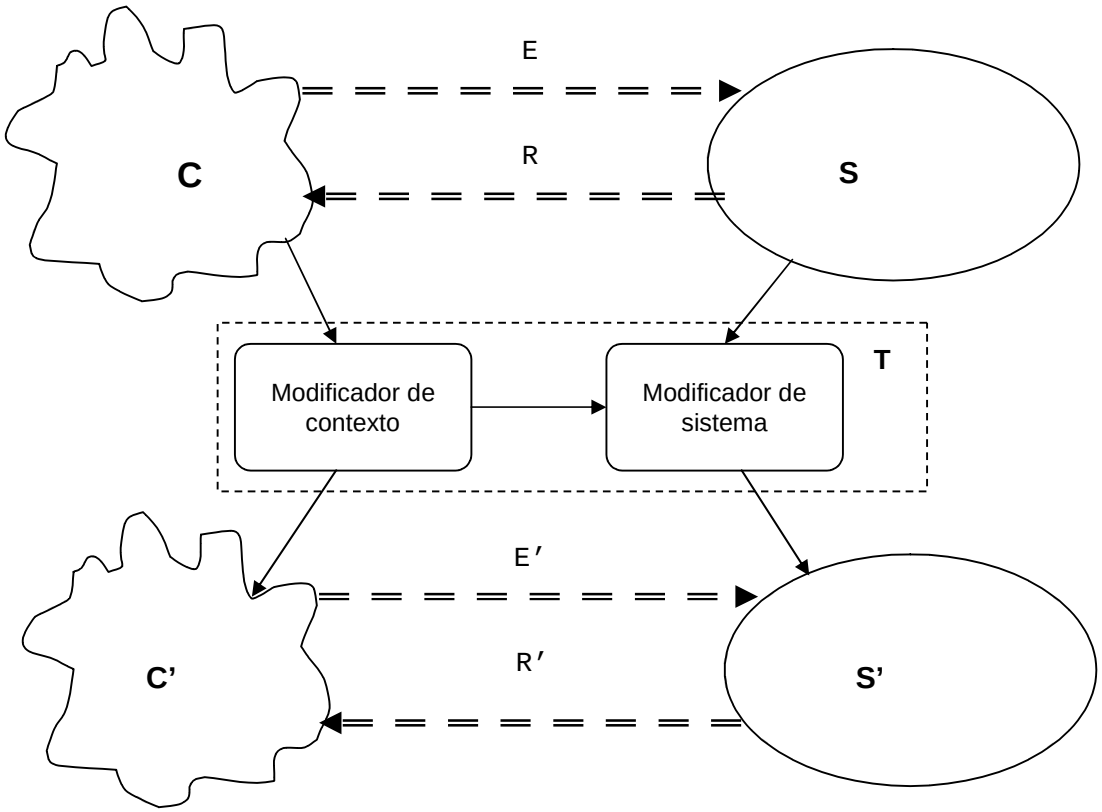


Figura 14: Esquema de adaptación. C y C’ Contexto inicial y final, S y S’ sistema inicial y final. : Contexto final. S: sistema inicial, S’. **E** y **R** estímulo y respuesta.

6 Modelo de adaptación

6.1 Impactos en objetos navegacionales

Como se vio anteriormente, las definiciones de objetos navegacionales referencian miembros de clases y asociaciones del modelo conceptual. Cuando estos miembros y asociaciones cambian, las definiciones de los objetos navegacionales quedan, en general, desactualizadas. Este fenómeno se denomina impacto de cambios del modelo conceptual en el modelo navegacional.

Los cambios están modelados, en la presente propuesta, con operaciones de refactoring de diferente nivel de complejidad como medio para proveer una base sistemática para su análisis. Así, el análisis de los impactos sobre el modelo navegacional se realiza a partir de estas operaciones.

En términos de operaciones atómicas de refactoring, aquella que agregue elementos al esquema conceptual de clases, no tiene impacto alguno en el modelo navegacional, ya que no altera las definiciones de sus componentes. Lo contrario ocurre con cualquier operación que implique el cambio de miembros, objetos o asociaciones a los que haga referencia alguna definición del modelo navegacional, por lo que toda operación que, expresada en términos atómicos contenga remove_[variable | method | class] o uninherit, potencialmente impacta en el modelo navegacional. En el apéndice B.2.4 se muestran algunas operaciones simples y sus correspondientes operaciones atómicas componentes.

La respuesta adaptativa sobre el modelo navegacional depende entonces de la caracterización del impacto según el tipo de modificación producida en el modelo conceptual.

El análisis de los posibles impactos que se realiza a continuación está basado en las modificaciones generadas a partir de operaciones atómicas.

6.1.1 Nodos

El encabezado de la definición de una clase de nodo es de la forma:

NODE nombre [FROM obj: Clase] INHERIT FROM: nodoPadre

Esta parte tiene como relación primaria con el esquema conceptual el objeto base observado por el nodo (obj) y su tipo (Clase).

- Las operaciones sobre clases tales como Rename_class(Clase,..), Remove_class(Clase) tienen como consecuencia la desactualización de la definición del nodo en general.
- Uninherit tiene como consecuencia que cada atributo o método heredado desaparece de la definición, por lo que las consecuencias deben ser analizadas desde cada uno de los atributos o métodos que lo componen.

La desactualización de la definición de un nodo requiere como acción su remoción del modelo navegacional, y de toda referencia al mismo desde otro nodo (como por ejemplo [HEREDA DE: ClaseDeNodo]), tanto vínculos, u otros objetos navegacionales secundarios. En el ejemplo de la clase navegacional de la figura 3 se puede suponer que la clase productos es renombrada a mercaderías, por lo que el tipo Productos ya no existe. Luego, el nodo ListaPrecios no puede instanciarse haciendo referencia al objeto de su clase.

Una excepción a esto es la operación Rename_class(..), desde la que se puede evitar la remoción del nodo, actualizando la referencia nodal contenida en la cláusula FROM o en INHERIT FROM.

El cuerpo de la definición de una clase de nodo es de la forma:

Ai: obj.mr
[Am]. obj.<Miembro generico: varmiembros>
Aj: obj asociacion1 ms
[An]. obj asociacion1 <Miembro generico: varmiembros>

o más general:
Aj: obj asociacion1 asociacion2 . asociacionN.mt
[An]: obj asociacion1 asociacion2 asociacionM.<Miembro generico: varmiembros>

- Asociacion1, Asociacion2,...son impactadas por cualquier cambio que afecte tanto las clases extremo como a las variables que concretan la asociación, tanto su existencia, su denominación, como su tipo.
- Cambios en mr, ms, mt, impactan por medio de cualquier operación que afecte sus tipos, denominaciones o parámetros.
- La operación uninherit, puede afectar a las de definiciones de atributos nodales desde un conjunto atributos y métodos de una clase conceptual, ya que de hecho puede significar la remoción simultánea de varios de estos.

Todos los impactos negativos a este nivel desactualizan la definición del atributo, no de la totalidad del nodo. La desactualización de la definición de un atributo nodal requiere su remoción de la definición del nodo. En el ya citado ejemplo de la figura 3, un cambio en la referencia de Productos a la clase Precios, hace que todos los atributos nodales que provienen de la clase precios queden desactualizados, como por ejemplo el atributo PrecioLista.

6.1.2 Vínculos (*links*)

El encabezado de un vínculo es de la forma:

Link: H
End1: Node1, Navigable = [True|False]
End2: Node2, Navigable = [True|False]
Label: lb
Ci:<..>
Cf:<..>

Está basado en objetos navegacionales, por lo que esta parte de la definición tiene una relación indirecta con el modelo conceptual. Esto significa que no es impactado directamente por la acción de operaciones de refactoring, sino por el efecto de estas sobre los nodos.

6.2 Impacto de operaciones compuestas

En general las operaciones elementales no se presentan en forma aislada, son parte de algún proceso de reestructuración. Dichos procesos se dan en el marco de la evolución de software y tienen objetivos como reusabilidad, mantenibilidad, etc. Suelen tener elementos comunes como, cambiar el uso de variables públicas por accesos indirectos, delegar cuando crecen las responsabilidades de ciertas clases, o cambiar relaciones de herencia a delegación cuando la estaticidad de la primera se convierte en un impedimento. Más aún, es en los procesos de reestructuración donde se aplican cambios más complejos como la introducción de design patterns.

Muchos de estos cambios pueden expresarse como una sola operación compuesta de operaciones elementales de refactoring, ejemplo de esto son la gran mayoría de las encontradas en el apéndice B. La composición permite ver los cambios estructurales como

operaciones de mayor nivel de abstracción, y analizar las repercusiones en el modelo navegacional desde ese nivel.

Como ejemplo se analiza el impacto de la creación de un acceso indirecto, mediante la operación compuesta Create_method_accesor y las posibles acciones a tomar desde el lugar de operaciones elementales y desde el lugar de la operación compuesta. Esta operación oculta el acceso a un atributo, modificando su visibilidad de pública a privada, y creando un método para accesarlo. Puede expresarse mediante las operaciones elementales (no necesariamente atómicas):

1. Change_scope_variable(var, privada)
2. Add_method(acceso_a_var())

Con la primera operación, cualquier definición de un elemento en un objeto navegacional que haga referencia a la variable *var* quedará desactualizado por lo que, para evitar inconsistencias, se debiera eliminar la definición del elemento que contenga referencias a la variable *var*. Luego, el agregado del acceso indirecto es intrascendente para cualquier nodo.

Sin embargo, el análisis desde la operación de mayor nivel (Create_method_accesor) permite observar que contiene una operación elemental que introduce un método que permite acceder a la variable y por lo tanto introducir la referencia al método acceso_a_var() en lugar del anterior acceso a la variable *var* evitando la eliminación de elementos de objetos navegacionales.

Concretamente, la operación compuesta contiene no solo una eliminación que puede impactar en objetos navegacionales, sino también su reemplazo. Lo que permite construir una estrategia de compensación a partir de las mismas operaciones componentes, que neutralice el impacto producido por las operaciones atómicas, o en general, de menor nivel.

6.3 Estrategias de adaptación

Los métodos que se presenta a continuación tiene por objetivo lograr que el comportamiento del modelo navegacional (o *sistema* en términos del esquema de adaptación de la figura 14) no se vea negativamente afectado por cambios en el modelo conceptual(o *contexto* en términos del esquema de adaptación de la figura 14). Contiene dos estrategias de adaptación diferentes, aunque complementarias, basadas en la relación entre ambos modelos mostrada en el capítulo 3, que consisten en:

1. Neutralizar el impacto de las modificaciones del modelo conceptual en el navegacional, a partir del uso de miembros genéricos en las definiciones navegacionales.

2. Generar las modificaciones en el modelo navegacional a partir de las operaciones de refactoring aplicadas al modelo conceptual.

6.3.1 Miembros genéricos

El primer método consiste en absorver cambios en la firma de las clases, generados por operaciones simples de refactoring, haciendo uso del agrupamiento de operaciones y/o variables públicas (miembros públicos) en miembros genéricos, definidos mediante los TaggedValues del modelo de extensión UML. Por un lado la eliminación y renombramiento de miembros públicos de clases del modelo conceptual, incorporados en miembros genéricos, no impacta en definiciones de clases navegacionales realizadas en base a miembros genéricos como se ve en el ejemplo del capítulo 3. Por otro lado, el agregado de miembros concretos a un miembro genérico que es parte de la definición de una clase navegacional, produce su incorporación automática a la clase navegacional, cuando está definida de la siguiente manera (ver capítulo 3):

Nodo N FROM obj:Clase de obj

.

.

[Am]: obj.<Miembro generico: varmiembros>

.

[An]: obj asociacion1 <Miembro generico: varmiembros>

como se puede observar, los cambios en los miembros contenidos en miembros genéricos no alteran de manera alguna la definición del modelo navegacional, basada en el miembro genérico. En cambio, si se modifica la clase navegacional en sí que está finalmente compuesta de miembros concretos, como se ve en el ejemplo representado las figuras 7 y 8.

Desde la óptica del esquema de adaptación presentado en la figura 14, el método de miembros genéricos prescinde de la entidad externa **T** para adaptarse a cambios en el contexto, es decir, se ajusta a la definición de sistema adaptativo, ya que el sistema contiene en sí mismo la capacidad de adaptación. Sin embargo, es un método limitado a agregados, eliminaciones y cambios de nombre de miembros de clases que pueden ser genéricamente agrupados.

Casos más complejos en la evolución de software, como delegación de responsabilidades, accesos abstractos, cambios en las asociaciones, etc. no pueden ser absorbidos por este método.

6.3.2 Redefinición de componentes

El segundo método responde a cambios más generales en el esquema de clases, realizados tanto con operaciones simples como compuestas de diferente grado.

Este método se basa en sensibilizar a los objetos navegacionales primarios a las operaciones que realizan los cambios en el modelo conceptual, y no a los cambios en si. Ante cada operación que realiza una modificación en el modelo conceptual, hay predefinida una operación que modifica el modelo navegacional para adaptarlo. Es decir que se mapean los cambios del modelo conceptual al navegacional.

En términos del modelo de refactorización basado en objetos presentado en el capítulo 4, la clase que encapsula la operación rehacer(), modificadora del modelo conceptual, también encapsula la operación mapear(), que hace lo propio con el modelo navegacional.

La operación mapear() recorre el conjunto de clases navegacionales en busca de referencias a los elementos que fueron modificados por rehacer() y realiza las modificaciones en las definiciones de los componentes navegacionales prefijadas en la operación.

Dado que las asociaciones, mediante las referencias que le dan entidad concreta, participan en igual nivel que clases y miembros en las definiciones de los atributos de nodos, se especifican también las asociaciones en el esquema de clases como se detalla en el capítulo 3.

En la figura 15 se muestra la clase abstracta Refactoring, raíz del conjunto de clases que encapsulan las operaciones de refactoring que permiten modificar las clases del modelo conceptual. Subclases de Refactoring son, por ejemplo, la clase substitute y la clase change_type de la figura 12. Las operaciones de la clase Association, visible desde Refactoring son llamadas cuando se requiere modificar asociaciones del esquema de clases. Las operaciones de la clase Maps son invocadas desde el método mapear, para efectuar los cambios en el modelo navegacional. Todas las clases de la figura 15 conforman la totalidad de la intervención externa representada por T en la figura 14 para la modificación de contexto (modelo conceptual) y sistema (modelo navegacional).

En la figura 16 se muestra un diagrama de secuencia con las clases que intervienen en la entidad externa T. Luego de que la operación rehacer(), propia de cada operación de refactoring, realice cambios en el código, operaciones de la clase Association completan los cambios del modelo conceptual actualizando, si corresponde, las asociaciones del esquema de clases. Posteriormente actúan las operaciones predefinidas de mapeo.

Las descripciones de las operaciones de Association se encuentran en el apéndice A.2 y las de la clase Maps en el apéndice A.3.

Las modificaciones pueden estar compuestas de otras operaciones. En estos casos se mapea desde la operación que contiene a las restantes. Como se ejemplifica en el apartado 6.2 (impacto de operaciones compuestas), una operación que es parte componente de otra no tiene “información” del cambio que se realiza en el modelo conceptual. En el ejemplo citado, la operación *change_scope_variable*, convierte en privada la variable *var*, lo que implica que cualquier miembro navegacional que se refiere a *var*, pierde visibilidad respecto de la misma, con lo que una operación de mapeo debiera eliminar el miembro navegacional, para evitar errores. Desde la óptica de la operación compuesta de mayor nivel, create_method_accesor, el mapeo debe cambiar las referencias desde la variable *var* al método *acceso_a_var*, adaptando las definiciones de esos miembros navegacionales.

