Universidad Nacional de La Plata

Facultad de Informática

Metodología dirigida por modelos para el diseño de Funcionalidad Volátil en aplicaciones Web

Autor: Lic. Mario Matias Urbieta

Director: Dr. Gustavo H. Rossi

Tesis presentada para obtener el grado de Doctor en Ciencias Informáticas

- Abril 2012-

Resumen

La popularidad y facilidad de acceso de las aplicaciones Web expone a una aplicación Web a exigencias de nuevas características realizadas por sus usuarios que ésta debe proveer para mantener cautivo al usuario implantando un estado de constante evolución. La evolución requiere usualmente modificaciones de funcionalidad existente o nueva funcionalidad para mejorar la experiencia del usuario en la aplicación Web. Muchas veces estos cambios son requeridos para mantener vigente a la aplicación, es decir acompañar a las tendencias del mercado. Los cambios introducidos pueden corresponder a un tipo de funcionalidad llamado volátil caracterizado por ser temporal, surgir de improviso y muchas veces por deber ser incorporada a la brevedad. Cuando esta funcionalidad es temporal, se incorpora al sistema para luego ser retirada de forma planificada en base a una fecha determinada o de forma espontanea en base a un evento de negocio. En este escenario, entre otras variables, se ve comprometida la mantenibilidad y estabilidad de la aplicación. Por otro lado, su inesperado surgimiento usualmente no permite una adopción fácil y económica ya que la aplicación no fue diseñada teniendo en cuenta esta nueva funcionalidad.

En esta tesis se presenta una metodología modular para dar solución a los requerimientos volátiles en aplicaciones Web. La metodología abordará el problema desde las etapas análisis brindando herramientas conceptuales para su adecuado diseño y posterior implementación. Es modular ya que puede complementar las metodologías de ingeniería Web más maduras; en esta tesis se utilizara como metodología de referencia OOHDM. En la etapa de análisis de requerimientos, se proveerán herramientas que permitan identificar, aislar, y gestionar inconsistencias de requerimientos volátiles. Para las tareas de diseño se proveerán herramientas teóricas que faciliten el modelado de los requerimientos de las aplicaciones Web brindando instrumentos para los diferentes modelos involucrados: conceptual, navegacional, y de interfaz. Finalmente, se proveerá una guía de implementación de éste tipo de funcionalidad con un análisis comparativo con la implementación de funcionalidad volátil ad-hoc.

Agradecimientos

A la mujer de mi vida Cecilia

Contenido

Capítulo 1 Introducción

1.1 Motivación

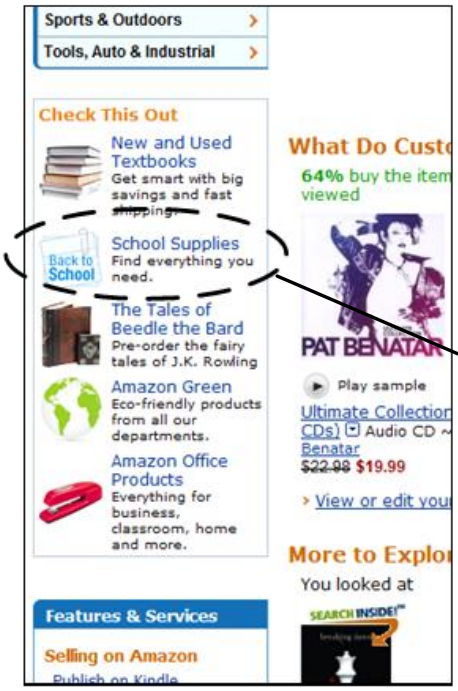
Una de las características más destacables de las aplicaciones Web es su continua evolución en respuesta a nuevos requerimientos o solo con el propósito de mantener su atractivo para los usuarios. El contexto actual de Internet expone a las aplicaciones Web a millones de potenciales usuarios para que las utilicen; más precisamente el 29% de la población mundial [1], [2], [3]. Ante tal exposición, el equipo responsable de estas aplicaciones Web se ve agobiado ante la necesidad de satisfacer las necesidades de la mayoría de estos usuarios para capturar/mantener clientes del negocio subyacente. Estas necesidades empujan a las aplicaciones Web a ponerse a la altura de las circunstancias mediante una evolución continua, con el principal motivo de no perder vigencia. Esta evolución no necesariamente resulta de una solicitud específica de algún usuario sino que puede surgir internamente desde la organización que brinda la aplicación Web con el objetivo de mejorar la experiencia del usuario.

Nuevos requerimientos surgen constantemente durante la vida de las aplicaciones Web donde su mayoría son incorporados para ser parte de la aplicación de forma permanente. Por ejemplo, una aplicación Web dedicada al comercio electrónico [4] de productos en general comienza a brindar un servicio de envío por correo de productos adquiridos por los clientes. Esta nueva característica de la aplicación implica la incorporación de un nuevo formulario donde se solicitan los datos del domicilio dónde realizar el envío. Esta nueva funcionalidad es un servicio que ha sido planificado por los responsables del área de ventas y que se encuentra disponible por una variedad de comercios, tanto físicos como electrónicos, donde se pudo comprobado su utilidad y estabilidad. Sin embargo, no todas las nuevas funcionalidades que se pueden presentar en una aplicación Web pueden ser planificadas de forma tal que su incorporación pueda ser diseñada e implementadas como el ejemplo anterior, sino que pueden ser solicitadas de forma espontanea para un evento de negocio en particular y para perdurar por período de tiempo indefinido. Por ejemplo, en el mismo sitio de comercio electrónico del ejemplo anterior, nuevos requerimientos se pueden presentar como ofertas especiales en ciertos periodos del año (Navidad, el día de los enamorados, etc.) para productos específicos, adecuación de contenido específicos correspondientes a nuevos lanzamientos tal como la incorporación de videos, funcionalidad para brindar ayuda luego de una catástrofe, entre otros.

Las funcionalidades que son implementadas una vez, relacionada con un evento esporádico e inesperado, por ejemplo basados en catástrofes naturales o campañas de marketing espontaneas,

y luego removida definitivamente, o son periódicamente activadas durante momentos particulares del año, son denominadas como Funcionalidades Volátiles [5].

En la Ilustración 1 e Ilustración 2 se muestra la promoción "Volver a la Escuela" (Back-to-School) en Amazon.com[6] que tiene como principal objetivo capturar clientes con necesidades bien conocidas adaptando gran parte de la aplicación Web para que la navegación general del éste simplifique el acceso de productos escolares. La Ilustración 1 muestra un link volátil (con una imagen de la promoción y una breve descripción) remarcado con una elipsis punteada en la página principal el sitio que permite navegar a un catalogo de productos que fueron seleccionados para la promoción; esta página puede ser apreciada en la Ilustración 2. Algunos días antes de que las vacaciones de verano terminen y durante el periodo en que las actividades académicas comienzan, los clientes pueden acceder a ciertas ofertas, más precisamente, pueden beneficiarse con el envío gratuito de productos adquiridos dentro de la promoción con algunas restricciones.



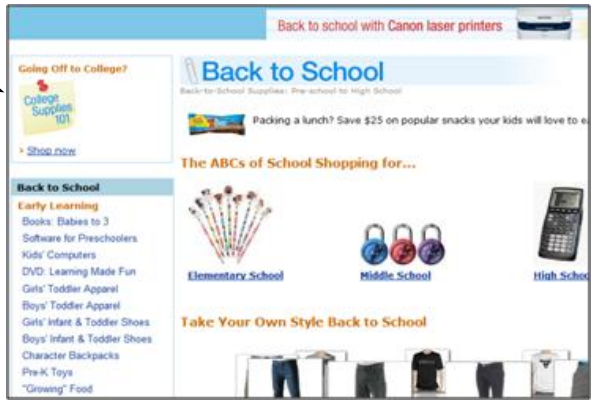


Ilustración 1 Link volátil a las ofertas de Vuelta a la Escuela.

Ilustración 2 Página principal de Vuelta a la Escuela

Los cambios requeridos para incorporar el link (con su imagen y descripción) y página con las promociones pueden parecer simples, sin embargo su incorporación desata cambios que se propagan en todas las dimensiones de una aplicación Web modificando reglas de negocio, estructura de datos, estructura navegacionales, interfaces de usuario, entre otros.

Este tipo de funcionalidad que se introduce en la aplicación Web de forma inesperada no solo está ligada a aplicaciones de comercio electrónico sino que se pueden encontrar en una diversa variedad de aplicaciones tal como Sistemas de Información Geográfico (GIS, *Geographic Information System*) [7], [8], Blogs [9], diarios online [10], entre otros.