El método *run* de la clase Refactoring es la única operación pública, y por lo tanto desde donde se invocan el resto de las operaciones. Los parámetros *mapear* y *cambiarAsociacion* son los medios por los cuales se determina la ejecución de las operaciones de mapeo para adaptar el modelo navegacional y cambio de asociaciones en el esquema UML respectivamente. En general, como se señaló anteriormente, esta es tarea de la operación de mayor nivel, por lo que en el resto de los casos el valor del parámetro será *null*. No hay imposición de limites al anidamiento de operaciones.

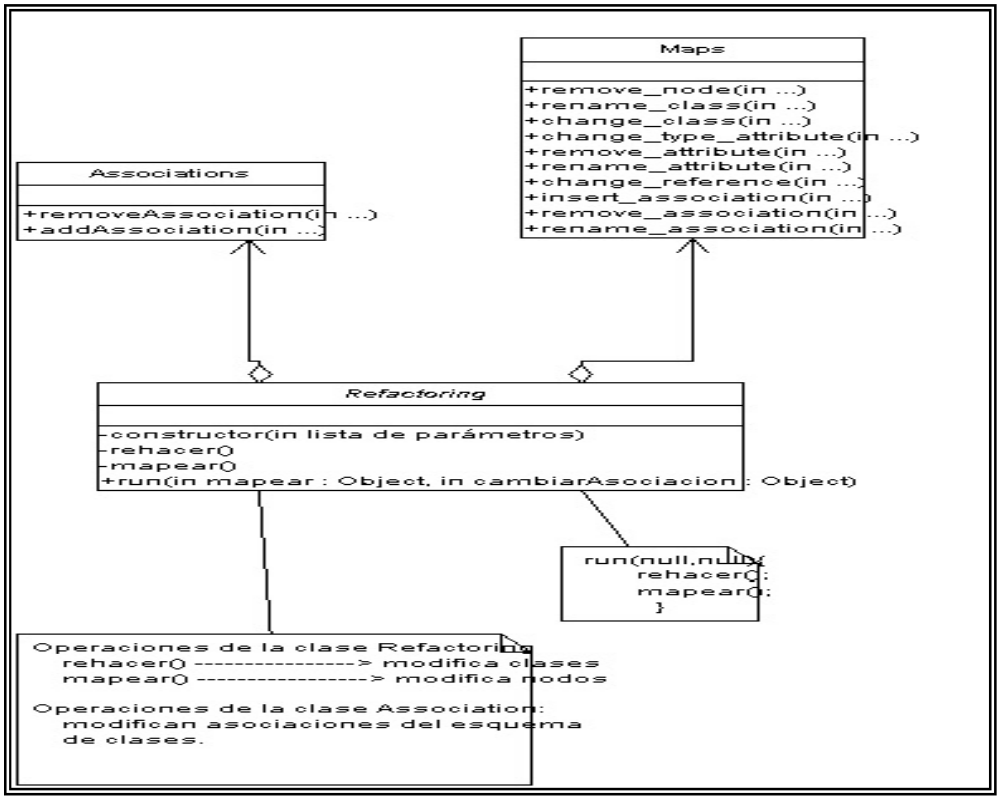


Figura 15: Clase abstracta Refactoring y operaciones de apoyo a mapeo y actualización de asociaciones.

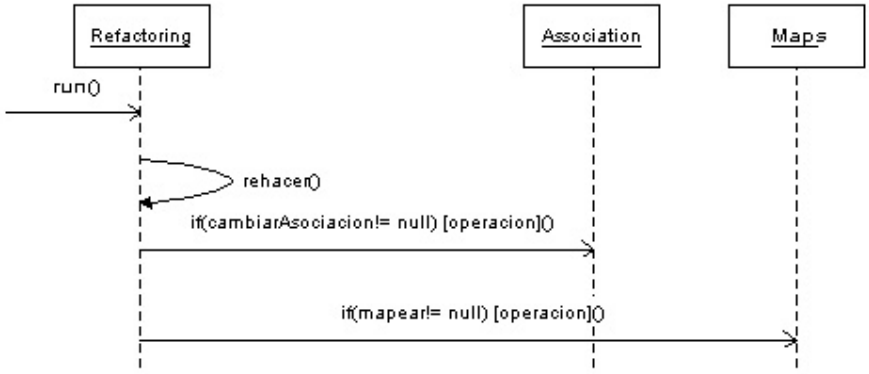


Figura 16: Diagrama de secuencias de las operaciones de modificación de los modelos conceptual y navegacional.

6.3.3 Operaciones simples

En este apartado se muestra el uso del método de redefinición de componentes navegacionales ante una operación de refactoring simple (no descompone en otras). Una vez terminadas, las operaciones (de refactoring y mapeo), el modelo navegacional mantiene sus propiedades de navegabilidad.

Un caso trivial de modificación del esquema de clases es renombrar un miembro de una clase. El miembro en cuestión, posiblemente es parte de la definición de una clase navegacional como en la figura 17, o más de una. En dicha figura, el método mX() es "observado" desde el nodo N (figura 17 A) . Una vez que la operación rehacer() de la clase Rename_method cambia la denominación del método a mY(), la operación mapear() de la misma busca en el modelo navegacional referencias al nombre, ahora desactualizado. En el ejemplo encuentra al atributo At de la clase C en el nodo N, el mapeo cambia entonces la referencia al método *mY*. La secuencia de estas operaciones se muestra en la figura 18, donde se omite la clase Associations ya que no se cambia ningún aspecto de la asociación entre las clases, si interviene la operación de mapeo change_reference.

Una posible implementación del mapeo de cambios basada en la operación de refactoring rename_method es la siguiente. Durante la instanciación de la clase de refactoring se introducen los parámetros que indican los elementos a modificar.

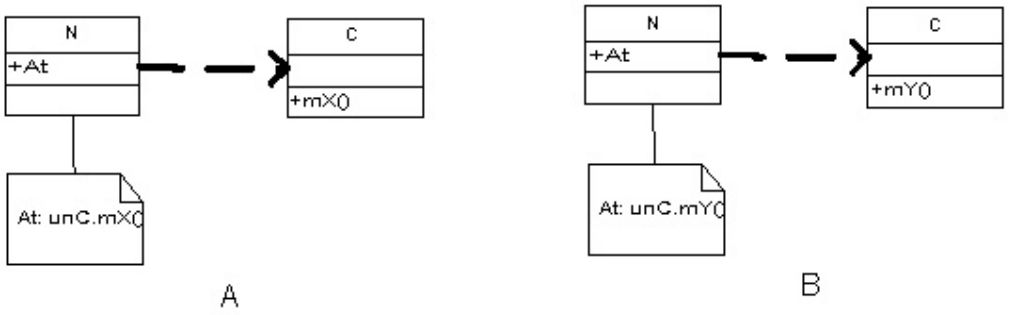


Figura 17: (A) Estado inicial de clase y nodo. (B) Efecto de mapeo tras renombrar un método.

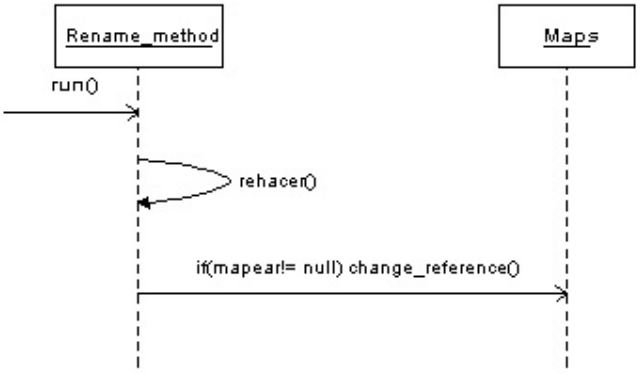


Figura 18: Diagrama de secuencias para rename_method.

Rename_method.Rename_method(C, mX, mY){ //Constructor

.

}

genéricamente la relación entre los cambios de ambos modelos se produce de la siguiente manera.

run(mapear = this, cambiarAsociacion = null){

rehacer(){

mX → mY;

if(mapear!=null)

mapear();

}

mapear(){

∀ Nodo:n; if mX ⊂ n ⇒ change_reference(mX , mY);

}

}

el valor de los parámetros de run(), indican que el objeto debe mapear hacia el modelo navegacional (mapear = this), pero no alterar las asociaciones (cambiarAsociacion = null). El resultado se muestra en la figura 17 B.

Hay operaciones de refactoring, en cuya clase no se define operación de mapeo, esto es debido a que los cambios que conlleva no pueden impactar en las definiciones de los objetos navegacionales. Esto ocurre, por ejemplo, en aquellos casos en los que la operación solo agrega elementos al esquema de clases, como add_[variable | method | class].

6.3.4 Operaciones compuestas

En términos generales, las operaciones con las que se modifica el modelo conceptual son compuestas. Difícilmente se encuentre en la práctica el uso de operaciones simples o atómicas en forma aislada. El uso de operaciones compuestas se traduce en un conjunto de modificaciones simultáneas en el modelo conceptual, producidas por sus componentes.

En cuanto a la relación con el modelo navegacional, un conjunto de modificaciones simultáneas en el modelo conceptual representa potencialmente un conjunto de operaciones de mapeo.

Un ejemplo significativo de esta situación es Create_method_accessor, cuyo impacto se analiza en el apartado 5.3. Los componentes de la clase de refactoring create_method_accessor se muestran en la figura 17. Si como condición inicial existe un acceso mediante una variable pública, la operación de mayor nivel delega en Change_scope_variable la tarea de ocultar su accesibilidad declarándola privada. Esta clase contiene además una operación de mapeo, que actúa precisamente cuando queda oculta la variable, ya que en cuanto a la observación desde el modelo navegacional es similar a su remoción, con lo que el mapeo debiera actuar eliminando toda definición en el modelo navegacional que contenga la variable, para evitar errores durante la navegación. Un ejemplo de esto se puede ver en la figura 20 A, donde la variable strNombre es pública, y parte de la definición, en el nodo N_cliente, del atributo Nombre: unCliente.strNombre. La aplicación de change_scope_variable la declara privada.

Change_scope_variable. Change_scope_variable (clase = “Cliente”, nombreVariable = “strNombre”, tipo = “private”){

.

.

}

run(mapear = this, cambiarAsociacion = null){

rehacer(){

typeOf(nombreVariable)= tipo;

if(mapear!=null)

mapear();

}

mapear(){

∀ Nodo:n;

∀ Atributo:a;if nombreVariable ⊂ a ∧ typeOf(nombreVariable) = “private” ⇒ remove_attribute(n,a);

}

}

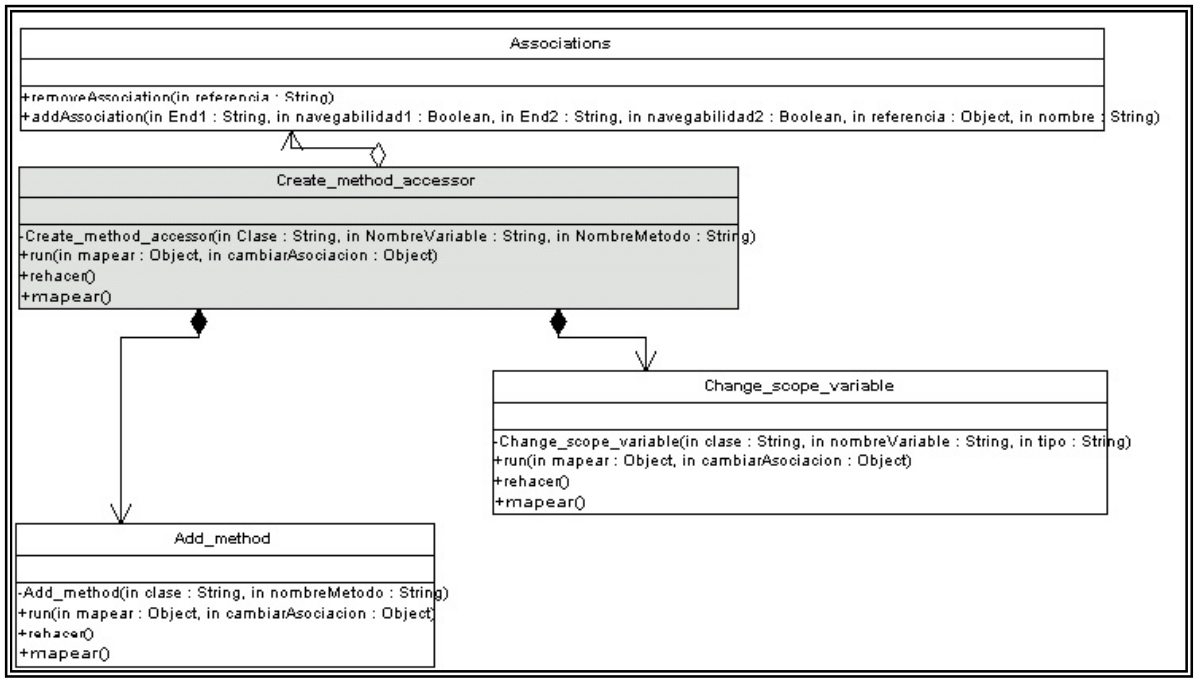


Figura 19: Composición de Create_method_accessor

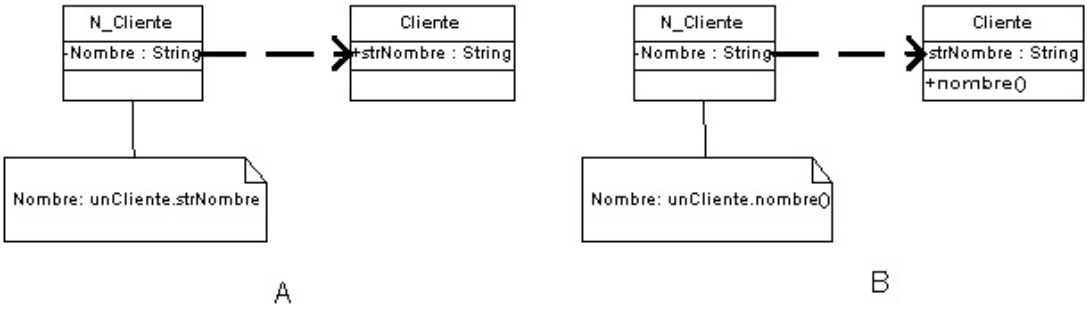


Figura 20: antes (A) y después (B) de la aplicación de Create_method_accessor y su respectivo mapeo.

Pero la semántica de la operación de mayor nivel (create_method_accessor) no es la de eliminar el acceso a strNombre, sino la de accesarla a través de una función. Desde su punto de vista no hay que eliminar el atributo nodal, sino por el contrario, mantenerlo cambiando su definición.

La operación de mayor nivel debe, entonces, evitar que se imponga la semántica de la de menor nivel, razón por la cual el parámetro mapear es igual a null. Es entonces create_method_accessor quien decide con exclusividad el mapeo.

En términos de implementación, toda operación mapear() (presente en aquellas clases de refactoring que impliquen algún posible impacto en el modelo navegacional) se ejecuta si se lo indica la operación de mayor nivel, mediante el parámetro no nulo de la operación run().

Entonces si, debe actuar la operación mapear() de Create_method_accessor:

La figura 21 muestra un diagrama de secuencias con las operaciones involucradas. Create_method_accessor crea el método por el cual accesar la variable mediante Add_method, luego cambia el alcance de la variable a accesar, hecho que impacta en el modelo navegacional, lo que obliga a cambiar las referencias al método de acceso.

mapear(){

∀ Nodo:n;

∀ Atributo:a;if nombreVariable ⊂ a ⇒ change_reference(n,a,nombre());

}

con lo que el modelo navegacional ha adaptado su definición, como se muestra en la figura 20 B.

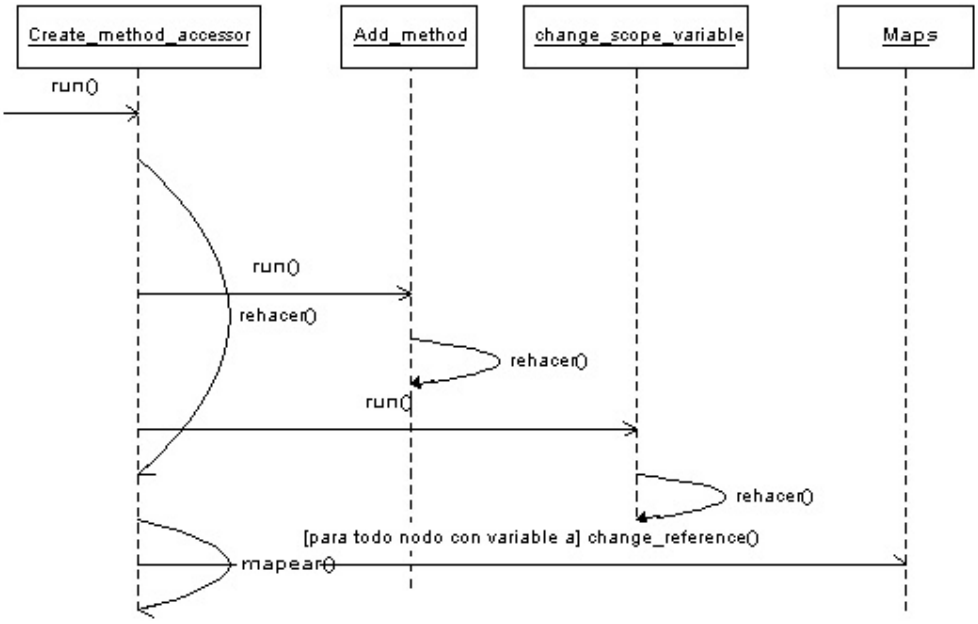


Figura 21: Secuencia de operaciones de creación de método de acceso y su correspondiente mapeo.

De esta manera puede observarse que la refactorización, tanto en el modelo conceptual como en el navegacional actúan bajo una estrategia de composición de operaciones, aunque la composición del mapeo no sigue directamente a la composición conceptual. La presencia del parámetro mapear permite esta independencia, y a la vez el manejo del mapeo por parte de las operaciones de mayor nivel.

6.3.5 Implementación alternativa

La realización de la transformación de Delegación haciendo uso del modelo de definición alternativo mostrado en el apartado 3.2 simplifica en parte las operaciones de las que se compone, al no necesitar crear la asociación *clienteConUbicacionCliente* en forma explícita en el esquema de clases. En cambio, esta se crea implícitamente con la variable unUbicacionCliente desde la operación de la clase de refactoring CreateExclusiveComponent. Sin embargo, en este caso, el esquema de clases queda desactualizado respecto del código.

7 Aplicación del modelo. Ejemplos de adaptación navegacional a la evolución de la aplicación.

Las operaciones están presentadas con una sintaxis similar a Java, aunque se introducen expresiones analíticas para evitar detalles de implementación que dificulten la lectura del sentido de la operación. Además en algunos casos se reemplazan variables con las que se implementan métodos con el contenido real de las mismas.

La especificación de las operaciones de este capítulo se encuentran en el apéndice A.

7.1 Delegación

7.1.1 Descripción

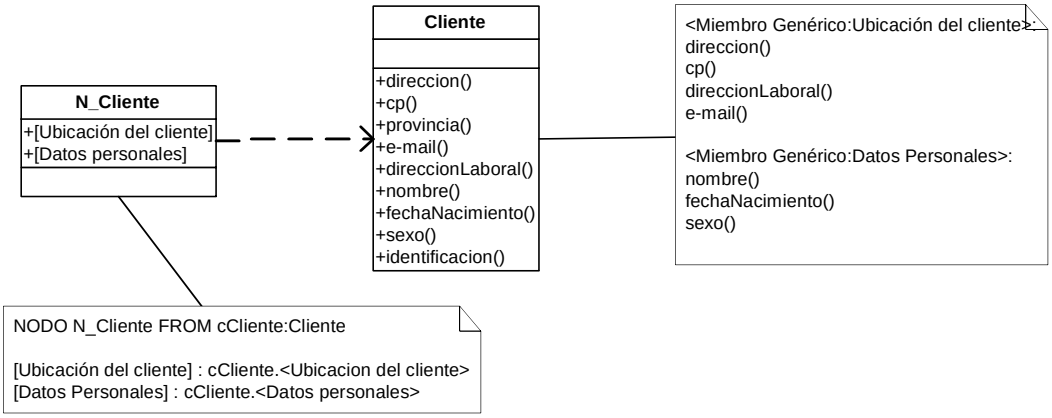
En el siguiente ejemplo se muestra un caso de Delegación en el modelo conceptual, con el efecto sobre el modelo navegacional y la aplicación del método. El punto de partida del ejemplo es la clase Cliente de la figura 20 A, donde además de los miembros se muestran las etiquetas que constituyen miembros genéricos. Como se señaló anteriormente, estas etiquetas (o tags) no afectan el funcionamiento de ningún modelo, sino solamente las definiciones de las clases navegacionales.

El nodo N_Cliente, está definido haciendo uso de las etiquetas de definición de miembros genéricos de la clase Cliente: <Ubicación del cliente> y <Datos personales>.

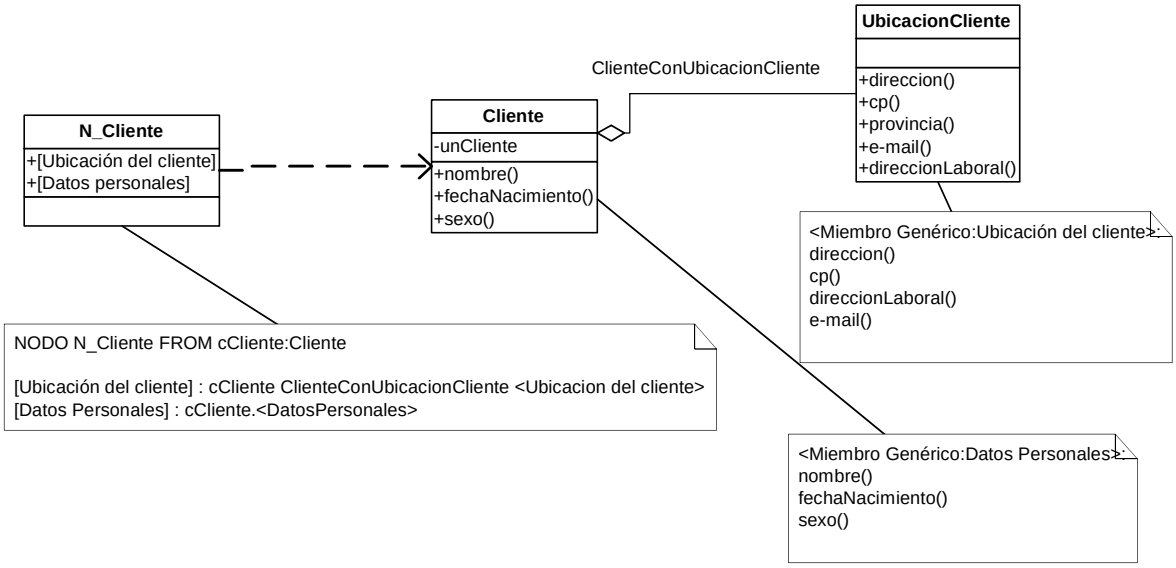
La refactorización Delegation, se aplica a la clase Cliente con el objetivo de delegar en una nueva clase, denominada UbicacionCliente parte de sus responsabilidades.

Delegation está compuesta de transformaciones de menor nivel, como se muestra en la figura 23. Algunas de estas transformaciones son, a la vez, también compuestas.

El constructor de Delegation, no mostrado en la figura, recibe como parámetros la clase delegante (Cliente), la clase delegada (UbicacionCliente) y el conjunto de miembros que serán transferidos de la primera a la segunda.



A



B

Figura 22: Esquemas conceptual y navegacional antes(A) y después(B) de la delegación.

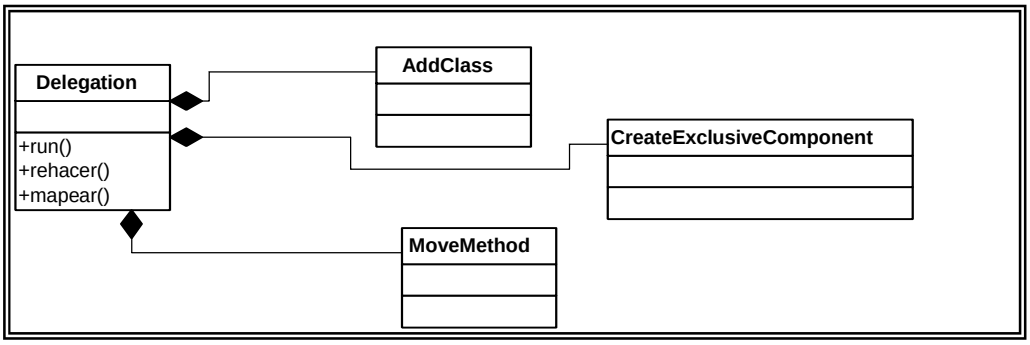


Figura 23: Composición de la clase Delegation. Por simplicidad no se muestra la asociación con la clase Association.

La aplicación de la operación rehacer() genera la siguiente sucesión de cambios en el esquema conceptual:

- Se crea una clase denominada “UbicacionCliente”. Tarea cuya responsabilidad es de la clase AddClass.
- Se agrega una variable en la clase delegante (Cliente) que referencia a la clase UbicacionCliente con el nombre: unUbicacionCliente, mediante CreateExclusiveComponent.
- La delegación de responsabilidades se concreta pasando los miembros preseleccionados de Cliente a UbicacionCliente, mediante MoveMember. Esta operación mueve tanto miembros concretos como genéricos establecidos en el parámetro ConjuntoDeMiembros. Como un mismo miembro concreto puede estar presente en dos miembros genéricos, la operación resuelve posibles duplicaciones.

También actualiza las asociaciones involucradas:

- Si bien la asociación existe por la creación de la variable unUbicacionCliente, no está especificada en el esquema de clases. Esta tarea es realizada por AddAsociation.

Finalmente deben mapearse los cambios al modelo navegacional:

- La operación mapear() es la encargada de actualizar las definiciones de las clases navegacionales. Para eso recorre todas las definiciones buscando los miembros eliminados de la clase Cliente para incorporarles el camino (path) desde esta a la clase UbicacionCliente. Las eliminaciones de la clase cliente producidas por el movimiento de métodos (MoveMethod), no producen efecto alguno en las definiciones del modelo navegacional, ya que como se vio en el capítulo anterior, se ejecuta solamente el mapeo de la operación de mayor nivel, es decir, Delegation.

La figura 22 B muestra la porción del esquema de clases modificado junto al nodo N_Cliente, redefinido después de la operación mapear() de la clase Delegation.

7.1.2 Implementación

En la figura 24 se muestra un diagrama de secuencia de las operaciones necesarias para implementar la delegación, actualizar el esquema de clases y luego mapear los cambios al modelo navegacional.

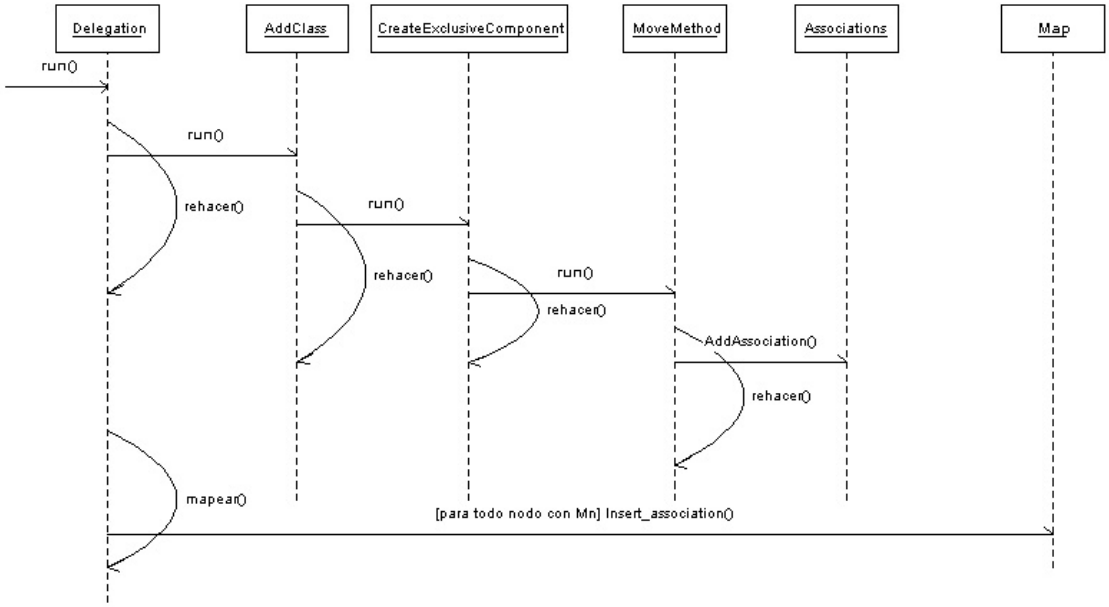


Figura 24: secuencia para delegación.

Se muestran algunos aspectos relevantes de la implementación de la refactorización Delegación aplicada a la clase Cliente (las especificaciones de las operaciones se encuentran en el apéndice A):

```
.
// Conjunto de miembros a transferir: lst = [<Miembro Genérico:Ubicación del Cliente>]

new Delegation("Cliente", lst , "UbicacionCliente"){   // constructor
.
ac = new AddClass("UbicacionCliente");
cec = new CreateExclusiveComponent("Cliente", "UbicacionCliente", "un" +
"UbicacionCliente",private);
mm = new MoveMember("Cliente", "UbicacionCliente", lst);
.
}

rehacer(){
        ac.run();
        cec.run();
        mm.run(null,this);
```

/* null bloquea el mapeo a clases navegacionales desde MoveMember,this habilita cambios en asociaciones */

new AddAssociation(End1("Cliente",navigable=False),
End2(classOf("UbicacionCliente"),navigable=True), reference = "un" +
"UbicacionCliente",label = string(End1) + "Con" +string(End2));
}
mapear(this){
∀ nodo:n, ∀ MiembroNavegacional:Mn, Mn ⊂ n
if ∃ MiembroConceptual:Mc, Mc.toString() ⊂ definicion(Mn) ∧ Mc ⊂ lst
Insert_association(n, Mn, string(End1) + "con" +string(End2), Mc)
}
}

La primera operación de rehacer, ac.run(), crea y agrega una clase de nombre "UbicacionCliente" al programa. La segunda, cec.run(), agrega a la clase Cliente una variable privada del tipo UbicacionCliente, denominada "unUbicacionCliente".

Luego, mm.run() mueve cada componente del conjunto lst, integrado por los miembros que pasan a la clase delegada. Cuando el componente es un miembro genérico, la operación se realiza para cada miembro concreto.

Si un miembro concreto es referente para una asociación con una tercera clase, MoveMember actualiza la asociación desde la nueva clase ("UbicacionCliente" en este ejemplo). Aunque no es una situación dada en el presente ejemplo.