Las aplicaciones Web denominadas *perpetual beta* ([11], [12]) debido a su estrategia ágil, fomenta el lanzamiento de nuevas funcionalidades tan pronto como sea posible (Por ej. Cada 30 minutos) para que el mismo usuario perfeccione la aplicación mediante su feedback directo (comentarios explícitos) e indirecto (detectado por los ingenieros de la aplicación a partil del patrón de uso de la aplicación). Estos requerimientos, pueden ser funcionalidades volátiles ya que nuevos requerimientos surgen espontáneamente motivados por analistas y en el caso de no ser adoptado por los clientes son removidos. En aplicaciones Web del tipo *perpetual Beta,* la funcionalidad es introducida de forma incremental como evolución, mayormente de forma transparente, sin romper la estructura y apariencia de la interfaz de usuario. En diferentes ocasiones estas funcionalidades son luego descartadas. De ahora en adelante se referenciará como "*Funcionalidad Beta"* funcionalidades de este tipo de aplicación. A pesar de que la vigencia de ésta funcionalidad es claramente es una decisión de negocio que incumbe a áreas de Marketing, ventas, etc. de una compañía., la gerencia de sistema debe asegurar que el diseño sea también estable y que no esté comprometido cada vez que hay un cambio.

A lo largo de esta tesis se presentarán una variedad de ejemplos para poder destacar la problemática existente asociadas a la funcionalidad volátil en diferentes dominios de negocio (aplicaciones GIS y comercio electrónico), y tipo de aplicación (convencionales y Aplicaciones de Internet Ricas).

1.1.1 Problemática presente en aplicaciones RIA

Muchos de los sitios publicados en internet están evolucionando del hipertexto convencional a Aplicaciones de Internet Ricas[13] (*RIA, Rich Internet Applications*). Incorporando características RIA, estos sitios son capaces de mejorar la usabilidad y en consecuencia atraer una mayor cantidad de potenciales compradores. Las funcionalidades RIA permiten mostrar datos relevantes de productos de forma más novedosa y efectiva. Por ejemplo, en las aplicaciones RIA que gestiona información de productos tal como en el ejemplo de la aplicación de comercio electrónico que se viene utilizando, es posible mejorar la experiencia de la búsqueda de productos de varias formas: (*i*) Una de ellas, al momento de introducir el nombre de un producto a buscar en un campo del formulario de búsqueda, se puede resolver la búsqueda en paralelo a medida que se ingresan los caracteres del nombre mostrando los resultados parciales de dicha búsqueda; de esta forma el usuario puede tomar conocimiento de los resultados más relevantes de forma inmediata presentados en un pop-up. O por otro lado, (*ii*) una vez listados los productos que concuerdan con el nombre ingresado en el formulario de búsqueda, se puede permitir presentar un detalle del producto (valorización, comentarios, imágenes, etc.) que se encuentra debajo del mouse sin

perder el contexto de la búsqueda ya que usualmente los listados presentan información básica de un producto para luego navegar hacia el detalle en otra página. Utilizando el esquema navegacional convencional, la solución requiere navegar hacia la página de un producto para conocer su detalle, requiriendo volver a atrás en la navegación en el caso de que no corresponda al producto deseado. En cambio la solución RIA, permite fácilmente conocer el detalle del producto sin navegaciones adicionales dado que se utilizan características ricas de interacción.

Volvamos al ejemplo del sitio de comercio electrónico. Los usuarios son capaces de navegar a través de un producto para conocer su detalle con funcionalidad RIA para mejorar la experiencia del usuario. Por ejemplo, en la Ilustración 3 se muestra como la interfaz permite intercambiar la foto principal del producto (la imagen que se encuentra del lado izquierdo a la descripción del producto) simplemente posicionando el mouse sobre alguna de las imágenes pequeñas que se encuentran debajo de la imagen principal (señalado con una elipse en la ilustración). Con el objetivo de capturar el conocimiento de los diferentes usuarios y facilitar su transferencia entre ellos, a continuación se presenta un caso particular de funcionalidad Beta donde se permiten anotaciones sobre la imagen del detalle de un producto para comentar diferentes aspectos del producto fácilmente tal como se puede en la Ilustración 4.

Ilustración 3 Interfaz del detalle de un Producto

Con la incorporación de esta funcionalidad RIA (por su taxonomía), el sitio de comercio electrónico satisface la curiosidad del cliente o la falta de conocimiento acerca del producto de una forma concisa con solo una mirada rápida, evitándole la necesidad de leer largos párrafos de descripción al usuario. Esta funcionalidad es una funcionalidad Beta ya que su permanencia en la aplicación dependerá de la adopción que sufra por los usuarios.

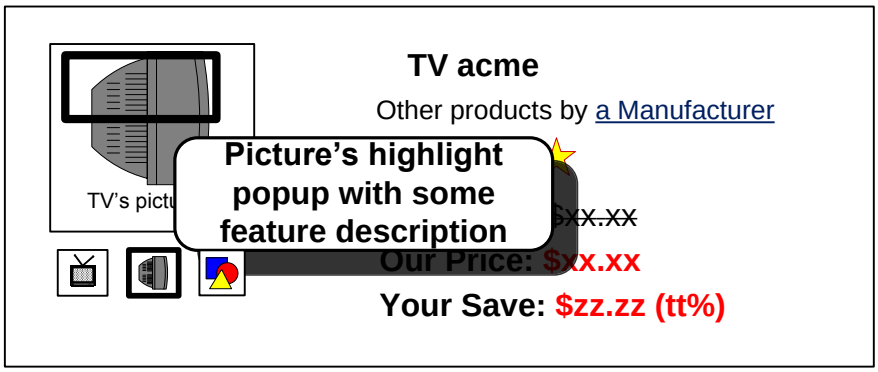


Ilustración 4 Imagen de producto con anotación RIA

Desde el punto de vista de diseño, este tipo de extensiones son usualmente un serio problema para arquitectos y desarrolladores de software debido a que ellos muchas veces no conocen cuándo la nueva funcionalidad será parte de la aplicación o si ésta va a durar por un período de tiempo (por ejemplo cuando el usuario no la "adopta").

1.1.2 Problemática presente en aplicaciones Web GIS

Veamos ahora un ejemplo en un dominio diferente a aplicaciones Web convencionales y RIA. Supongamos que necesitamos extender la aplicación de comercio electrónico con un módulo Web para la gestión del servicio de envío a domicilio de los productos adquiridos. Para mejorar la funcionalidad del sistema, se decidió agregar algunas funcionalidades típicas de sistemas GIS para poder generar reportes de rutas y seguimiento del paquete donde se encuentra el producto. Un reporte de ruta es un plan, utilizado por el conductor de camiones, que provee la ruta más corta desde la compañía hasta el destino del producto. Una vez que el requisito de transporte de encomienda es registrado, el cliente puede ingresar a la aplicación Web de la compañía para dar seguimiento a la ruta seguida por el transporte utilizado donde se encuentra el paquete en tiempo real. Para hacer el ejemplo más concreto, supongamos que nosotros queremos enviar un paquete desde el punto de venta (A) hasta un domicilio particular (B). Para planear el envío del paquete, el sistema debe resolver la ruta más corta entre ambos puntos. La Ilustración 5 muestra el mapa de la ruta que se debería utilizar para el transporte del paquete.

Una situación que surge de forma imprevista y que tiene un tiempo de vida no determinado (pero finito) es el mantenimiento de un segmento de calle (identificado con líneas perpendiculares en la misma imagen) que impide su tránsito. La presencia de este segmento de ruta no disponible genera un impacto severo en la aplicación ya que si ésta no se adapta, la funcionalidad de hoja de ruta dejaría de ser útil. La aplicación debería ser capaz de tener en cuenta ésta situación para evitar el segmento de calle en mantenimiento al momento de elaborar la hoja de ruta. De esta

forma, el mejor camino desde el punto de venta actual hasta el domicilio destino (por ejemplo utilizando la distancia más corta del conocido algoritmo de búsqueda A* [14]) es uno de mayor longitud que el origina como se puede apreciar en la Ilustración 6.

Ilustración 5 Ruta de envío directa de encomienda desde punto de venta (A) hasta domicilio particular (B)

Ilustración 6 Ruta de envío directa de encomienda desde el punto de venta (A) hasta el domicilio particular (B) teniendo en cuenta una refacción temporal de una calles

Si la posibilidad de indicar que un segmento de la calle puede sufrir obras de mantenimiento no fue planeado, y, en consecuencia, diseñado cuando la aplicación fue construida, ésta necesitará ser mejorada mediante el establecimiento de un nuevo criterio de búsqueda de caminos que desprecie segmentos intransitables.