Solo se muestran algunos aspectos de las operaciones, por simplicidad.

AddAsociation crea la asociación entre ambas clases. No aporta funcionalidad, pero si actualiza el esquema estático conceptual UML. Además, incorpora a la asociación la referencia que la concreta .

Finalmente, la operación mapear() recorre todos los nodos del conjunto de vistas navegacionales, y analiza para cada miembro navegacional, si es afectado por el movimiento de algún miembro referenciado, redefine el atributo, incorporando el camino desde la clase base del nodo hacia la nueva clase *host* del miembro, haciendo uso de la asociación.

7.1.3 Evolución de un esquema de clases y un nodo

En el ejemplo anterior se describió la aplicación de una refactorización compuesta. En el actual, el objetivo es mostrar una secuencia de transformaciones que realizan la

evolución del esquema de clases del modelo conceptual, y el proceso correspondiente de adaptación del modelo navegacional.

La evolución y la adaptación correspondiente se analizan desde diferentes puntos de partida (1.1, 1.2 y 1.3), donde el modelo navegacional, representado por un nodo, asume diferentes responsabilidades.

La numeración principal corresponde a la secuencia evolutiva. La secundaria, a las particularidades con que el esquema evoluciona según los diferentes puntos de partida.

7.2 Evolución por hitos

Los hitos de la evolución son los siguientes:

1. Esquema inicial

1.1. En la figura 25 se muestra un punto de partida. Una clase CD está asociada con la clase VideoClip. La creación de la instancia referenciada, no es en este caso, responsabilidad de la clase CD, sino de alguna clase no mostrada. El nodo N_CD tiene como clase base a CD y muestra video clips mediante la asociación de la clase base con la clase VideoClip.

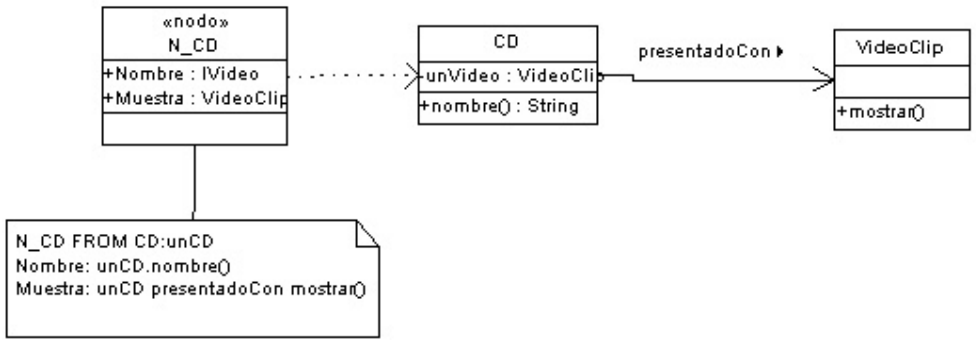


Figura 25: Esquemas navegacional y conceptual. CD no crea instancia de Video.

1.2. En la figura 26 se muestra un punto de partida levemente diferente, donde CD tiene responsabilidad de crear la instancia de VideoClip, aunque el nodo N_CD se mantiene independiente de dicha responsabilidad.

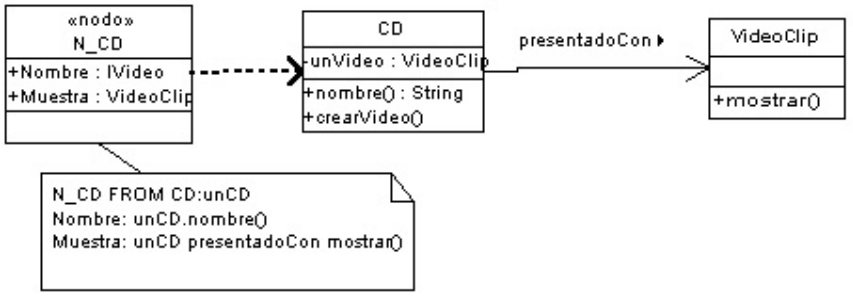


Figura 26: Esquemas navegacional y conceptual. CD crea instancia de Video. N_CD no "observa" la creación de la instancia.

1.3. En la figura 27, además de la responsabilidad de CD, el nodo N_CD hace uso de esa responsabilidad para crear la instancia (mediante un evento en la interfase o por efecto de la navegación previa).

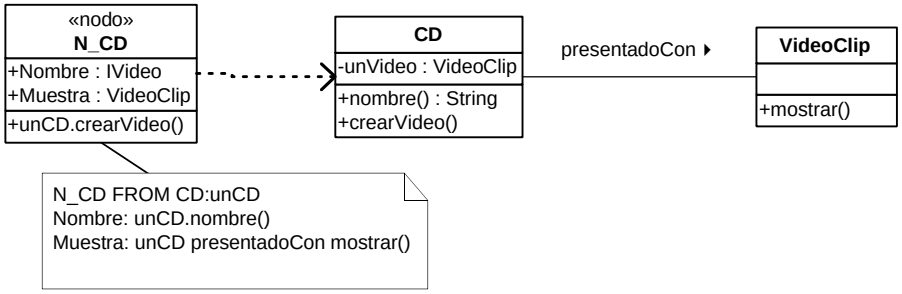


Figura 27: Esquemas navegacional y conceptual. CD crea instancia de Video. N_CD crea la instancia de Video.

2. Acceso por interfase

Mediante la refactorización AccesoPorInterfase, se crea la interfase IVideo con un conjunto prefijado de operaciones desde la clase VideoClip y se declara que esta última implementa la interfase.

La clase cliente de VideoClip, debe convertirse en cliente de IVideo, para lo que se crea una referencia (unIVideo). En este punto aparecen dos alternativas (al menos desde un punto de vista especulativo):

- La referencia a VideoClip (unVideoClip) continúa teniendo sentido, si solo parte de las operaciones llamadas desde CD son ahora operaciones de la nueva interfaz. En este caso, la clase CD estará asociada a ambas clases, mediante referencias distintas.
- Por el contrario, en general, el total de las operaciones que CD invocaba previamente de VideoClip, serán ahora operaciones de IVideo, por lo que al crear la referencia unIVideo, podremos eliminar unVideoClip. Esto significa que la asociación de CD con VideoClip es reemplazada por la de CD con IVideo.

En el ejemplo presente se sigue la segunda opción, aunque la implementación de la refactorización AccesoPorInterfase prevee ambas posibilidades.

2.1. Este apartado sigue el proceso desde 1.1.

El proceso que opera la secuencia de transformaciones es el siguiente:

- Se instancia la clase transformadora AccesoPorInterfase, que opera primero cambios en el modelo conceptual a través de sus dos componentes:
  - Abstraction: crea una interfase a partir de la clase VideoClip
  - AbstractAccess: cambia la referencia de la clase cliente, CD, con VideoClip hacia la interfase.
  - Se cambia la asociación, basándose en el cambio anterior, creando la nueva "CdconIVideo".
- AccesoPorInterfase, como refactorización de mayor nivel, mapea los cambios al modelo navegacional, revisando los nodos que en su definición contienen referencias a la asociación eliminada, cambiándola por la nueva.

La figura 28 muestra en el esquema el resultado de las transformaciones anteriores. En la figura 25 se muestran las operaciones con mayor detalle.

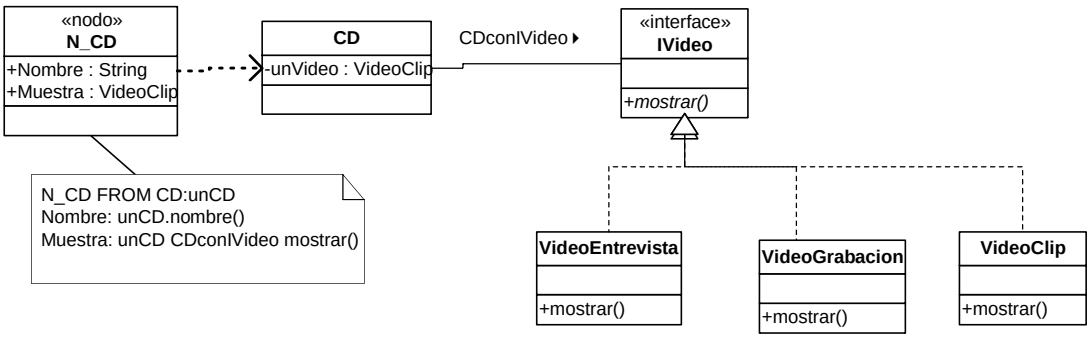


Figura 28: Acceso mediante interfase

- Se instancia la clase Acceso por interfase

new AccesoPorInterfase(“CD”, “VideoClip”, “IVideo”, conjuntoMetodos = [mostrar()]);

- se ejecuta la función run() que llama primero a la operación rehacer(), con las que modifica el esquema de clases (las asociaciones son cambiadas por AbstractAccess) y luego a mapear(), para adaptar los nodos observadores.

```
run(mapear = this,cambiarAsociaciones = null){
    rehacer(){
        /* Abstraction crea la interfase IVideo desde la clase
        VideoClip, copia la firma de los métodos del último parámetro
        en la interfase. No mapea ni cambia asociaciones */
        a = new Abstraction(“VideoClip”, “IVideo”, [mostrar()]);
        a.run(null,null);
        /* AbstractAccess crea la referencia y asociación con la
        interfase y elimina la anterior en caso de no ser necesaria. No
        mapea cambios por que AccesoPorInterfase se reserva esta
        tarea para sí. */
        aa = new AbstractAccess(“CD”, “VideoClip”, “IVideo”,
        [mostrar()]);
        aa.run(null, this);
        }

    mapear(){
        ∀ nodo:n; ∀ atributo:a, a ⊂ n
            if ∃ miembro:m, m ⊂ definicion(a) ∧ m ⊂
            conjuntoMetodos
                rename_association(n,a, refAbstracta =
                ”CdconIVideo”);
        }
}
```

Figura 29: Operaciones de Acceso por interfase

La figura 28 contiene además el agregado de dos clases que implementan la interfase IVideo, cuyas operaciones no se muestran, ya que no tienen efecto alguno en las definiciones de clases navegacionales.

2.2. Este apartado sigue el proceso desde 1.2.

El método crearVideo() no es observado desde el modelo navegacional, por lo tanto, este último no es afectado. Resultando el mapeo idéntico al de 2.1.

2.3. Este apartado sigue el proceso desde 1.3.

El acceso por interfase tiene efecto en la implementación del método crearVideo(), pero no en su interfaz, por lo que no interesa al modelo navegacional. Resultando el mapeo idéntico al de 2.1.

3. Delegación

Se delegan los miembros que tienen que ver con la instanciación de las clases Video. Para eso, la modificación Delegation crea la clase *creaVideo*, mueve el método crearVideo(), y las variables usadas por el, como *discriminador,* a la nueva clase. A partir de las clases delegantes y delegada, y de la variable con la que la primera hace referencia a la segunda, delegación crea la definición de la asociación que denomina CDconcreaVideo.

La operación MoveMember, componente de Delegation, a partir de la referencia en la clase delegada a con IVideo, crea la definición de la asociación creaVideoconIVideo.

Las dos definiciones de asociaciones, aunque desde componentes distintos son creadas con la operación addAsociacion.

No se detallan los componentes de la operación, dado que, es un caso similar al del ejemplo anterior.

3.1. Este apartado sigue el proceso desde 2.1.

Solo hay cambios en el esquema conceptual, ya que la creación del video no es observada desde el modelo navegacional.

3.2. Este apartado sigue el proceso desde 2.1.

Aunque los cambios repercuten en la clase base del nodo N_CD, la situación es similar al punto anterior, ya que el nodo no interviene en la creación.

3.3. Este apartado sigue el proceso desde 2.1.

La operación mapear() modifica la definición de la operación de creación de la instancia video por parte del nodo, de la manera siguiente:

```
public mapear(this){
    ∀ nodo:n
        ∀ MiembroNavegacional:Mn, Mn ⊂ n
            if ∃ MiembroConceptual:Mc, Mc.toString() ⊂
        definicion(Mn) ∧ Mc ⊂ lst
                Insert_association(n,Mn, string(End1) + "con"
            +string(End2),Mc)
    }
```

donde el miembro navegacional impactado es crearVideo() del nodo N_CD, y el miembro conceptual contenido en la definición es crearVideo() ahora de la clase CreaVideo.

El mapeador mueve la referencia desde unCD a unCreaVideo a través de la asociación CDconcreaVideo, con la operación de mapeo moverPorAsociacion.

En la figura 30 se muestra el resultado.

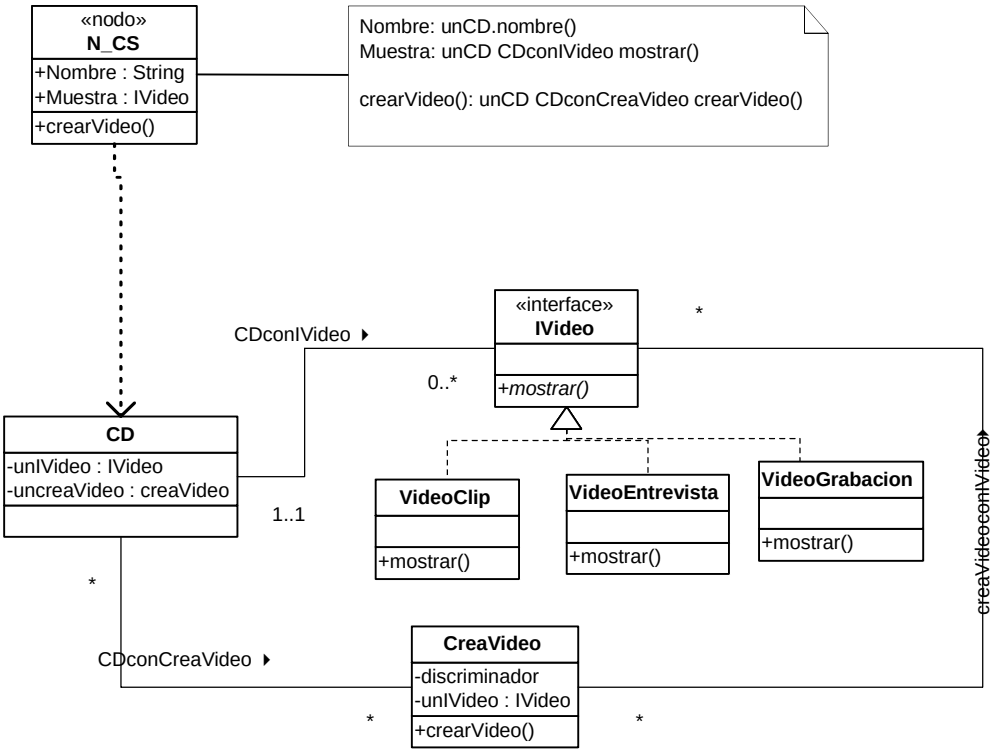


Figura 30: Esquema final de la evolución, donde el nodo tiene responsabilidad de crear instancias de Video.

4. Mapeo y patterns

Los design patterns de GoF aportan en general desacoplamiento entre clases que requieren de servicios y clases que los brindan. Como se señala en el apéndice B, la introducción automática de patterns es la meta más importante de varios autores de refactoring, dada su importancia cuando se pretende una buena evolución del software. Sin embargo, en los ejemplos presentados no se ha tratado la introducción de ninguno en particular. La razón es que en general, las estructuras con las que se introducen patterns no se incluyen en las definiciones nodales sino que son parte de la responsabilidad del modelo

conceptual, por lo que su introducción no impacta más allá que en la necesidad de producir cambios en referencias o la introducción de alguna asociación en la cadena de definición de un atributo nodal. Por ejemplo, la introducción de la interfase IVideo, en el punto 2 de la evolución, y el acceso abstracto a través de ella a las clases que la implementan, es similar desde un punto de vista estructural, a la introducción del pattern Strategy, y de igual manera que en el ejemplo de evolución, solo se requiere de la actualización de la asociación en el modelo conceptual.

La instanciación de las clases que implementan IVideo desde CreaVideo puede evolucionar hacia el pattern FactoryMethod, caso que se muestra en la figura 31.

Siguiendo la línea de evolución que comienza en los apartados 1.1 y 1.2 del ejemplo, donde el nodo no “observa” la creación de las instancias, por lo que la aplicación del pattern no tiene efectos en la definición nodal.

Siguiendo la línea de evolución que comienza en el apartado 1.3, el nodo tiene responsabilidad de instanciación de las clases que implementan IVideo, sin embargo, “no ve” la estructura mediante la cual lo lleva a cabo. Esto ocurre en el modelo conceptual, y la creación de la estructura comienza y termina en la refactorización conceptual, sin llegar al mapeo ya que la operación N_CS.crearVideo() permanece inalterada..

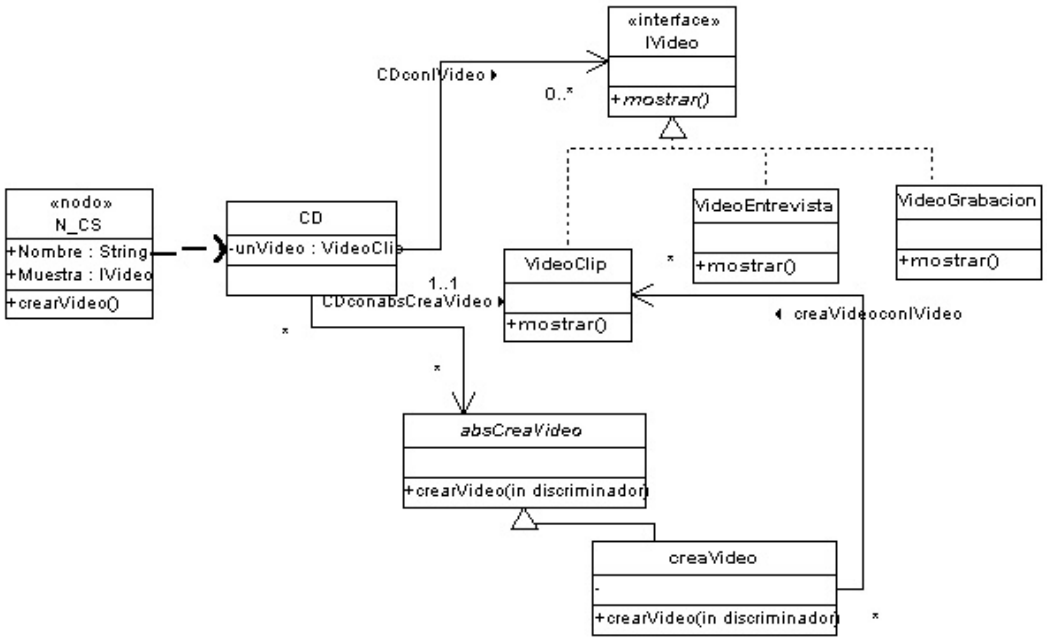


Figura 31: Aplicación de Factory Method

8 Conclusiones

8.1 Problemática y propuesta presentada

En la presente tesis se discutió, desde una perspectiva particular, la problemática de las aplicaciones web, como eje del creciente interés que despiertan estas en los campos industrial, comercial y científico, y entre los impulsores de desarrollos y búsquedas de soluciones a tradicionales y nuevas problemáticas del software. En este tipo de aplicaciones, confluyen algún paradigma tradicional, especialmente el orientado a objetos, con la navegación hipermedial. Pero además hay métodos de diseño de aplicaciones web que hacen eco de la idea de separar responsabilidades y establecer un esquema de colaboración al estilo del pattern Observer, entre los objetos que componen la lógica del dominio de la aplicación y objetos sobre los cuales se desarrolla la navegación. Como ejemplo de estos métodos se encuentra OOHDM, en referencia al cual se realizó la propuesta.

La metodología OOHDM contiene en su representación, dos modelos orientados a objeto, uno definido en base al otro. El primero, denominado modelo navegacional, representa el conjunto de objetos por los que es posible navegar observando al segundo, denominado modelo conceptual y que representa el modelo de dominio de la aplicación.

La existencia de ambos modelos, navegacional y conceptual, permite desde diferentes perspectivas navegacionales, reusar las clases del modelo conceptual, pero desde una perspectiva inversa, que es la óptica de la propuesta presentada, el hecho de que una parte del software se defina en base a otra genera por parte del primero una elevada dependencia respecto del último. Este fenómeno, en el contexto de la constante evolución del software establece la problemática de las desactualizaciones del modelo dependiente, en este caso, el modelo navegacional.

La propuesta, que se presentó con el objetivo minimizar el mantenimiento del conjunto de vistas que componen el modelo navegacional ante cambios en el modelo conceptual, se compone de dos métodos. Ambos se basan en el concepto OOHDM de que los objetos navegacionales en cualquier vista, están definidos sobre elementos del esquema de clases del modelo conceptual, o sea, en sus atributos y métodos visibles externamente, y hacen referencia a la firma de los mismos, y no a la definición de los métodos.

Los dos métodos de adaptación presentados pueden coexistir, aún teniendo características opuestas. Uno se basa en desarrollar definiciones genéricas de los miembros del modelo conceptual, lo que permite adaptar dinámicamente, en tiempo de ejecución los

atributos del modelo navegacional definidos sobre dichos miembros. El otro es de carácter estático, lo que se adapta es la definición de los miembros navegacionales y por lo tanto es una tarea que se realiza en tiempo de diseño.

En este segundo método, las modificaciones en el esquema de clases del modelo conceptual, sistematizadas mediante las operaciones de refactoring, son analizadas para determinar si el elemento modificado es parte de la definición de uno o más nodos (clases del modelo navegacional) pertenecientes a alguna vista, en cuyo caso, un mapeo mediado por condicionantes que tienen que ver con el contexto del elemento nodal, determina el cambio correspondiente a producir en la definición de cada nodo, que es entonces lo que se adapta a la nueva versión del esquema de clases. Por otra parte, este método de adaptación de las definiciones, no impone límites en cuanto a la complejidad de las transformaciones, siempre y cuando las mismas se expresen como operaciones de refactoring. Es incremental, ya que paralelamente al desarrollo de las operaciones de refactoring, se pueden definir las operaciones de mapeo correspondientes.

La adaptación, tiene por objetivo evitar la intervención del equipo de desarrollo en las clases que modelan distintas perspectivas navegacionales, cada vez que parte del modelo conceptual es blanco de modificaciones, produciendo un ahorro significativo en mantenimiento. Si requiere, en cambio, de la creación de una función de mapeo por cada operación de refactoring que plantee un posible impacto en el modelo navegacional.

Las operaciones de refactoring son además la línea divisoria a partir de la cual es factible, bajo el presente método, la automatización de los cambios en el modelo navegacional. Esto se debe a que las clases navegacionales no son una consecuencia de las conceptuales, sino un producto de análisis de requerimientos y diseño propia de ese nivel de abstracción, cuya implementación se refiere a las clases conceptuales.

Como se muestra en el apéndice B, el marco de transformaciones de software orientado a objetos es mucho más amplio que el descripto por refactoring, ya que otras transformaciones incluyen la ampliación de las características del software. Pero dado que el modelo navegacional tiene una ingeniería de requerimientos propia, las ampliaciones al modelo conceptual no son trasladables hacia aquel en forma automática, desde una perspectiva de diseño. Sin embargo, como se muestra en el ejemplo de evolución, si son agregados válidos las ampliaciones que provienen de subtipos, cuando se generaliza alguna clase existente observada desde el modelo navegacional.

En particular, el modelo navegacional, en OOHDM, se construye además con entidades tales como vínculos, contextos navegacionales, etc., que fueron definidas como secundarias a lo largo del trabajo presentado, dado que se definen en base a las clases navegacionales primarias, para quienes está dirigido el método presentado. Sin embargo, precisamente esa definición, de carácter formal, las hace pasibles de la aplicación en

segunda instancia del método de adaptación propuesto, es decir, de la misma forma en que se construyen funciones de mapeo desde en modelo conceptual a las clases navegacionales primarias, habrá que definir funciones de mapeo desde estas hacia las clases navegacionales secundarias.

Los elementos que forman parte de las capas superiores de OOHDM, tales como ADV (Abstract Data Views) e interfases, no fueron parte de la estrategia de adaptación presentada. Dada sus particularidades, fundamentalmente por representar un paradigma diferente, requiere posiblemente, del planteo de otros métodos de adaptación. En pos de lograr automatizar el mantenimiento de todas las capas componentes del modelo OOHDM, su abordaje es un tema pendiente.

8.2 Diseño y código

A lo largo del trabajo se ha dado importancia a la implementación del método de redefinición de componentes tanto en el código como en el lenguaje de diseño, procurando mantener la relación entre estos luego de las modificaciones. Para eso se realizó una pequeña modificación en la definición de asociaciones respecto de la propuesta en UML. La relación entre diseño y código es una carencia en los trabajos de refactoring orientados al código (no en aquellos que parten de grafos, como se analiza en el apéndice B), donde si se parte de un determinado esquema de clases, este es incoherente con el código luego de la ejecución de las operaciones, de manera tal que la “automatización” de cambio de código lograda mediante las operaciones de refactoring debe complementarse con el uso de la corrección del diseño por otros medios (manual, herramientas de ingeniería inversa, etc.). Esta situación es salvable con la propuesta de actualización de asociaciones presentada en este trabajo.

8.3 Hacia la generalización

Haciendo abstracción de las definiciones de los objetos navegacionales, la problemática de una parte del software que cambia, y otra, que al estar relacionada con esta queda desactualizada es sumamente amplia. El *pattern observer* por ejemplo, puede implementarse con múltiples observadores sobre un observado. Cuando se modifica este último, quedan desactualizados los observadores. Planteado el problema con este grado de generalidad, se requiere de un abordaje desde una óptica más abstracta, que permita generar respuestas con independencia de particularidades tales como definiciones nodales.

Esto presenta otra óptica a las operaciones de refactoring, las que requieren del cumplimiento de condiciones iniciales estrictas que en muchos casos tienen que ver con referencias en otras partes del software a algo que se pretende cambiar. Bajo el concepto de adaptación, es posible minimizar las condiciones iniciales, trasladando parte de la responsabilidad a la adaptación de la definición de partes dependientes. Por ejemplo, renombrar una variable solo se puede realizar bajo la condición de que no haya referencias a la misma desde otras partes del programa, la adaptación del código puede ser un camino para minimizar las restricciones que imponen este tipo de condiciones modificando las referencias.

En síntesis, se ha desarrollado un método para adaptar clases observadoras que funcionan como nodos en la navegación hipermedial, pero la potencialidad del mismo puede ser estudiada desde el ámbito más general de la evolución de programas orientados a objeto. Este es el desafío más importante desde el punto de vista del desarrollo ulterior del método.

9 Referencias

[BAR 00] Luciano Baresi, Franca Garzotto, and Paolo Paolini From Web Sites to Web Applications:New Issues for Conceptual Modeling. ER Workshops 2000: 89-100 http://www.elet.polimi.it/upload/baresi/pub/papers/WWWCM.pdf

[CER 00] Stefano Ceri, Piero Fraternali, Aldo Bongio. Web Modeling Language (WebML): a modeling language for designing Web sites. WWW9 / Computer Networks 33(1-6): 137-157 (2000) webml.elet.polimi.it/webml/upload/ ent5/1/www9.pdf

[CIN 00] Cinneide M. PhD Tesis: Automated Application of Design Patterns:A Refactoring Approach. 2000 www.cs.ucd.ie/staff/meloc/home/ papers/thesis/thesis.pdf

[CON 99] Conallen Jim. Modeling Web Applications with UML. CACM 42(10): 63-70 (1999) http://www.conallen.org/whitepapers/webapps/ModelingWebApplications.htm

[DEM 03] Serge Demeyer, Bart Du Bois, Hans Stenten, Pieter Van Gorp. Refactoring: Current Research and Future Trends. Electronic Notes in Theoretical Computer Science 82 No. 3 (2003) win-www.uia.ac.be/u/lore/refactoringProject /publications

[EET 03] Niels Van Eetvelde, Dirk Janssens. A Hierarchical Program Representation for Refactoring. UniGra'03 Preliminary Version win-www.uia.ac.be/u/ lore/ refactoringProject

[FOW 03] Sitio oficial de Martin Fowler. www.martinfowler.com 2003

[GAM 95] Gamma E, Helm R, Jhonson R, Vlissides J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley 1995. ISBN 0-201-63361-2.

[GOM 01] Gómez Jaime, Cachero Cristina. Conceptual Modeling of Device-Independient Web Applications. IEEE Multimedia April June 2001. págs 26/38.

[GOR 00] Pieter Van Gorp, Hans Stenten, Tom Mens, Serge Demeyer. Formal UML Support for the semi-automatic Application of object-oriented Refactorings. win-www.ruca.ua.ac.be /u/bdubois/papers/ 2000

[GUP 97] Gupta A., Mumick I., Rao J., Ross K., Adapting Materialized Views After Redefinitions: Techniques and a performance study. 1997 cnrc.columbia.edu/~kar/pubsk/adaptation.ps

[HUR 96] Hürsch W , Seiter L. Automating the evolution of object oriented systems. www.ccs.neu.edu/research/demeter/biblio/HS94-auto-evol.html. ISOTAS 1996: 2-21

[KER 02] DRAFT of Refactoring To Patterns, 2002, Joshua Kerievsky, Industrial Logic, Inc. industrialogic.com/papers/rtp017.pdf

[KIF 92] Querying object oriented databases. Kifer M., Kim W, Sagiv Y. SIGMOD Conference 1992: 393-402 www.cs.sunysb.edu /~kifer/papers.html

[KIM 90] Kim W. Introduction to object-oriented databases- MIT Press 1990.

[LEE 97] Amy J. Lee, Anisoara Nica, Elke Rundensteiner. The EVE Framework: View Evolution in an Evolving Enviroment. Technical report WPI-CS-TR-97-4 Worcester Politechnic Institute davis.wpi.edu/dsrg/EVE/tech/tech.html

[MEN 02] Mens, T., S. Demeyer and D. Janssens, Formalizing behaviour preserving program transformations, in: Graph Transformation, Lecture Notes in Computer Science 2505 (2002), pp. 86/301

[OPD 92] Opdyke W. PhD Tesis: Refactoring Object Oriented Frameworks. 1992 citeseer.nj.nec.com/opdyke92refactoring.html

[PAL 95] Palsberg J., Xiao C., Lieberherr K. Eficient Implementation of Adaptive Software. TOPLAS 17(2): 264-292 (1995) http://www.ccs.neu.edu/home/lieber/Publications.html

[PRE 95] Pree W., Sikora H. Application of Design Patterns in Commercial Domains. OOSPLA'95 Tutorial 11, AustinTexas Oct 95.

[RAE 01] Real Academia Española. www.rae.org.es. Diccionario oficial. Actualización 2001

[REF 03] www.refactoring.com. Catálogo en ..catalog/index.html

[ROB 99] Roberts Donald B. PhD Tesis: Practical Analysis for Refactoring. St-www.cs.uiuc.edu/~roberts/thesis.ps.