Usualmente, las funcionalidades de búsquedas de caminos son implementadas utilizando algoritmo de búsqueda de grafos tal como el A*. Por ejemplo, este algoritmo consulta los nodos adyacentes de un nodo dado para obtener el nodo adyacente de menor coste que será navegado para intentar alcanzar el nodo destino. La incorporación de la funcionalidad volátil que bloquea calles, afecta el algoritmo A* core, ya que este requiere ahora deberá tener en cuenta en el proceso de búsqueda de los adyacentes a nodos adyacentes que representen nodos bloqueados para no producir resultados inválidos. Este comportamiento puede replicarse en todos los algoritmos de búsqueda utilizados en la aplicación (Dijktra[15], A*, breadth-first [15], etc.).

Adicionalmente, se deberá presentar en la interfaz de usuario, por ejemplo con un color diferente, las calles cortadas para entender el porqué las mismas no son utilizadas en la búsqueda modificando no sólo la capa conceptual de la aplicación sino también la presentación.

1.1.3 Funcionalidad volátil entrelazada

Como se mencionó hasta ahora, las funcionalidades volátiles pueden encontrarse en diferentes tipos de aplicaciones Web convencionales, en aplicaciones Web GIS y RIA. Además de la complejidad inherente de la incorporación de cualquier funcionalidad volátil por su alto impacto en tareas de codificación y gestión de cambios, la variedad de funcionalidades volátiles con la que tiene que lidiar una aplicación complejiza la situación aún más. Varias funcionalidades volátiles pueden convivir en un determinado componente, o grupo de ellos, afectando el mismo conjunto de recursos, componentes, estructuras de datos, etc..

La funcionalidad volátil puede también afectar un subconjunto de objetos irregular; por ejemplo, algunos CDs de un comercio pueden estar involucrados en una promoción, algunos videos de un artista pueden aparecer en ciertas novedades para su promoción, etc..

En la Ilustración 7 se puede apreciar una página del sitio de comercio Amazon[6] correspondiente al CD de Norah Jones. Por otro lado, en la Ilustración 8 se muestra la página del mismo CD que se ve afectado por dos funcionalidades volátiles: un video corto del artista que será retirado luego de algunas semanas, conocido como "Video promocional", y un link a la sección de ventas de "*San Valentín*" (referenciado como *St. Valentine*).

Ilustración 7 Vista del CD de Norah Jones Not Too Late en Amazon.com

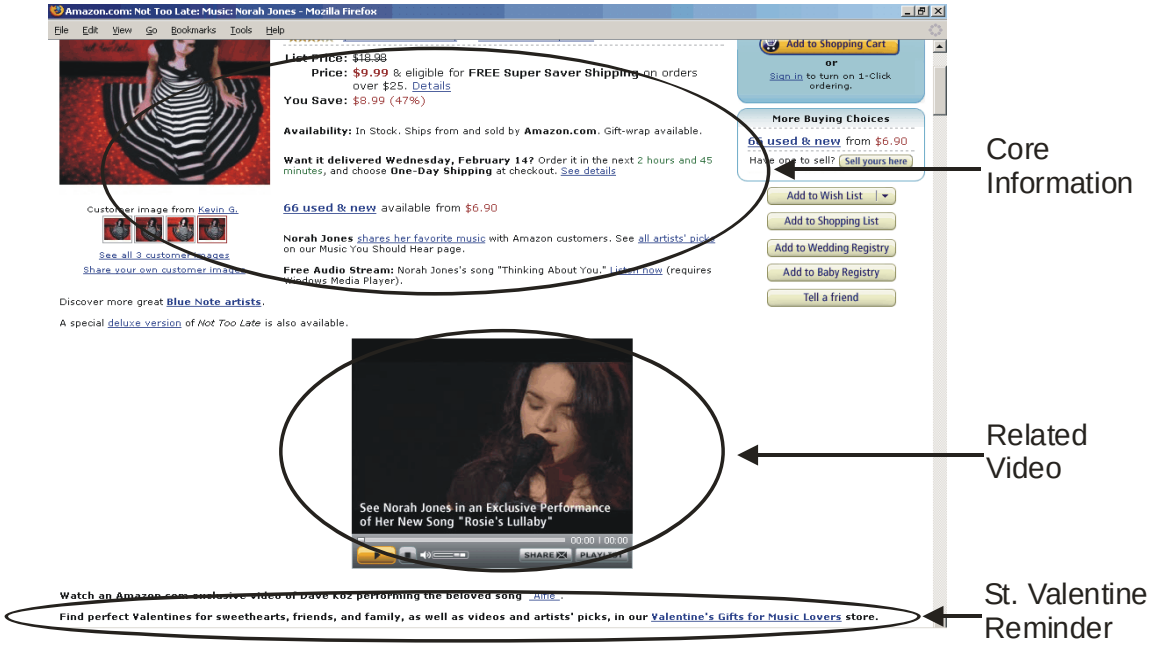


Ilustración 8 Diferentes funcionalidades volátiles en un CD en Amazon.com

Siguiendo con el ejemplo, el período de vigencia del video promocional será establecido por diferentes eventos de negocio. Si se acaba el stock del CDs, un evento de negocio puede

dispararse para indicar que el video no es más vigente requiriendo así su anulación. En el caso del link a la sección de San Valentín, siendo esta una promoción regida por fechas fijas, probablemente, unos días después de la 14 de febrero[1] los links a la sección de la promoción podrán ser retirados. En esta situación irregular, el mantenimiento de la aplicación se ve desfavorecido cuando las funcionalidad base correspondiente a la presentación de la información del producto es el destino de diferentes funcionalidades volátiles correspondientes a "*San Valentín*" y "*Video relacionado"*. La activación o desactivación de estas dos nuevas características de la aplicación pueden no ser sincrónicas complicando las tareas de mantenimiento y requiriendo así esfuerzo independiente para cada uno en las tareas de modificación del código fuente, pruebas e instalación. Por ello, esto resulta ser una tarea altamente propensa a error por ende incrementando las probabilidades de introducir una falla en la aplicación.

1.2 Resumen de la motivación

A partir los ejemplos expuestos anteriormente, lo que caracteriza todo el problema no es solo la *naturaleza inesperada* del nuevo requerimiento sino también su volatilidad. Estos no son frecuentemente previstos durante el diseño de la aplicación y cada vez es más común que sean válidos por un período de tiempo desconocido. Esta situación compleja caracteriza el problema de volatilidad y su impacto en el desarrollo de software de aplicaciones Web.

Por cada requerimiento volátil podemos identificar un patrón de volatilidad, un conjunto de características de la aplicación y objetos que se ven afectados, y varias características adicionales que serán examinadas en esta tesis.

En general, la incorporación de una nueva funcionalidad volátil en una aplicación Web puede comprender, por ejemplo:

(i) la publicación de nuevos tipos de contenidos,

(ii) la extensión de estructura de datos existentes,

(iii) la creación de nuevas estructuras de contenido,

(iv) la implementación de nuevos comportamientos en la aplicación y operaciones de usuario,

(v) y la modificación del "look and feel" de una página.

Debido a las exigencias de los tiempos de salida al mercado y la ausencia de enfoques sistemáticos, la necesidad de dichas funcionalidades es usualmente resuelta en la etapa de

[1] Fecha oficial en Argentina para el día de los "Enamorados". Conocido también como San Valentín.

implementación: nuevos componentes y código son agregados a la aplicación cuando la funcionalidad tiene que ser incorporada y luego, una vez que cumple su propósito, los mismos son removidos o desactivados. Esta práctica, que comprende la edición iterativa del código fuente de la aplicación, a largo plazo, puede tener un impacto negativo en la aplicación, introduciendo errores debido a que son tareas manuales y deteriorando la calidad.

La funcionalidad volátil a nivel de modelo puede causar problemas; ya que los requerimientos volátiles pueden ser, por su naturaleza intrínseca, imprevistos; los diferentes modelos de la aplicación (conceptual, navegacional, e interfaces) usualmente no están "preparados" para la correspondiente incorporación. Estos últimos deben ser editados dos veces, cuando se agrega y retira la lógica asociada a la funcionalidad volátil; con el riesgo del alterar el código e introducir errores.

El refactoring[16] de modelos de diseño para dar lugar modularmente a las nuevas funcionalidades puede ser un desperdicio de esfuerzo y recursos, debido a que podríamos caer en un sobre diseño de la aplicación "solo" por una pieza de funcionalidad que probablemente desaparecerá más tarde.