[ROS 96] Rossi G. Tesis doctoral: Um Método Orientado a Objetos para o Projeto de Aplicaçao. . www-lifia.info.unlp.edu.ar/~fer/oohdm/. 1996

[ROS 01] Gustavo Rossi, Daniel Schwabe, Robson Mattos Guimarães. Designing Personalized Web Applications. http://www.inf.puc- rio.br/~schwabe/papers/ WWW10.PDF WWW 2001: 275-284

[ROS 02] Gustavo Rossi, Daniel Schwabe, Juan Danculovic, Leonardo Miaton. Patterns for Personalized Web Applications. www.mmmbook.com/xprogrammer/source/ EuroPlop2002-Koch.pdf

[SCH 98] Daniel Schwabe and Gustavo Rossi, "An Object Oriented Approach to Web-Based Application Design", Theory and Practice of Object Systems 4(4), 1998. Wiley and Sons, New York, ISSN 1074-3224) http://www.inf.puc-rio.br/~schwabe/papers/TAPOSRevised.pdf

[SCH 99] Schwabe D., Vilain P. Notaçao da Metodologia OOHDM versiioin 1.1 inf.univali.br/~adhemar/engenharia/oohdm.pdf 1999

[SEI 96] Seiter L. PhD Tesis: Design pattrns for managing evolution. 1996 http://www.ccs.neu.edu /home/lieber/theses-index.html

[TAX 03] www.program-transformation.org /twiki/bin/view/Transform/CategoryTaxonomy 2003

[TOK 99A] Tokuda L. PhD Tesis: Evolving Object-Oriented Designs with Refactorings swt.cs.tu-berlin.de/lehre/seminar/ss02/Papers/Tokuda99/

[TOK 99B] Tokuda L, Bartory D. Automating three modes of Evolution for Object-Oriented Software Architectures. COOTS 1999: 189-202 www.usenix.org/publications/library/proceedings/ coots99/full_papers/tokuda/tokuda_html

[UML 03] www.rational.com/uml

[WID 85] Widrow B, Stearn S. Adaptive Signal Procesing. Prentice Hall, Inc. 1985.

[XIN 99] Xin Zhang, Elke Rundensteiner. Data Warehouse Mainteneance Under Concurrent Schema and Data Updates. . ICDE 1999: 253 www.cise.ufl.edu/~jgreenbe/papers/18.pdf

[YOD 02] Joseph W. Yoder, Ralph E. Johnson: The Adaptive Object-Model Architectural Style. WICSA 2002: 3-27. http://www.adaptiveobjectmodel.com/WICSA3/ArchitectureOfAOMsWICSA3.htm

Apéndices

A Especificación de elementos del modelo

A.1 Especificación de operaciones rehacer de clases de refactorización (refactoring).

Las especificaciones mostradas, lejos de ser un listado exhaustivo de especificaciones, detalla las operaciones del capítulo final, de ejemplificación del modelo.

A.1.1 Acerca de la lectura de las operaciones

Junto al nombre de la operación se encuentra el nombre del autor de la misma.

Son versiones modificadas respecto de las del autor, en las que:

Se modifican las asociaciones, en consonancia con los cambios en las referencias.

Se relaciona las modificaciones con el mapeo al modelo navegacional.

Las instrucciones referidas a las asociaciones se encuentran en negrita cursiva

Las instrucciones referidas a mapeo se encuentran en negrita.

Para resaltar los aspectos de especificación se utiliza sintaxis Java, sin embargo, para evitar los detalles de implementación, en ciertos casos se utilizan expresiones analíticas.

A.1.2 AbstractAccess (Cinnéide)

Condiciones iniciales: La clase *contexto* originalmente ve a la clase *concreta* mediante una referencia. Esta operación crea una nueva referencia hacia la interfase *inf*, denominada “un”+string(inf), la que será usada por aquellos métodos expresados en el conjunto *conjuntoMetodos*. En la figura A.1 se muestra el efecto de la aplicación de la refactorización.

Si *conjuntoMetodos* es el total de métodos, elimina la referencia anterior a la clase *concreta*.

Agrega la definición de la nueva asociación denominada *contexto* + *“con”*+ *inf*.

Es una versión modificada de la de O’Cinnede, que tiene en cuenta, las referencias no usadas por métodos (eliminándolas), y creando las definiciones de las asociaciones nuevas.

Si un método es implementación de uno declarado en la interfase, es decir, es parte del conjunto *conjuntoMetodos*, modifica la referencia asociación con la clase concreta a una referencia a la asociación con la interfase.

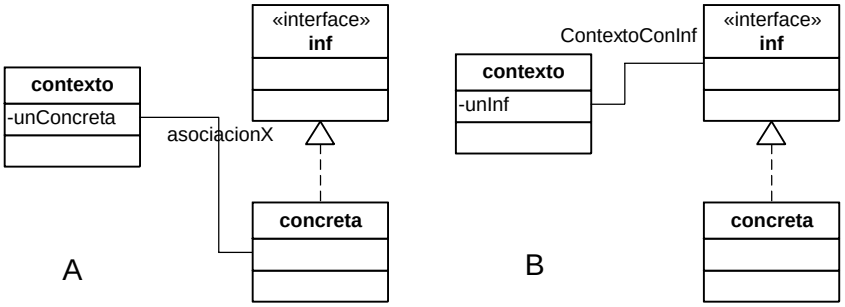


Figura A1: (A) Estado original del esquema. (B) Luego de la aplicación de AbstractAccess.

```
class abstractAccess{
        private boolean mapear ;
        /* Constructor: conjuntoMetodos es el conjunto de métodos, que se declaran en la
interfase.*/
        AbstractAccess(String contexto, String concreta, String inf, set conjuntoMetodos){
                mapear = true;
        }
        public run(Object mapear, Object cambiarAsociacion){
        rehacer();
                /* si la operación de mayor nivel es la encargada de mapear, esta no mapea */
                if (mapear != null)
                        this.mapear();
        }
        public rehacer(){
                private boolean borraReferencia;
                addVariable(String refInterfase="un"+string(inf)); /* agrega refInterfase del
        tipo inf */
                addAssociation(End1(contexto,navigable=False),End2(inf,navigable=True
                ),reference = refInterfase,label = string(End1) + "con" +string(End2));
                ∀ o:ObjectRef, typeOf(o)=concreta, containingClass(o)=contexto
                        if (nameOf(containingMethod(o)) ∈ conjuntoMetodos){replace(o,inf )}
                        else {borraReferencia = False};
                        refConcreta = nameOf(typeOf(o)=concreta);
                        if borraReferencia {remove(refConcreta)};
                        if borraReferencia {removeAssociation(ref= refConcreta)};
                };
        //Mapeo:
        public mapear(this){
        ∀ nodo:n
        ∀ MiembroNavegacional:Mn, Mn ⊂ n
                if ∃ MiembroConceptual:Mc, Mc ⊂ definicion(Mn) ∧ Mc ⊂ conjuntoMetodos
```

```
//La nueva asociación es con la interfase
                    rename_association(n, Mn,refConcreta,refInterfase)
        }
}
```

A.1.3 Abstraction(Cinnéide)

Crea una interfase a partir de una clase concreta, agrega el vínculo implements a la clase concreta respecto de la interfase. En la figura A.2 se muestra el efecto de la aplicación de la refactorización.

Es una operación que no modifica las asociaciones conceptuales, ni impacta en los objetos navegacionales, ya que solo agrega una interfaz.

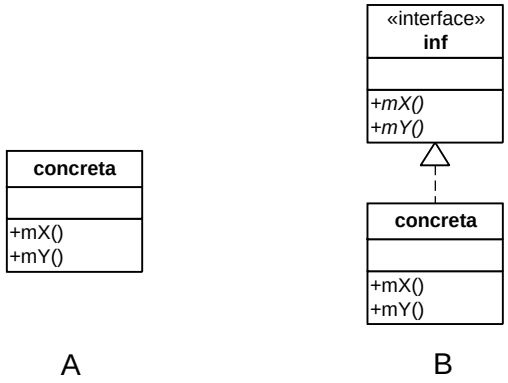


Figura A.2 : (A) Estado original del esquema. (B) Luego de la aplicación de Abstraction.

```
class abstraction{
private boolean mapear = false ;

        // String[] setOfMethods es un agregado, para decidir que métodos contiene la
interfase.
        Abstraction(String concreta, String nombreInterfase, String[] setOfMethods){..};
        rehacer(){
                Interface inf = abstractClass(concreta, nombreInterfase, String[]
setOfMethods);
                // Crea la inf, copia la firma de los métodos  setOfMethods
                AddInterface(inf); //agrega la interfase al programa
                addImplementsLink(concreta, nombreInterfase);
                // agrega implements la intrfase a la clase concreta
}
        public run(Object obj){
                rehacer();
                /* no hay mapeo, la operación de abstracción solo agrega , no impacta en los
        objetos navegacionales */
                                                }
}
```

A.1.4 AccesoPorInterfase

Genera una interfase *nombreInterfase* a partir de una clase *concreta* para los métodos especificados en *conjuntoMetodos*. Modifica las referencias de las clases clientes contexto hacia la interfase. Está compuesta Abstraction y AbstractAccess como se muestra en la figura A.3 No se muestra figura de su aplicación, ya que es la aplicación secuencial de aquellas.

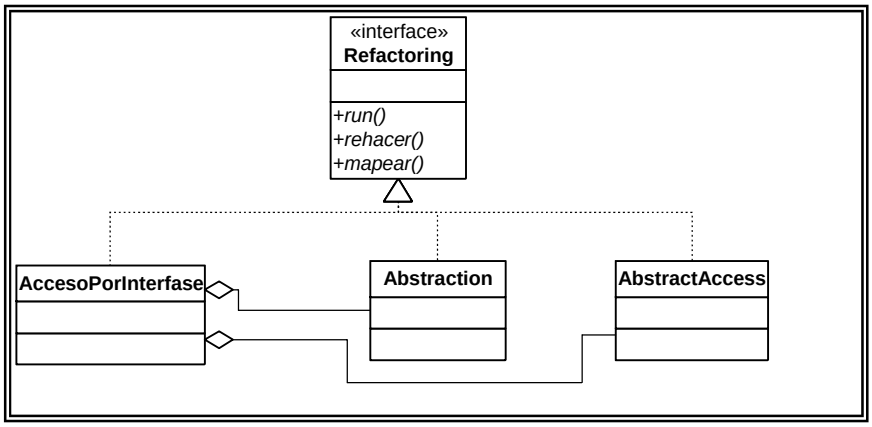


Figura A.3: Composición de Acceso por interfase.

```
class accesoPorInterfase{
private boolean mapear = false ;

/*Constructor*/
accesoPorInterfase(String contexto[],String concreta,String nombreIntrfase,set
conjuntoDeMetodos){..}

rehacer(){
        unaAbstraction = new Abstraction(concreta,nombreIntrfase, conjuntoDeMetodos);
        unaAbstraction.run(); //no mapea, no cambia asociaciones

        ∀ contexto:c
                unAbstractAccess = AbstractAccess( c, concreta, nombreIntrfase,set
        conjuntoDeMetodos);
                unaAbstractAccess.run(null, this); //se bloquea el mapeo, si cambia
        asociaciones
}

mapear(){
```

∀ nodo:n; ∀ MiembroNavegacional:Mn, Mn ⊂ n
if ∃ MiembroConceptual:Mc, Mc ⊂ definicion(Mn) ∧ Mc ⊂ conjuntoMetodos
rename_Association(n,Mn,refConcreta,refAbstracta);
}

```
public run(){
        rehacer();
        mapear();
        }
}
```

A.1.5 Delegation(Cinnéide)

Mueve parte de una clase existente *context* a una clase componente *delegationName* y crea una asociación denominada context + "con" + delegationName, navegable desde la primera a la segunda, referenciada como *"un"* + *delegationName*, entre la clase existente y su componente. En la figura 20 se muestra un esquema de la composición de Delegation. En la figura A.4 se muestra el efecto de la aplicación de la refactorización.

Si la parte de context, movida a delegationName, implica el movimiento de una asociación con una tercera clase, actualiza la misma.

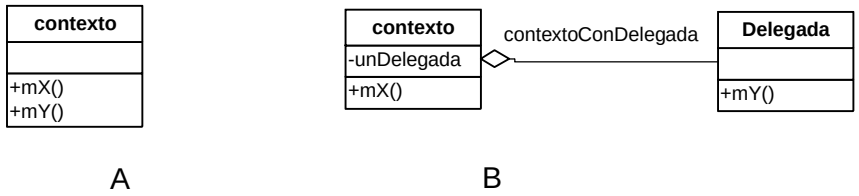


Figura A.4: (A) Estado original del esquema, (B) delegación de mY() en la nueva clase Delegada.

```
Class Delegation{
        AddClass ac;
        CreateExclusiveComponent cec;
        MoveMember mm;
        Delegation(String context, SetOfMembers conjuntoMiembros, String
delegationName){
                ac = new AddClass(delegationName);
                cec = new CreateExclusiveComponent(context, delegationName, "un" +
        delegationName, private);
                mm = new MoveMember(context, delegationName, conjuntoMiembros);
        }
        rehacer(){
                ac.run();
```

```
            cec.run(null,null);
            mm.run();
            if (cambiarAsociacion!=null)
                addAssociation(End1(contexto,navigable=False),End2(classOf(delegationName),navigable=True),reference = delegationName,label = string(End1) + "con" + string(End2));
}

    public mapear(){
        ∀ nodo:n
            ∀ MiembroNavegacional:Mn, Mn ⊂ n
                if ∃ MiembroConceptual:Mc, Mc ⊂ definicion(Mn) ∧ Mc ⊂ conjuntoMiembros
                Insert_association(n, Mn,string(End1) + "con" +string(End2), Mc)
                }
    run(Object mapear, Object cambiarAsociacion){
        rehacer();
        if (mapear!=null)
            mapear();
        }
}
```

A.2 Especificación de asociaciones

Nota: Esta definición se refiere a asociaciones binarias, de navegación única

Nombre: String

End1: Class ; isNavigable: boolean

End2: Class ; isNavigable: boolean

//Referencia es un agregado a la especificación UML. Es una instancia de End1

Referencia1: Object

A.2.1 Operaciones de definición de asociaciones

Las operaciones siguientes son miembros de la clase Associations.

Associations
+removeAssociation(in referencia : String) +addAssociation(in End1 : String, in navegabilidad1 : Boolean, in End2 : String, in navegabilidad2 : Boolean, in

Figura A.5: clase Associations

A.2.2 Agregar asociación

Descripción: Crea la especificación de una nueva asociación

Firma:

addAssociation(Class End1(boolean navigable), Class End2(boolean navigable), Object reference ,String label);

A.2.3 Eliminar asociación

Descripción: elimina las referencias a una asociación

Firma:

removeAssociation(Object reference);

A.3 Funciones de apoyo al mapeo. Clase Maps

Basado en el análisis de impactos de modificaciones del modelo conceptual sobre el navegacional se sintetizan ahora los tipos necesarios de adaptaciones que requiere este último. Es decir las operaciones mediante las cuales se mapearán los cambios en el modelo navegacional.

A.3.1 Remoción de nodos

Descripción:

- Elimina el nodo cuyo nombre es *nombreNodo.*
- Requiere de la remoción previa de todas las referencias a *nombreNodo* existentes en objetos navegacionales secundarios y en los primarios ya sea por agregación/composición o que heredan de este.

Firma:

Remove_node(String nombreNodo)

Comentarios:

Esta operación se dispara cuando fue eliminada la clase padre del nodo y no es reemplazada por otra, o cuando se eliminó el objeto base o su clase.