Con el propósito de facilitar la incorporación de funcionalidad volátil dentro de aplicaciones Web simplificando su evolución y reduciendo el riesgo de introducción de errores, se definió, a lo largo de varias investigaciones, una metodología que permite el diseño sistemático, implementación y automatización de la activación/desactivación de funcionalidad volátil como parte de un enfoque asistido por modelos.

Esta metodología es modular y, como se describe en el trabajo, aunque es presentado en el contexto del enfoque Object Oriented Hypermedia Design Method (OOHDM)[17], sus principios son aplicables a cualquier metodología de diseño Web. Esta metodología consiste en un conjunto de lineamientos que permiten separar el núcleo, o core, (comportamiento estable de la aplicación) de funcionalidades volátiles (comportamiento temporal) en la etapa de diseño y este es soportado por un framework que permite la integración de funcionalidades volátiles con funcionalidades núcleo en la etapa de validación Cazon (será descripto en las secciones siguientes).

1.3 Contribuciones

La tesis presenta una metodología para la identificación, análisis, diseño e implementación de funcionalidades volátiles. En la que se desglosan las siguientes contribuciones:

- Proveer un marco de trabajo conceptual para caracterizar funcionalidad volátil de acuerdo a sus características. Se promoverá la discusión sobre las características más importantes de las funcionalidades volátiles siendo estas ilustradas con diferentes ejemplos.

- Facilitar una metodología de análisis, diseño e implementación de funcionalidad volátil. Presentar una metodología con herramientas y conceptos que permitirá de forma modular, transparente y no intrusiva la implementación de funcionalidad volátil.
    - Ilustrar, mediante ejemplificaciones las funcionalidades volátiles en diferentes casos de estudio donde se aplica el enfoque cubriendo las etapas del ciclo de desarrollo de software: análisis, diseño, implementación y testing. Los ejemplos cubrirán diversos tipos de aplicaciones tal como E-commerce y GIS para demostrar la eficacia de la metodología
    - Mostrar una metodología novedosa para la identificación de inconsistencia de requisitos de interacción (Usuario-Maquina). Para ello, se realizará una caracterización de inconsistencias en requerimientos de una aplicación Web dependiendo de la taxonomía del conflicto, un método modular para detectar las inconsistencias que fácilmente puede complementar cualquier método de ingeniería Web Maduro sin importar su estilo: ágil o unificado, y un conjunto de ejemplos que ilustran la solución
    - Brindar una metodología sistemática para el diseño de interfaces de usuario RIA. La metodología es una extensión liviana de OOHDM en la cual las interfaces son especificadas utilizando Abstract Data View y aprovechando la naturaleza orientada a objetos de la misma. La metodología también toma prestado algunos conceptos de la orientación a aspectos para lidiar con composiciones *crosscuting* a nivel de interfaz.
- Prototipos de aplicaciones que brindan soporte a esta metodología.
    - Se presentará un prototipo para la identificación de conflictos en la etapa temprana de identificación y análisis de requerimientos,
    - y un framework Web que implementa las herramientas conceptuales definidas para el diseño de los modelos conceptuales, navegacionales e interfaz de una aplicación Web.

1.4 Estructura de la tesis

La tesis está estructurada en los siguientes capítulos:

Capítulo 2 ***"Background***": en esta sección se presentarán métodos, herramientas y conceptos en los que esta tesis utiliza como base para el desarrollo de la solución a la problemática disparada por las funcionalidades volátiles. Por ejemplo, se presentará OOHDM debido a que es la base de la metodología desarrollada en esta tesis, o Programación Orientada a Aspecto debido a que se utilizan conceptos de este paradigma en la solución desarrollada en esta tesis.

Capítulo 3 ***"Trabajos relacionados***": se referenciarán trabajos relacionados directa o indirectamente a la problemática surgida por la funcionalidad volátil. Según se requiera, se destacarán las diferencias del trabajo discutido con la solución propuesta en esta tesis.

Capítulo 4 ***"Caracterización de funcionalidad volátil***": se analizarán las características fundamentales de la funcionalidad volátil. Se establecerá un marco de trabajo para facilitar su análisis.

Capítulo 5 ***"Metodología de desarrollo de funcionalidad volátil***": en este capítulo se introduce la metodología que brinda soporte a las funcionalidades volátiles. Se presenta una descripción general cada uno de los pasos de la metodología para luego ser será desarrollado en las secciones siguientes. La metodología cubre casi todo el proceso de desarrollo de una aplicación Web, desde la identificación de los requerimientos hasta su implementación.

Capítulo 6 "Análisis de requerimientos": en este capítulo se presenta uno de los aportes más novedosos de la tesis, una metodología para la detección y resolución de conflictos de interacción Web en la etapa de análisis de requerimientos. Se introduce el concepto de conflicto de requerimientos de interacción, se lo caracteriza y provee una solución modular que puede ser utilizada en otras metodologías de diseño de aplicaciones Web.

Capítulo 7 ***"Modelo Conceptual"*:** a partir del esquema de trabajo definido en el ***Capítulo*** 5 "***Metodología de desarrollo de funcionalidad volátil***", en este capítulo se describe cómo modelar los objetos de negocios volátiles utilizando diagramas UML y MATA como herramientas principales. Posteriormente se discutirá como llevar los modelos diseñados en base a una Orientación a Aspectos a un paradigma Orientado a Aspecto para aquellas plataformas que no soporten AOP.

Capítulo 8 ***"Modelo navegacional"*:** al igual que en el modelo conceptual, en este capítulo se proveen herramientas conceptuales para diseñar el aspecto navegacional de una funcionalidad volátil. Además, se describirá brevemente como diseñar e implementar funcionalidades volátiles que comprometen el proceso de negocio subyacente de un concern core.

Capítulo 9 ***"Modelo de Interfaz "*:** en este capítulo se presenta otro de los aspectos más novedosos de esta tesis, el diseño aspectos de interfaces de usuario volátiles. Utilizando ADVs y ADV-Charts se diseñan interfaces y mediante extensiones, desarrolladas en esta tesis, se especifican cambios interfaces no invasivos a la interfaz core.

Capítulo* 10 *"Gestión del ciclo de vida de las funcionalidades volátiles": se presenta un Lenguaje Específico de Dominio (DSL) que será utilizado para indicar de qué forma se activarán y desactivarán las funcionalidades volátiles. Se utilizarán ejemplos para describir claramente cada concepto.

Capítulo 11 ***"Framework de soporte de la metodología: Cazon":*** aquí se presenta la arquitectura de un prototipo de framework Web llamado Cazon. Este framework permite implementar los conceptos discutidos en las secciones anteriores.

Capítulo* 12 *"Evaluación técnica de pruebas": se discutirá cuál es el impacto de las funcionalidades volátiles en aplicaciones Web. Habiendo medido aspectos de calidad de código fuente, se compara el impacto de utilizar una metodología de desarrollo convencional en comparación a la propuesta en esta tesis.

Capítulo* 13 *"Conclusiones": como indica su nombre, en este capítulo se concluye la tesis destacando los aspectos principales de la misma.

Capítulo* 14 *"Trabajos futuros": Finalmente, se enumeran las líneas de investigación abiertas en las que se continuará el trabajo.

Apéndice A - *"Abreviaturas"*: Se enumeran y describen cada una de las abreviaturas utilizadas en la tesis.

Apéndice B - *"Producción científica"*: Se enumera la producción científica asociada a la tesis.

Apéndice C - *"Términos"*: Se hace una breve reseña de los términos más importantes utilizados a lo largo de la tesis.

Capítulo 2 Background

2.1 Metodologías de desarrollo Web

El desarrollo de aplicaciones Web involucra decisiones no triviales de diseño e implementación que inevitablemente influyen en todo el proceso de desarrollo, afectando la división de tareas. Los problemas involucrados, como el diseño del modelo del dominio, modelo navegacional y la construcción de la interfaz de usuario, tienen requerimientos disjuntos que deben ser tratados por separado. A partir de esta separación de intereses, surgen las metodologías de desarrollo de aplicaciones Web que permiten especificar los requerimientos atacando cada uno de sus aspectos más importantes: el modelo conceptual, navegacional y de interfaz de usuario. El modelo conceptual define cuáles serán los conceptos/objetos del negocio que serán manipulados en la aplicación. El modelo navegacional permite describir qué información será presentada usualmente agrupada en un Nodo y de qué forma se interactuará con esta información a partir de las relaciones conceptuales; un nodo, por ejemplo, indica el criterio con el que se mostrarán lo objetos de negocio. Finalmente el modelo de interfaz de usuario especifica de qué forma se presentará la información al usuario y como éste la percibirá en términos de elementos visuales.