A.3.2 Cambio de nombre de la clase base

Descripción:

Actualiza el nombre de la clase base en el encabezado de definición de un nodo.

Firma:

Rename_class(String nombreNodo; String viejoNombre; String nuevoNombre)

Comentarios:

Esta operación puede dispararse por un cambio en el nombre de la clase base.

A.3.3 Modificación de la clase base

Descripción:

Actualiza el nombre de la clase base en el encabezado de definición de un nodo

chequea previamente la equivalencia de los miembros que componen la firma de las clases y que son a su vez usados en las definiciones de los atributos nodales. Aquellos atributos que referencian miembros no existentes en la nueva clase son eliminados.

Previamente deben haber sido actualizadas las asociaciones que parten de la clase base, aunque corresponde a tareas de las asociaciones y no de nodos.

Firma:

Change_class(String nombreNodo; String viejoNombre; String nuevoNombre)

Comentarios:

Esta operación se dispara por un cambio de la clase que se constituye como clase base

A.3.4 Cambio de tipo de atributos nodales

Descripción:

Cambia el tipo del atributo *nombreAtributo*.

Firma:

Change_type_atribute(String nombreNodo; String nombreAtributo; String nuevoTipo)

Comentarios:

Esta operación se dispara por el cambio del tipo del miembro servidor.

A.3.5 Remoción de atributos nodales

Descripción:

Elimina el atributo del nodo

Firma:

Remove_attribute(String nombreNodo; String nombreAtributo)

Comentarios:

Esta operación puede dispararse por desaparición del miembro servidor del modelo conceptual.

A.3.6 Cambio de nombre de atributos nodales

Descripción:

Cambia el nombre del atributo *nombreAtributo* por el *nuevoNombreAtributo*.

Firma:

Rename_attribute(String nombreNodo; String nombreAtributo; String nuevoNombreAtributo)

Comentarios:

Esta operación puede dispararse por cambio de nombre del miembro servidor o por cambio del miembro servidor en si.

A.3.7 Cambio de referencia a un miembro conceptual

Descripción:

Cambia la referencia del nodo desde un miembro del modelo conceptual hacia otra.

Firma:

Change_reference(String nombreNodo; String nombreAtributo, String referencia)

Comentarios:

Esta operación puede dispararse por cambios en el miembro servidor como por ejemplo luego de la operación de refactoring create_method_accessor.

A.3.8 Inserción de asociaciones

Descripción:

Agrega la referencia a una asociación en la cadena de definición de un atributo nodal, la asociación debe ser parte de un camino de asociaciones conceptuales entre el objeto base y el miembro servidor.

Firma:

Insert_association(String nodo; String atributo; String nombreAsociacion; String elementoDerecha)

Comentarios:

Esta operación se dispara cuando se agrega una clase en el camino de asociaciones entre el objeto base y el miembro servidor.

A.3.9 Eliminación de asociaciones

Descripción:

Quita la referencia a una asociación en la cadena de definición de un atributo nodal, en correspondencia con el camino de asociaciones conceptuales entre el objeto base y el miembro servidor.

Firma:

Remove_association(String nodo; String nombreAsociacion)

Comentarios:

Esta operación puede dispararse cuando en el modelo conceptual se quita una clase en el camino de asociaciones entre el objeto base y el miembro servidor.

A.3.10 Cambio de nombre de asociaciones

Descripción:

Cambia la referencia a una asociación.

Firma:

Rename_association(String nodo; String atributo; String viejoNombre; String nuevoNombre)

Comentarios:

Esta operación puede dispararse como parte de un cambio de alguna asociación en el camino de asociaciones entre el objeto base y el miembro servidor.

A.4 Especificación de nodos

La definición de los atributos nodales se expresa mediante el *path* de referencias y asociaciones existente entre un objeto de la clase base del nodo y el miembro al que hace referencia el atributo:

NODO N FROM obj: Clase1 [INHERIT FROM : NP]

.

Ai: obj.mR

Aj: obj asociacion1 asociacion2 ... asociacionN-1 mS

Ai, Aj son atributos nodales

obj es una variable hace referencia a la instancia en particular que observa el nodo **N**.

mR es el miembro de la clase Clase1 de la cual **obj** es instancia y que provee contenido al atributo **Ai**.

asociacion1 asociacion2 ... asociacionN-1 es la cadena de asociaciones del modelo conceptual. En esta cadena:

asociacion1:

End1 : Clase1

End1 : Clase2

Reference: <una referencia a un objeto de Clase2>

.

.

AsociacionN-1:

End1 : ClaseN-1

End1 : ClaseN

Reference: <una referencia a un objeto de ClaseN>

mS es miembro de la clase ClaseN.

.

Definición alternativa:

Prescinde de las asociaciones bajo la definición UML, los path expressions se construyen con las referencias con las que un objeto hace referencia a otro:

NODO N FROM obj:Clase1 [INHERIT FROM : NP]

Ai: obj.mr

.

Aj: obj.ref1.ref2.....refn.mS

.

donde ref1 es un miembro de la clase Clase1 de la que obj es instancia por el que referencia a un objeto de clase Clase2, ref2 es un miembro de la clase Clase2 por el que referencia a un objeto de clase Clase3, finalmente refn es la referencia a la clase que contiene el miembro mS.

En la figura A.6 se muestra un esquema de referencia gráfica a amos tipos de definiciones.

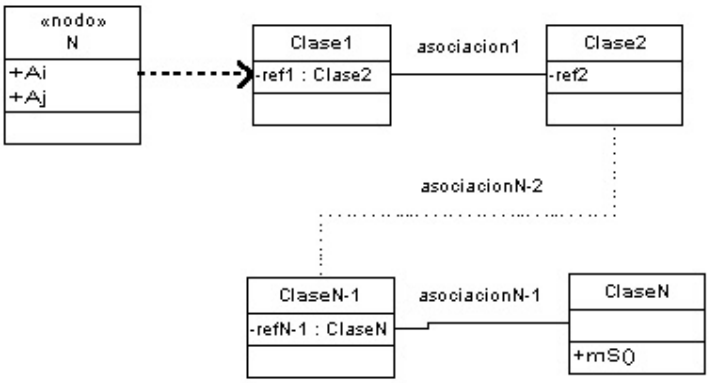


Figura A.6: Esquema de referencia para definición de atributos nodales.

B Refactoring

B.1 Notas acerca de refactoring

B.1.1 Transformaciones y refactoring

De acuerdo a la definición dada en lo que para la mayoría de los autores en el trabajo fundacional, la tesis doctoral de [OPD 92], refactoring es una transformación de un programa orientado a objetos que preserva su comportamiento. la interpretación del alcance de la preservación del comportamiento es unánime en torno a las características funcionales de un programa, aunque discutida en torno de las no funcionales como tamaño del programa, tiempos de ejecución de tareas, etc.

Refactoring es una categoría particular contenida en la de las transformaciones de software, que engloba a un conjunto cuya taxonomía, propuesta en [TAX 03] es la siguiente:

- Translation
    - Program Migration
    - Program Syntesis
        - Program Refinement
        - Program Compilation
        - Code Generation
    - Reverse Engeneering
        - Decompilation
        - Architecture Exctraction
        - Documentation Generation
        - Software Visualization
    - Program Analysis
- Rephrasing
    - Program Normalization
    - Program Optimization
        - Program Specialization
        - Deforestation
        - Super Compilation
    - Program Refactoring
        - Program Obfuscation
    - Software Renovation / Reengeneering

donde además se presentan como sinónimos de transformación a:

- Meta Programming
- Software Generation
- Generative programming
- Progam synthesis
- Program refinement
- Program calculation

B.1.2 Definición y objetivos de refactoring

M. Fowler, en su sitio oficial [FOW 03], define refactoring como el proceso de cambio de un sistema de software de manera tal que no altera el comportamiento externo del código, mientras si mejora su estructura interna.

El aspecto distintivo de refactoring respecto del resto de las transformaciones es que modificando su estructura interna, no altera (preserva) el comportamiento del software. Esto significa que ante los mismos estímulos de entrada al programa antes y después de la reestructuración, la respuesta resultante será la misma. Esto no solo incluye los aspectos semánticos del programa, sino también los sintácticos, y es ese sentido que esté libre de errores.

Los estudios fundacionales importantes datan de principios de en los ’90, destacándose de los iniciales la ya citada tesis de Opdyke. Los demás trabajos importantes sobre refactoring son posteriores a la aparición del trabajo de GoF acerca de design patterns, permitiendo desarrollar operaciones complejas enfocadas a soluciones comunes en la reestructuración de programas. Los patterns son estados finales deseables en los programas, cuya aplicación automatizadas ocupa gran parte de las metas actuales de refactoring.

En el fondo del tema refactoring está la reusabilidad. Esto es, convertir código existente para lograr que sea reusable de manera de que el software existente esté en condiciones de soportar nuevos aportes. En términos de refactoring, hay que distinguir entonces, los nuevos aportes o servicios de un programa, y el soporte para brindarlos. Particularmente, el eje del trabajo de Opdke está en brindar soporte al diseño iterativo de frameworks.

La meta de automatización, requiere que la introducción de código se realice en forma disciplinada. Sin embargo, además de la automatización, también es importante la

abstracción lograda mediante el conjunto de operaciones, permitiendo analizar e implementar transformaciones desde una perspectiva alejada del “bajo nivel”.

Otro aporte de refactoring, compartido y complementado con design patterns, es el de proveer un modo común a los programadores y diseñadores de reestructurar programas orientados a objeto.

B.1.3 Enfoques para refactoring

Los enfoques para controlar la preservación del comportamiento ante la reestructuración varían entre formales, semiformales y no formales. Además, dentro de cada enfoque se han desarrollado diferentes estrategias para abordar los cambios. Por otra parte, y con diferentes características, los enfoques con cierto grado de formalidad tienen en común el análisis de ciertas condiciones en el programa, previas a la aplicación de los cambios, las que se denominan condiciones iniciales.

Enfoque formal

En la línea de los enfoques formales Hursch y Seiter [HUR 96] presentan una línea dirigida a frameworks orientados a objetos que consiste de:

1) la descripción formal de las dependencias entre los principales componentes de un framework orientado a objetos, como uno de los elementos más destacables.
2) una definición de equivalencia en el comportamiento, también formal.
3) un modelo de proceso para mantener la consistencia y comportamiento entre componentes.

Se sustenta en el concepto de minimización de dependencias entre componentes, en una línea diferente a la de los patterns de GoF, a partir de la Ley de Demeter y el SW orientado a objetos propuesto por Lieberr [PAL 95], donde la estructura y el comportamiento del programa están mínimamente acoplados.

Enfoque desde lenguaje de diseño

Otra línea es la que plantea reformular el lenguaje de diseño, para lograr por un lado abstracción del lenguaje de implementación y por el otro evitar pensar las operaciones complejas como composición de primitivas. Luego desde el diseño se plantea modificar el código. En esta línea se ubican:

- El trabajo [GOR 00] que tiene el objetivo de explotar UML como lenguaje de diseño y pensar refactoring en ese nivel, abstrayéndose de la sintaxis particular de lenguajes (y sus dialectos). El objetivo es finalmente usar UML como base de generación

de código. Sin embargo, muchas de las operaciones de refactoring tienen que ver con la definición de métodos. Un ejemplo trivial como rename_class, puede modificar el nombre de una clase referenciada en el cuerpo de un método, y por lo tanto es una modificación con consecuencias no observables desde UML. Esta tipo de situaciones, más el agregado de herramientas para el chequeo de condiciones iniciales y finales, los lleva a proponer un conjunto de extensiones a UML para dar soporte al refactoring de programas.

- El trabajo [MEN 02] aborda la complejidad de ciertas transformaciones, desde una propuesta de representación del código fuente de aquellas partes del programa cuyo comportamiento debe preservarse, basada en grafos, junto con un conjunto de reglas de reescritura de los grafos para cumplir con esas condiciones. Considera que las debilidades actuales derivan de que la definición enfoca el tema desde el tiempo de ejecución del programa, mientras que las herramientas atacan el problema sobre el código fuente principalmente a través de las pre y post condiciones. El trabajo se presenta como un estudio de factibilidad, para ver si la reescritura de grafos puede representar formalmente aquello que es exactamente preservado cuando se realiza refactoring. Concluye positivamente, expresando que: los grafos permiten expresar los cambios en forma independiente del lenguaje en forma concisa y a la vez precisa, como así también probar la premisa de preservación del comportamiento.

- En [EET 03], se propone flexibilizar la representación anterior, jerárquicamente, dada la necesidad de cambiar la representación completa cuando se producen pequeños cambios, fenómeno que genera grandes esfuerzos de mantenimiento en aplicaciones complejas.

Estas propuestas requieren de una relación unívoca desde el diseño al código, que es quién en definitiva debe modificarse si el objetivo es la automatización de la aplicación de refactoring.

Enfoques orientados al código

En una línea orientada a refactorizar directamente el código de lenguajes industriales (en particular C++), Opdyke presenta un conjunto funcional de operaciones de refactoring, dividida en operaciones en bajo (26 operaciones mostradas en B.1) y alto nivel, compuesta por las primeras.

Opdyke

Para evitar cambios en el comportamiento Opdyke plantea una lista de invariantes (condiciones iniciales) que deben cumplir las operaciones, seis detectables en tiempo de compilación como:

- La existencia de una única superclase por clase
- Nombres de clases distintos en el programa
- Nombres de miembros distintos por clase y en la jerarquía (siempre que no sobreescriban)
- Variables miembros no redefinidas en subclases
- Firmas compatibles en funciones miembros redefinidas en subclases
- Asignaciones correctas de tipos según las características de tipificación del lenguaje.

Tienen que ver con aspectos sintácticos. Solo se verifican algunas (las necesarias) por cada operación. Como ejemplo, en la figura B.1 , se presenta la operación c*reate_member_function* con el correspondiente chequeo de condiciones que debe cumplir para ejecutarse, en base a los invariantes.

En los aspectos semánticos, plantea condiciones que son especificadas para cada operación, que Opdyke engloba como un séptima condición a cumplir. Por ejemplo, cuando una variable es movida de una clase a otra a través de una asociación, las referencias a la misma deben actualizarse donde quiera que se encuentren.

El testeo de las condiciones se efectúa mediante el uso de un conjunto predefinido de funciones de diferente grado de complejidad, invocadas según la operación de refactoring a realizarse.

La condición de igual comportamiento en las operaciones simples es testeada respecto del cumplimiento de siete condiciones o invariantes. En cambio la condición de igual comportamiento en las últimas es testeada respecto de las simples, ya que son sus componentes.

También aborda, como autor fundacional el tema del dominio de las operaciones, el cual está muy enfocado en C++, considerando clases y sus miembros, de todo tipo de visibilidad. Incluye en el dominio la función main(), como única excepción a funciones no miembros de clases.

create member function.

Arguments: function F, class C.
Preconditions:
1. $\forall$ memberFunction $\in$ C.locallyDefinedMemberFunctions, memberFunction.name $\neq$ F.name.

2. $\forall$ F2 $\in$ inheritedMemberFunctions(C), (F2.name = F.name) $\Rightarrow$ matchingSignatureP (F, F2).

3. $\forall$ F3 $\in$ functionsThatOverride(F), matchingSignatureP(F, F3).

4. $\forall$ F2 $\in$ inheritedMemberFunctions(C), (F2.name = F.name))
($\forall$ class $\in$ C $\cup$ subclassesOf(C), unrefdOnInstancesP(F2, class)) (semanticallyEquivalantP F, F2).