Las metodologías de Web maduras tal como OOHDM, UWE, WebML, OO-H, entre otras (ver [17] para conocer en detalle cada una) son ejemplos de metodologías que facilitan el diseño de una aplicación Web atacando los aspectos (conceptual, navegacional y de interfaz de usuario) por separado.

A continuación se presentará OOHDM como metodología de cabecera en ésta tesis. Sin embargo, los conceptos que se desarrollarán dentro de esta tesis pueden ser migrados a las metodologías de diseño de aplicaciones Web nombradas anteriormente.

2.1.1 OOHDM

Object-Oriented Hypermedia Design Method (OOHDM)[18] es un método orientado a modelos para el desarrollo de aplicaciones Web. Este método permite a los diseñadores especificar una aplicación Web mediante el uso de varios meta-modelos especializados: conceptual, navegación y de interfaz de usuario. Cada meta-modelo pone foco en diferentes aspectos de una aplicación. Una vez que estos modelos han sido especificados para una aplicación dada, es posible generar código en tiempo de ejecución que implemente la aplicación; es decir, los diseños de la aplicación. Existen varios intérpretes de estos modelos encargados de producir una aplicación Web basada en ellos : el ambiente HyperDE[19] y el framework Cazon[20].

OOHDM utiliza mecanismos de abstracción y composición diferentes en un framework orientado a objetos, para permitir por un lado, una descripción concisa de elementos de información correspondientes al negocio subyacente y, por el otro lado, la especificación de escenarios de navegación complejos en base a los datos del modelo conceptual y transformaciones de interfaz para hacer percibible la información indicada en el modelo anterior. Los principios de OOHDM han sido aplicados sobre una variante del método, SHDM[21], en la cual el modelo de datos es utilizado está basado en RDF y RDFS[22].

En OOHDM una aplicación Web es construida en un proceso de cinco pasos: análisis de requerimientos; diseño conceptual; diseño navegacional; diseño de interfaz e implementación. Cada paso de diseño se enfoca en un aspecto de diseño particular, y, en consecuencia, un modelo es elaborado a partir de esto.

A continuación se resumen los cinco pasos tomando como ejemplo ilustrativo una aplicación Web de catalogo de películas y series de televisión. Esta aplicación Web permite la navegación de información relacionada a las diferentes películas tal como en IMDB[23]. Por cada película se conoce su director, actor, genero, comentarios de usuarios, etc.. Las relaciones con otras películas pueden ser navegadas tal como "primeras partes", "continuaciones" o películas con el mismo director entre otros criterios. Una descripción completa y descriptiva del ejemplo puede ser leída en [17].

2.1.1.1 Análisis de requerimientos

El paso inicial paso es obtener los requerimientos de los stakeholders[2]. Para esto, es necesario primero identificar los actores y las tareas que ellos deben realizar en casos de uso. Luego, los casos de uso son acopiados (o bosquejados) para cada tarea y tipo de actor utilizando Diagramas de Interacción de Usuario[24] (UID, *User Interaction Diagram*). Estos diagramas proveen una representación concisa utilizando una metáfora grafica del flujo de información entre el usuario y la aplicación durante la ejecución de una tarea. Los UIDs son validados con el actor del caso de uso y rediseñado, si fuese necesario, a partir del retorno que haya brindado este último.

En Ilustración 9 se muestra el UID correspondiente al caso de uso "Buscar película por nombre". Con las elipses se grafican los paso de interacción (*Interaction*), dentro de estos se enumeran los elementos de datos que participarán de la interacción tanto de escritura (utilizando un rectángulo) o de solo lectura (sin detalle). También es posible indicar conjuntos de datos usando puntos suspensivos "…" como prefijo al nombre del dato. La transición de una interacción a otra se especifica mediante una flecha llamada transición (*Interaction*) y las operaciones que son

[2] En la bibliografía en ingles se utiliza el término de *stakeholder* para indicar todos aquellas personas interesados en el sistema tal como los gerentes de área con sus intereses comerciales, los usuarios con sus necesidades operativas, etc..

disparadas desde una interacción dada se especifican con una línea que se desprende de la interacción con una cabeza redonda y rellena en el otro extremo.

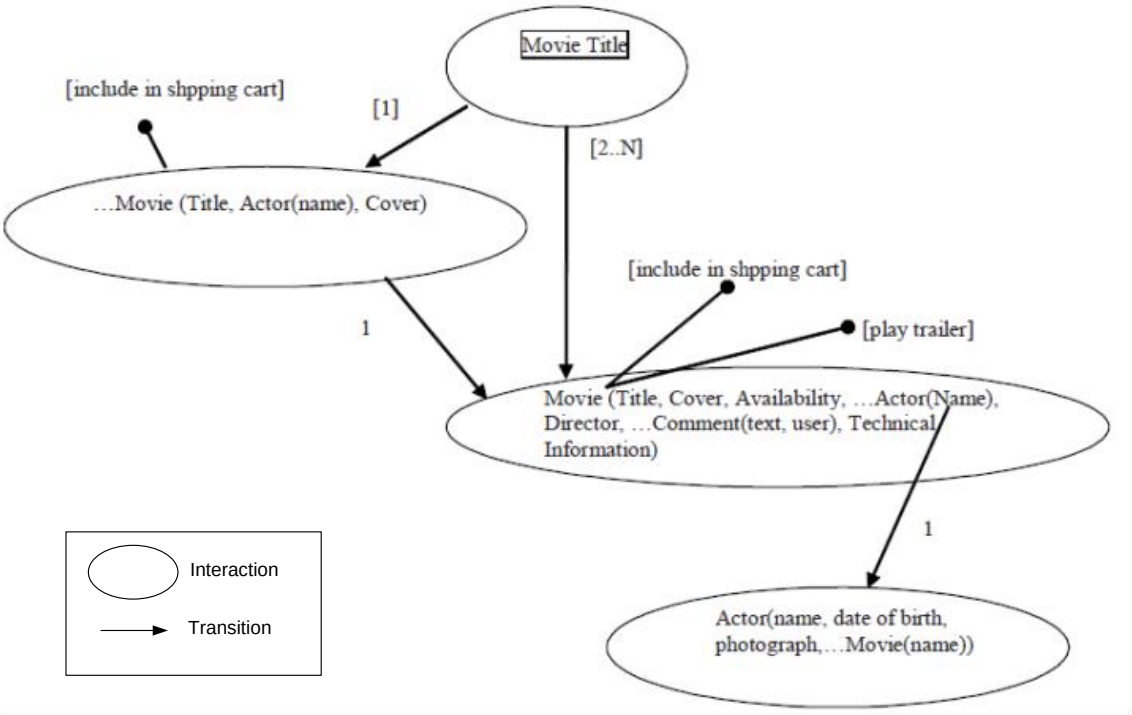


Ilustración 9 UID correspondiente al caso de uso de "Buscar película por nombre" [17]

En este proceso de relevamiento de requerimientos, una guía con reglas y heurísticas[24] permiten extraer un modelo conceptual básico a partir de los UIDs.

2.1.1.2 Diseño conceptual

En este paso se elabora un Modelo Conceptual de dominio de la aplicación utilizando principios de modelado orientados a objetos. En este paso solo se enfoca en la semántica del dominio de aplicación y no en los tipos de usuarios y tareas. OOHDM utiliza el meta-modelo de clases de UML[25] (*Unified Modelling Language*), con pequeñas extensiones, para expresar el diseño conceptual.

La Ilustración 10 corresponde al modelo conceptual de la aplicación simplificado en detalle para mejorar su legibilidad destacando las entidades pero evitando detalles como la navegabilidad de las relaciones y su aridad. Este modelo se obtiene a partir de una versión básica derivada desde los UIDs (siguiendo las reglas definidas en [24]) a la que se le aplicaron iteraciones de refinamientos donde se elimina información redundante; se aplican buenas prácticas de diseño de software tal como generalizaciones y especializaciones, patrones de diseño orientado a objetos[26], entre otro.

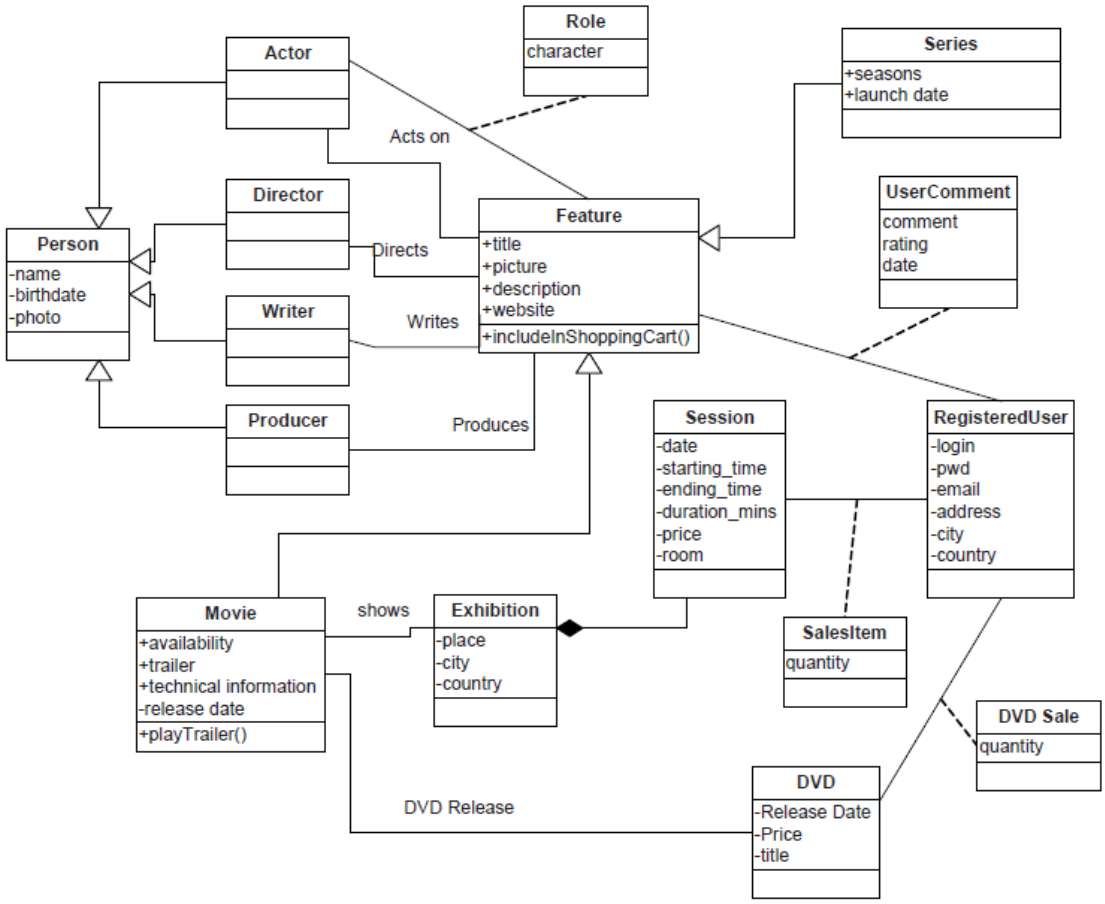


Ilustración 10 Modelo conceptual de la aplicación Web del catálogo de películas [17]

2.1.1.3 Diseño navegacional

En OOHDM, una aplicación Web es concebida como una vista navegacional sobre un Modelo Conceptual. Esto refleja la mayor innovación de OOHDM (que también fue adoptado por UWE[17] y WebML[17]) la cual reconoce que los objetos que el usuario navega no son objetos conceptuales sino un tipo de objetos que se construye a partir de ellos para soportar tareas y una presentación adecuada de la información. En otras palabras, para cada perfil de usuario se puede definir una vista navegacional en base a la tarea que este tipo de usuario debe realizar, dicha vista refleja los datos y relaciones en el esquema conceptual. Estas definiciones pertenecen al Modelo Navegacional. En OOHDM existe un conjunto de tipos de básicos predefinidos de clases navegacionales usuales en aplicaciones de hipermedia: Nodo, Links, Anchors y estructuras de acceso. Los Nodos en OOHDM representan la vista lógica sobre el Modelo Conceptual definidos a partir de un lenguaje de consulta. Desde una perspectiva orientada a objetos, los nodos implementan una variante del patrón Observer[26] ya que presentan una vista particular de los objetos de negocio. Los cambios en el Modelo Conceptual son notificados inmediatamente a los

nodos (objetos observadores) y, por otro lado, los nodos pueden invocar mensajes de los objetos del modelo conceptual a partir de eventos surgidos en la interfaz de usuario. En la actualidad, el patrón Observer es implementado utilizando requerimientos HTTP para actualizar la información presentada al usuario bajo demanda y solo un conjunto de tecnologías basadas en el uso de Javascript con XML de forma asincrónica [13] (AJAX, *Asynchronous JavaScript and XML*) tal como Google Web Toolkit[27] (GWT) soporta la notificación de cambios tal como lo indica el patrón.

Cuando se habla de Links, nos referimos a la realización hipermedial de las relaciones del modelo conceptual así como también las asociaciones. Es importante destacar que pueden existir Links que no reflejen relaciones en el Modelo Conceptual que permiten mejorar, por ejemplo, la navegación de la aplicación. Las estructuras de acceso, tal como los índices, representan puntos de inicio de la navegación.

Aplicaciones Web diferentes (en el mismo dominio) pueden contener topologías de Nodos y Links debido a los perfiles de usuario. En la aplicación de ejemplo Internet Movie Database (IMDB), la vista administrativa de un DVD puede indicar que por cada copia disponible cuándo esta debe ser retornada mientras que la perspectiva de un cliente no lo mostrará.

En la Ilustración 11 se puede ver el Modelo de Navegación compuesto por Nodos y Links que rigen la navegación de la aplicación. Los nodos son usualmente contrapartes del modelo navegacional, sin embargo muchas veces nodos específicos de la navegación pueden definirse como el caso del nodo “Orden” que permite modelar el pedido de compras que está realizando el usuario en el momento y agrupa todas las películas que éste desea adquirir.

Los nodos navegacionales son objetos que pueden poseer métodos que encapsulan lógica específica de navegación de la aplicación tal como la resolución de ciertos datos mediante la ejecución de una consulta sobre el Modelo Conceptual.

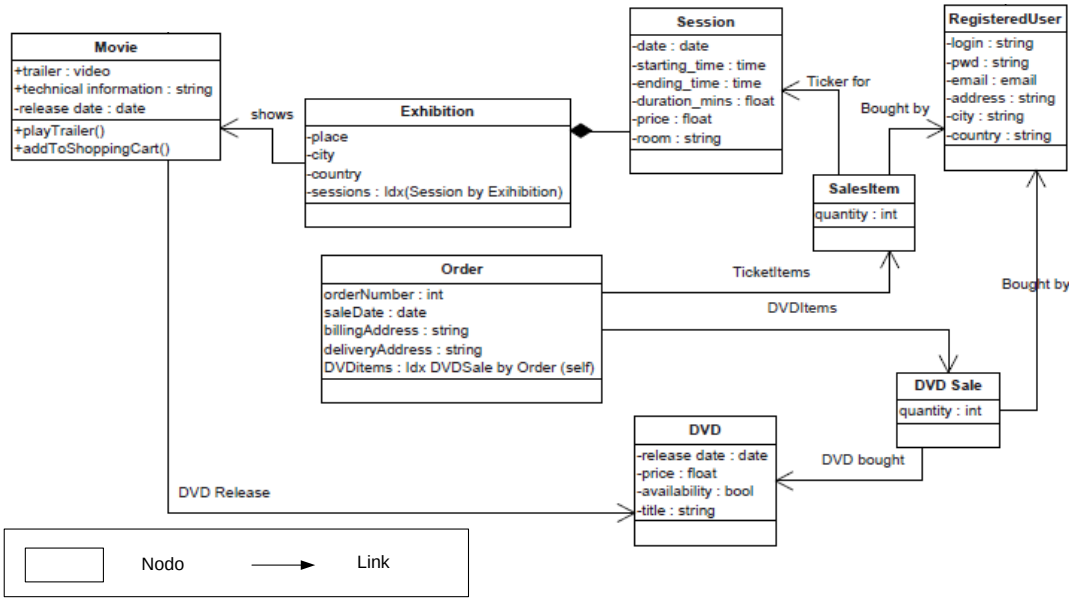


Ilustración 11 Modelo Navegacional reducido del ejemplo de la aplicación Web de catalogo de películas

La mayoría de las aplicaciones Web permiten realizar tareas que involucran un conjunto de objetos que representan conceptos similares tal como *libros por autor, CDs por cantante, hoteles en una ciudad,* etc. OOHDM estructura el espacio de navegación en conjuntos llamados contextos de navegación. Estos poseen las mismas alternativas de navegación y son significativos para cierto paso de alguna tarea que un usuario realice.

Cada contexto navegacional es un conjunto de nodos, y es descripto especificando sus elementos, su estructura interna e índices asociados. Los contextos navegacionales juegan un rol análogo con respecto a la navegación que las clases con respecto a la estructura y comportamiento de los objetos. Por ejemplo, se puede modelar un conjunto de *actores* en una *película* dirigida por un *director*, el conjunto de *copias* de DVD de una *película*, etc.. Para un ejemplo detallado de los contextos navegacionales, ver [17].