Las tres primeras condiciones corresponden a invariantes sintácticos, la cuarta a un invariante semántico: si la nueva función colisiona con una existente, el cuerpo de la misma debe adaptarse a la de la función reemplazada asegurando consistencia.

Figura B.1: Precondiciones para Create_member_function

Tokuda y Bartory

En la misma línea, los trabajos de Tokuda y Bartory [TOK 99B] están orientados a soportar los patterns de Gamma, en cuyo sentido agrega las operaciones complejas que se muestran en el apéndice B.2.3 y hots-spots meta patterns de Pree [PRE A]. En la figura B.2 se muestra una descripción de la operación de introducción del pattern Singleton según Tokuda.

Tokuda y Bartory postulan además, sin demostrar, que si las operaciones individuales que componen la compleja preservan el comportamiento, esta también lo hace.

Singleton

singleton[C]
Purpose: Crear una clase que tiene solo una instancia.
Arguments: **C (nombre de la clase a crear)**
Enabling Conditions:
1. **C** debe ser una única clase.

Figura B.3: Refactoring para Singleton

Están basados en el método de Opdyke en cuanto al testeo de las 7 invariantes (que denomina condiciones habilitantes) como condición inicial para la realización de las operaciones, aunque agrega 3 invariantes dirigidos a evitar errores en la instanciación de clases, como por ejemplo las producidas por referencias a clases abstractas que en un estado anterior del programa lo eran a clases concretas, y un cuarto invariante relacionado a problemas relativos al cambio de tamaño y al layout del programa.

Cinneide

Mel Ó Cinneide [CIN 00] también continúa en la línea iniciada por Opdyke. Aunque replantea el análisis de las condiciones iniciales en dos aspectos.

Por un lado considera que, si bien los objetivos al refactorizar son comunes, como por ejemplo introducir un pattern, el estado inicial desde donde se aplicará la operación de refactoring normalmente difiere de una aplicación a otra. Algunas aplicaciones, suelen tener algunos elementos relacionados con la introducción de un pattern y faltarles otro. En otros casos, que es la situación contemplada por los autores de refactoring, no hay ninguna clase, interfase o miembro creada en el sentido de la introducción de un pattern. Esta situación se denomina “green field”.

Cinneide considera que las condiciones iniciales con las que debe enfrentarse la aplicación de una operación de refactoring varía desde “green field” hasta estados avanzados, cercanos al estado final que resulta de la aplicación de la operación. El estado, distinto del “green field” desde donde parte la refactorización se denomina *precursor*. Por ejemplo, en el caso de Factory Method, la clase Creator probablemete exista en el programa, creando y usando instancias de Product, como se muestra en la figura B.3 ,pero no con la flexibilidad que permite la fácil extensión a otras clases Product, como en definitiva es el objetivo del pattern.

También considera la posible presencia de antipatterns, esto es, una mala solución que hay que revertir hacia una buena solución y que no representa ni una situación de green field ni un antipattern.

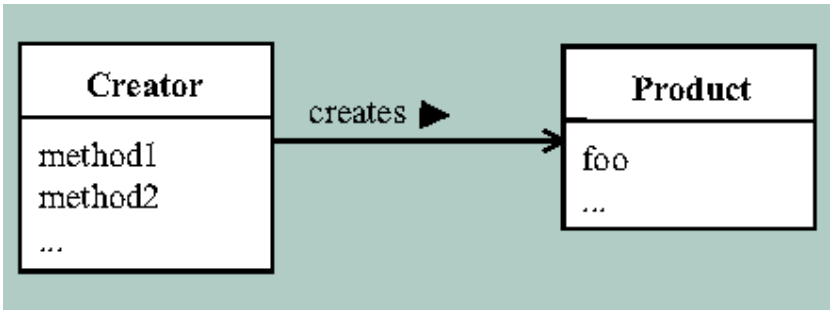


Figura B.3: Precursor de Factory Method

En cuanto a la condición de igual comportamiento, Cinneide se autodefine como semiformal, ya que considera imposible para lenguajes industriales testear formalmente esta condición, por lo que aplica el chequeo de condiciones en un sentido similar a Opdyke. Para la representación de las reglas de análisis de condiciones iniciales utiliza lógica de predicados de primer orden.

Las operaciones de menor nivel de complejidad, no divisibles en otras más simples las denomina operaciones primitivas. Crea funciones de análisis que tienen por misión chequear las condiciones iniciales necesarias para la preservación del comportamiento del programa y las funciones de ayuda con las que extrae información del programa que está siendo transformado, como por ejemplo la firma de un método para construir una interfaz. Operaciones primitivas, funciones de análisis y funciones de ayuda se muestran en B.2.2.

Las minitransformaciones son operaciones compuestas en base a las anteriores y se muestran también en B.2.2. Finalmente estas y las anteriores sirven a la composición de operaciones de introducción de los design patterns de Gamma.

Roberts

Donald Robers [ROB 99] se basa en las operaciones primitivas de Opdyke, pero centrándose en la construcción de operaciones complejas, principalmente patterns, como una secuencia estricta de las anteriores. En el análisis de la secuencia genera una regla de dependencia entre dos refactorings R1 y R2: Si R2 no puede aplicarse legítimamente sin aplicarse previamente R1, entonces R2 es dependiente de R1. Si en cambio pueden conmutarse, entonces son independientes.

Genera un nuevo paradigma de refactoring, eliminando gran parte del análisis previo de carácter estático y realizando el refactoring en tiempo de ejecución, agregando postconditions, o análisis posterior a la implementación de las operaciones.

Dada la dificultad de verificar la condición de igual comportamiento, redefine refactoring (hecho cuestionado por Cinnede) como:

Un par R=(pre,T,P), donde pre es la precondición que el programa debe cumplir, y T la transformación a realizar y P se refiere a las postcondiciones. Pre es especificada con cálculo de predicados de 1º orden

Además presenta una herramienta de refactoring llamada “Refactoring Browser” desarrollada en Smalltalk donde encapsula en objetos las operaciones, las herramientas de análisis.

Otros

Un enfoque en el que no se encuentra testeo de condiciones de preservación de comportamiento es el de Fowler [REF 03] , en cuyo sitio se puede encontrar también un catálogo de operaciones desarrolladas en su código Java.

Una situación similar presenta [KER 02] en cuanto a la carencia de testeo de condiciones de preservación de comportamiento. También tiene desarrollado una catálogo con su correspondiente código Java desarrollado. Propone además una lista de refactorings hacia la aplicación de patterns, a llevar a cabo a partir del resultado de algunos husmeadores de código (code smells) que describen potenciales problemas en el código. La lista se muestra en la tabla siguiente:

Smell	Refactorings
Conditional Complexity	*Move Embellishment to Decorator (73)* *Replace Conditional Calculations with Strategy (50)* *Replace State-Altering Conditionals with State (154)*
Combinatorial Explosion	*Replace Conditional Searches with Specification (99)*
Duplicated Code	*Chain Constructors (19)* *Introduce Polymorphic Creation with Factory Method (43)*
Inappropriate Intimacy	*Encapsulate Composite with Builder (64)*
Indecent Exposure	*Encapsulate Classes with Creation Methods (27)*
Large Class	*Extract Creation Class (34)*
Long Method	*Compose Method (119)* *Move Accumulation to Collecting Parameter (94)*
Long Parameter List	*Replace Conditional Calculations with Strategy (50)*
Primitive Obsession	*Replace Implicit Tree with Composite (60)* *Replace Conditional Calculations with Strategy (50)* *Replace Enum with Type-Safe Enum (146)* *Replace State-Altering Conditionals with State (154)* *Replace Conditional Searches with Specification (99)*
Switch Statement	*Replace Conditional Calculations with Strategy (50)*

Miscelaneas

En [DEM 03] se mencionan algunos formalismos usados en refactoring, entre los que se destacan:

Program slicing: para extracción de funciones o procedimientos, esta técnica está basada en grafos que representan las dependencias del software, que son usados para garantizar la condición de igual comportamiento.

Graph transformations: (ya citado en el presente capítulo) el software mismo es representado mediante un grafo, y las transformaciones mediante transformaciones del grafo.

Software metrics: se usan para medir la calidad del software y detectar la necesidad de producir refactoring.

También cita autores que proponen la aplicación de técnicas existentes, basadas en: especificaciones algebraicas, máquinas de estado finito, cálculo, etc.

En relación a los lenguajes de implementación, los trabajos de Tokuda y Opdyke están orientados a refactorizar programas en lenguaje C++, mientras que los de Cinneide a Java.

B.1.4 Herramientas

Se han desarrollado herramientas específicas para refactorizar el código de diversos lenguajes

Para Python, Luciano Ramalho desarrolló *Python transcription* [RAM (www.hiper.com.br/python/refactor/index.html)

Jean-Christophe Zeus desarrolló herramientas para Perl y C++ se encuentran en (jczeus.com/[refac_perl | C++] respectivamente, desarrolladas por.

Una lista de herramientas para lenguajes específicos se encuentran en www.refactoring.com/tools.html[www.program-transformation.org /twiki/bin/view/Transform/]

B.1.5 Conclusiones

El conjunto de autores cuyos trabajos se orientan a modificar el código, no tienen salvo algunas excepciones, nomenclaturas ni definición de composición de operaciones comunes respecto de refactoring. Esto habla de cierta inmadurez de la disciplina.

Un tema central en los trabajos más importantes es el de la preservación del comportamiento, respecto del cual los niveles de formalidad de tratamiento son variados. Los autores que se independizan del código, haciendo uso de grafos tienden a ser más formales al observar el cumplimiento de la definición. Los que se orientan al código inevitablemente definen ciertas heurísticas con las que axiomáticamente tratan la premisa de igual comportamiento. una excepción es la tesis de Roberts, quien dada la dificultad de dar tratamiento formal a la premisa de igual comportamiento, deja la definición de lado, cambiándola por un conjunto de condiciones que garantizará cumplimiento la operación a llevarse a cabo.

B.2 Operaciones de refactoring

Por simplicidad, no se dan los argumentos de las funciones ni las condiciones iniciales y finales.

B.2.1 Operaciones de refactoring según Opdyke.

En esta clasificación se encuentran 26 operaciones básicas, las transformaciones más complejas se realizan en base a la combinación de estas.

1. **Creación de entidades:**
   (a) create empty class
   (b) create member variable
   (c) create member function.

2. **Eliminación de entidades:**
   (a) delete unreferenced class
   (b) delete unreferenced variable
   (c) delete member functions

3. **Cambio de entidades:**
   (a) change class name
   (b) change variable name
   (c) change member function name
   (d) change type
   (e) change access control mode
   (f) add function argument
   (g) delete function argument
   (h) reorder function arguments
   (i) add function body
   (j) delete function body
   (k) convert instance variable to pointer.
   (l) convert variable references to function calls
   (m) replace statement list with function call
   (n) inline function call
   (o) change superclass

4. **Movimiento de variables:**
   (a) move member variable to superclass
   (b) move member variable to subclasses

5. **Operaciones de nivel intermedio**
   (a) abstract access to member variable
   (b) convert code segment to function
   (c) move class

B.2.2 Operaciones de refactoring según Cinnéide.

En esta clasificación se encuentra una primera división entre funciones de análisis y funciones de ayuda, que sirven para extraer información del programa que está siendo transformado, y las operaciones primitivas de refactoring, que son funciones básicas de refactorización.

Operaciones primitivas y funciones de análisis

- **Funciones de análisis y de ayuda**
  1 absorbParameter
  2 abstractClass
  3 argument
  4 classCreated
  5 classOf
  6 constructorInvoked

7 containingClass
8 containingMethod
9 contextFree
10 createEmptyClass
11 createsSameObject
12 createWrapperClass
13 declares
14 de.nes
15 equalInterface
16 exhibitSameBehaviour
17 hasSingleInstance
18 implementsInterface
19 initialises
20 isAbstract
21 isClass
22 isClonable
23 isExclusiveComponent
24 isInterface
25 isPrivate
26 isPublic
27 isStatic
28 isSubtype
29 localVars
30 makeAbstract
31 methodsInvoked
32 nameOf
33 noOfArguments
34 noOfParameters
35 parameter
36 returnsObject
37 returnsSameObject
38 returnType
39 sigOf
40 superclass
41 superclasses
42 typeOf

- **Funciones primitivas de refactorización**

1 addClass
2 addGetMethod
3 addImplementsLink
4 addInterface
5 addMethod
6 addSingletonMethod
7 createExclusiveComponent
8 makeConstructorProtected
9 moveMethod
10 parameteriseField
11 pullUpMethod
12 replaceClassWithInterface
13 replaceObjCreationWith. . .
14 useWrapperClass

Minitransformaciones y patterns

Transformaciones más complejas se construyen en base a estas. La búsqueda del autor está orientada a la construcción de operaciones que permitan introducir patterns, en ese camino genera 6 operaciones de nivel intermedio deniminadas minitransformaciones:

- Abstraction
- EncapsulateConstruction
- AbstractAccess
- PartialAbstraction
- Wrapper
- Delegation

Finalmente, en base a estas y las anteriores construye las transformaciones necesarias para introducir algunos de los pattens GoF.

B.2.3 Operaciones de refactoring según Tokuda.

Esta clasificación es similar a la anterior, en cuanto la división entre operaciones básicas y operaciones de introducción de patterns. Aunque hay diferencia en gran parte de las operaciones básicas respecto de las de Cinnéide.

Operaciones de refactorización del esquema de clases

1 add_variable
2 create_variable_accessor
3 create_method_accessor
4 rename_variable
5 remove_variable
6 push_down_variable
7 pull_up_variable
8 move_variable_across_object_boundary
9 create_class
10 rename_class
11 remove_class
12 inherit
13 substitute
14 rename_method
15 remove_method
16 push_down_method
17 pull_up_method
18 move_method_across_object_boundary
19 extract_code_as_method
20 declare_abstract_method
21 structure_to_pointer

Operaciones de inrtoducción de patterns

22 add_factory_method
23 create_iterator
24 composite
25 decorator
26 procedure_to_command
27 *singleton*

B.2.4 Atomización de operaciones de refactoring

Add_variable(..)	
Add_method(..)	
Create_method(...)	
Pull_up_variable(..)	Remove_variable(v, Class1) + add_variable(v, Class2)
Pull_up_method(..)	Remove_method(m, Class1) + add_method(v, Class2)
Create_class(..)	
Inherith(..)	
Change_scope_variable(..)	Remove +add
Change_scope_method(..)	Remove +add
Change_type_variable(..)	Remove +add
Change_type_method(..)	Remove +add
Rename_variable(..)	Remove +add
Rename_method(..)	Remove +add
Remove_variable(..)	
Remove_method(..)	
Push_down_variable(..)	Remove +add
Push_down_method(..)	Remove +add
Unhinherit(..)	
Rename_class(..)	Remove +add
Remove_class(..)	Remove +add

Se detallan argumentos solo en los casos de las operaciones pull_up por simplicidad, ya que el objetivo de la tabla es simplemente las operaciones que intervienen al atomizar las de la columna de la izquierda. Además solo se muestran aquellas operaciones cuya atomización es trivial.

Inherit y Uninherit son un caso espacial a tener en cuenta. Desde el punto de vista de una operación sobre el esquema de clases conceptuales son irreductibles, sin embargo, desde el punto de vista del impacto sobre el modelo navegacional, su efecto es agregar (Inherit) o remover (uninherit) atributos y/o métodos.

Dado que cualquier operación que implique agregar variables o métodos al modelo conceptual, es intrascendente a las definiciones de los objetos navegacionales, podemos concluir en base a la tabla anterior, que el impacto está siempre basado en una operación elemental de remoción o uninherit.