2.1.1.4 Diseño de interfaces

El diseño de interfaces se puede desdoblar en dos tareas: el diseño estructural y el diseño de comportamiento. En las dos subsecciones siguientes se describirá cada una de estas tareas.

2.1.1.4.1 Describiendo el contenido de interfaces de aplicaciones Web

OOHDM utiliza Vistas de Datos Abstractas (ADV, *Abstract Data View*) para representar de manera abstracta las interfaces de usuario que requiere una aplicación Web. Un ADV tiene una

estructura (expresada como un conjunto de atributos), comportamiento (definido como un conjunto de mensajes o eventos externos que éste puede manejar) y puede ser definido recursivamente componiendo otros ADVs. Dado su naturaleza estructurada, los ADVs pueden ser mapeados fácilmente en documentos XML facilitando así su escritura y edición. Del lado izquierdo de la Ilustración 12 (mostrada en la motivación), se muestra el ADV correspondiente al componente *ChangeablePicture* de la interfaz Web presentada en Ilustración 4. Este ADV está compuesto de forma anidada por otros ADVs primitivos como Imagen *Picture* o *Text*, mostrando como los componentes serán percibidos por el usuario. Del lado derecho de la Ilustración 12 se muestra la pantalla esperada correspondiente ADV y con líneas punteadas se muestra la relación entre los elementos concretos y abstractos. Es importante notar que la posición de los objetos anidados en el ADV refleja la apariencia (look and feel) de la interfaz siendo de esta forma un medio efectivo para validar con el cliente cómo será el resultado final del desarrollo de la interfaz.

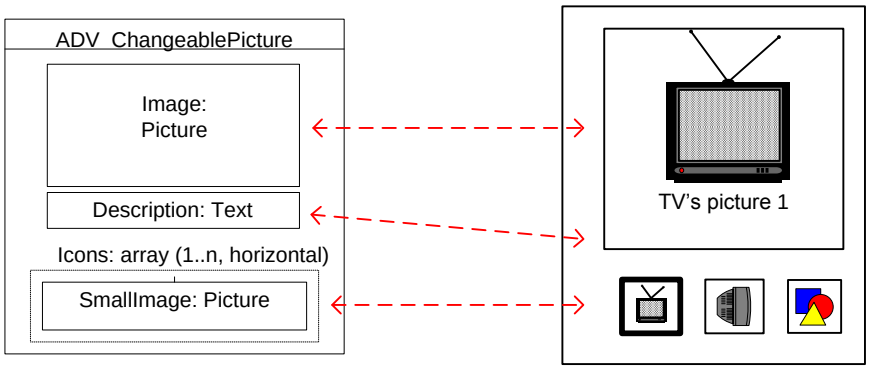


Ilustración 12 ADV e interfaz concreta del componente ChangeablePicture

Los ADVs observan Objetos de Datos Abstractos (ADO, *Abstract Data Objects*), conocidos como "dueños" de ADV, con dos objetivos: ser una vista de los datos y disparar comportamientos de aplicación o interfaz. Los ADV de la Ilustración 12 obtiene sus contenidos del correspondiente ADO; en este caso, un nodo del modelo navegacional tal como puede apreciarse en la Ilustración 13. La relación entre el ADV y su ADO es descripta utilizando diagramas de configuración (*configuration diagram*), una combinación entre clases UML y diagramas de colaboración, mostrando qué mensajes son intercambiados entre el ADV (actuando como cliente de consulta) y el ADO (en role de servidor de datos). En la Ilustración 13, el ADV *ChangeablePicture* resuelve información invocando los métodos *getImage()* y *getText()* del nodo *ChangeablePicture*. Esta información es utilizada para presentar los componentes de la interfaz concreta: *Image*, *Description* y *SmallImage* (arreglo de elementos). El parámetro "i" hace referencia al índice de la imagen seleccionada en el arreglo.

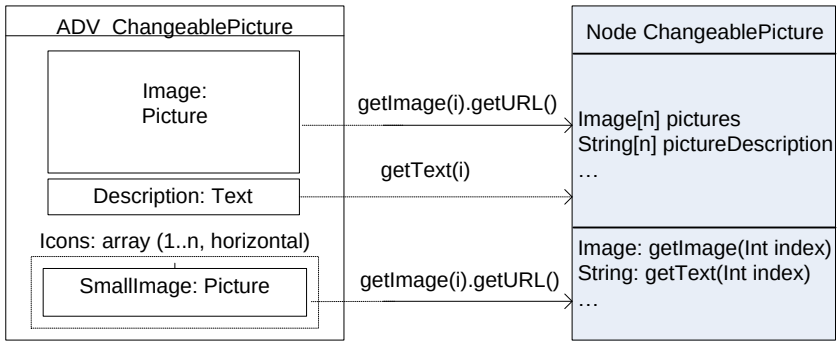


Ilustración 13 Diagrama de configuración para el componente ChangeablePicture

En las aplicaciones Web convencionales, usualmente se especifica un ADV anidado observando un único ADO. Esto puede ser la causa de que el mismo nodo tenga varios ADVs (por ejemplo para proveer muchas interfaces), pero no es común que dos nodos (ADOs) yuxtapongan un único ADV. En aplicaciones Web RIA, podemos necesitar que un ADV consuma información desde diferentes ADOs (nodos) de acuerdo con las características de las interfaces.

Nótese que los aspectos de comportamiento de una interfaz de aplicación Web convencional son simples. Cuando un *anchor* es seleccionado, el ADV del nodo actual debe cerrarse y el correspondiente destino del *link* (otro nodo) debe abrirse. En consecuencia, hay otros comportamientos en aplicaciones Web interesantes pero no simples como, por ejemplo, permitir autocompletado de formularios. Se explicará el mecanismo de expresar comportamientos de ADVs en la próxima Sección (2.1.1.4.2) cuando se trate la problemática de las interfaces dinámicas RIA.

Luego de que la interfaz ha sido completamente especificada, los modelos conceptuales, navegacionales y de interfaz son implementados en plataformas particulares. Con el objetivo de facilitar la adopción de la metodología OOHDM, se ha implementado un framework de ejecución , llamado Cazon [28][20], que soporta la generación semi-automática de código de modelos OOHDM, incluyendo la instanciación de páginas desde modelos navegacionales OOHDM.

2.1.1.4.2 Describiendo la dinámica de RIAs con ADV-Charts

Las aplicaciones Web RIAs están caracterizadas por tener una interfaz dinámica y comportamiento rico que permite mejorar la experiencia del usuario. Mientras que los ADV nos permiten expresar las características estáticas de las aplicaciones convencionales, OOHDM utiliza ADV-Charts para especificar los aspectos dinámicos de este tipo de aplicaciones.

Los ADV-Charts son una variante de las maquinas de estado que permiten expresar las transformaciones de una interfaz que pueden surgir a partir de la interacción con un usuario. Estas son similares a los StateCharts[29].

Como puede apreciarse en la Ilustración 14, una transición en un ADV-Chart es anotada con un identificador (ID), el/los evento/s que la causan, una precondición que debe ser satisfecha para disparar la transición, y una post condición que es obtenido después de que esta transición se procesada. La post condición es expresada en términos de propiedades de objetos que son alterados luego que la transición ocurre.

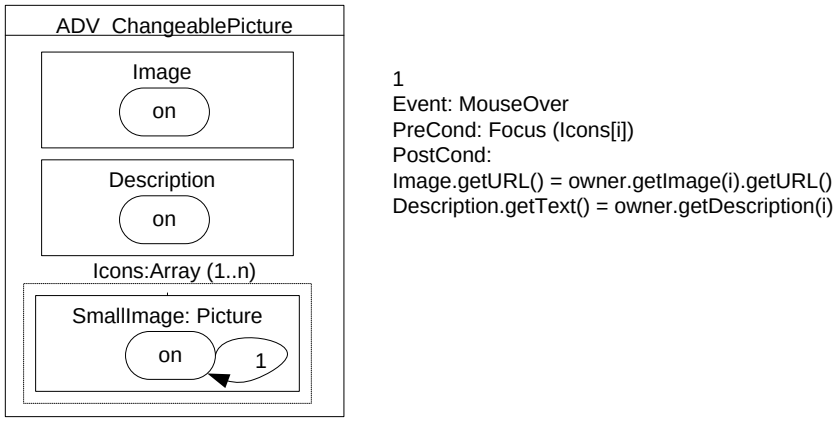


Ilustración 14 ADV-Chart Transición de estados simple

En el ejemplo, también se utiliza la función *Focus* que indica la posición del cursor y una pseudo variable *PerCont* (para referir el contexto de percepción) para indicar los objetos que son percibibles por el usuario. Cuando un objeto es "agregados" al conjunto *PerCont* este pasa a ser percibible por el usuario, caso contrario, cuando es "retirados", este deja de ser visible. La palabra clave *owner* referencia al ADO observado que puede ser utilizado como parte de una definición de transición para consultar su estado.

En la ilustración, se puede ver el ADV-Chart especificando el comportamiento del ADV *ChangeablePicture*: cuando el mouse se encuentra sobre un ícono, los Widget gráficos correspondientes a la imagen actual y la descripción deben ser actualizados con datos icono seleccionado. El objeto propietario, u *owner*, del ADV es consultado por la información requerida en la actualización utilizando la posición en la lista (o índice) como parámetro. La flecha negra hacia el mismo estado señala que el componente *SmallImage* va a retornar al estado inicial luego de que la transición está completa.

Los ADV-Charts pueden componerse(a través del anidamiento de estados) indicando como los ADVs internos son afectados cuando el usuario interactúa con el sistema. Estos pueden también ser utilizados (en combinación con los diagramas de configuración) para indicar la forma en la que las operaciones navegacionales son disparadas por los eventos de interfaz.

Mientras que los estados anidados en un ADV siguen la semántica de los Statecharts, significando que un ADV puede estar en algunos estados (mediante los operadores lógicos "Y" u "O"), el

anidamiento de ADVs dentro de estados indica que el ADV es perceptible por el usuario en ese estado.

2.1.1.5 Implementación

La implementación consiste en mapear los objetos del Modelo de Navegación e Interfaz a una aplicación final y puede requerir elaborar arquitecturas (por ejemplo, Cliente-Servidor) en las cuales las aplicaciones son clientes son Browsers Web para compartir el servidor de base de datos que contiene los objetos del modelo conceptual. Un numero de aplicaciones basadas en DVD como también aplicaciones Web han sido desarrollados utilizando OOHDM como metodología y empleando una diversidad de tecnologías tal como Java (J2EE), ASPX (.NET), Lua (CGILua), ColdFusion y Ruby (Ruby on Rails) para llevar a cabo la implementación.

HyperDe y Cazon son ambientes que implementan de forma nativa los conceptos de OOHDM y están basados en Ruby on Rails[30] y Java respectivamente.

2.1.2 WEBSPEC – DSL para especificar aplicaciones Web

WebSpec[31] es un leguaje visual donde su principal componente para especificar requerimientos es el diagrama WebSpec el cual utiliza *interacciones*, *navegaciones* y *comportamiento rico* para especificar requerimientos en aplicaciones Web.

Un diagrama WebSpec define un conjunto de escenarios que la aplicación Web debe satisfacer. Una interacción (denotado como un rectángulo redondeado) representa un punto donde el usuario puede interactuar con la aplicación usando objetos de interfaz (widgets). Cada diagrama se define en base a dos elementos principales:

- Las *interacciones* tienen un nombre (único por diagrama) u puede tener widgets tal como Etiquetas, listas, etc.

- La *transición* (tanto una navegación o comportamiento rico) es gráficamente representada con flechas entre *interacciones* mientras que el nombre, precondiciones y acciones disparadoras son presentadas como etiquetas sobre estas. En particular, el nombre aparece con el prefijo '#', la precondición entre {} y la acción en las líneas siguientes.

Los escenarios especificados por un diagrama WebSpec son obtenidos atravesando el diagrama usando un algoritmo de búsqueda en profundidad (depth-first). Este algoritmo comienza desde un conjunto especial de *interacciones* denominadas de "inicio" (*interacciones* bordeadas con líneas punteadas) y sigue con las transiciones (*transiciones* o *comportamiento rico*) del grafo (diagrama).

Un ejemplo de los conceptos de WebSpec se presentan en la Ilustración 15, donde se especifica el escenario: "Como cliente, me gustaría buscar productos por nombre y ver todos sus detalles" en

una aplicación de comercio electrónico. *Home* representa el punto de inicio de la especificación y contiene dos widgets: campo de texto *searchField* y botón *search.*

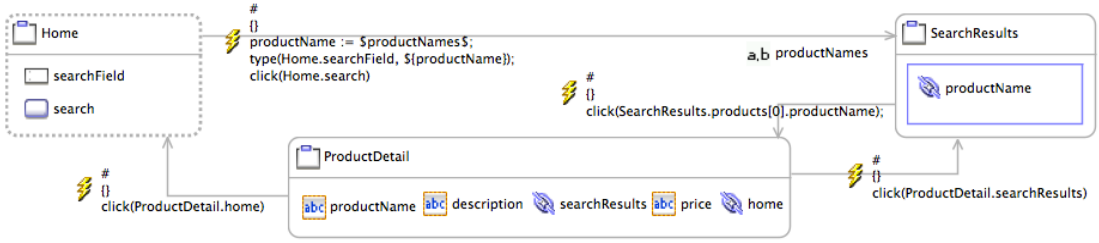


Ilustración 15 Diagrama WebSpec para el requerimiento "Buscar película por nombre"

2.2 Modularización en Software

2.2.1 Cohesión - Definición

La cohesión es una métrica de la POO que permite medir que tan bien los elementos (variables y métodos) de una clase (de la programación orientada a objetos) están relacionados entre ellos. Una clase no cohesiva realiza dos o más funciones no relacionadas. Un diseño con una baja cohesión debería ser reformulado para que los componentes sean reestructurados en dos o más componentes pequeños.

Esta métrica es usualmente calculada en las clases de modelos OO bajo el nombre de Falta de Cohesión de Métodos (LCOM, *Lack of COhesion of Methods*). Existen diferentes variantes para medir la falta de cohesión[32] LCOM1, LCOM2, LCOM3 y LCOM4; en esta tesis se utilizará la variante LCOM4.

En la Ilustración 16 podemos apreciar la relación entre los elementos de una Clase llamada *Chofer*. Estos elementos (variables y métodos) fueron agrupados (con cajas de líneas punteadas) en tres conjuntos según cómo están relacionados. Como se puede ver, en el primer conjunto (con un "1" en la cabecera del conjunto) se dispone de una variable *Auto* y tres mensajes *parar()*, *manejar*() e *irHasta*(); donde los dos primeros manipulan la variable *Auto* y el último interactúa con el método *Manejar*(). Luego, de forma similar al caso anterior, el segundo grupo (con cabecera "2") solo tiene una variable de instancia llamada *cerebro* y un método que la manipula: *dormir*(). Finalmente, en el último grupo (con cabecera "3") se puede apreciar que existe un método aislado llamado *tomarTe*(). Este agrupamiento indica que el valor de la métrica LCOM4 es de 3 debido a que los tres aspectos de la clase *Chofer* detectados no están relacionados de ninguna forma. Nótese que mientras más alto es este valor más baja es la cohesión. Por cuestiones de alcance, en esta tesis no se desarrollará la métrica LCOM4; ver [32] para más información.

Se recomienda el rediseño de las clases con LCOM4 mayor a 1 ya que la misma posee responsabilidades o comportamientos disjuntos. Es decir, proveen funcionalidad que probablemente corresponda a dos clases diferentes.

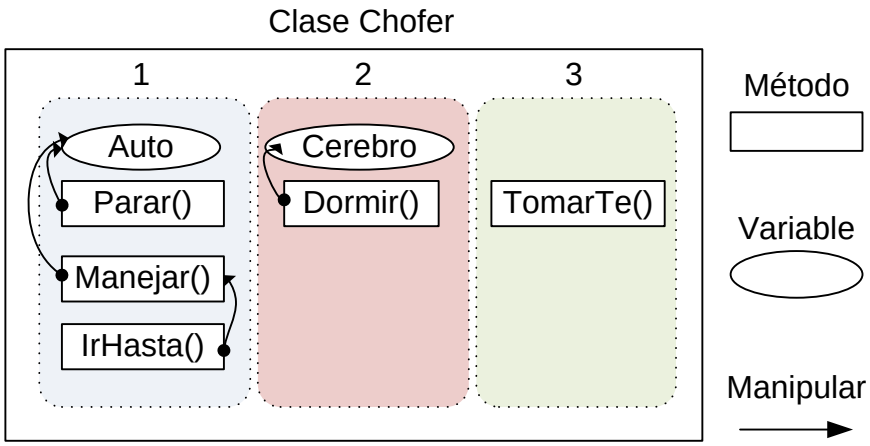


Ilustración 16 Ejemplos de cohesión en clase Chofer

2.2.2 Acople - Definición

El acople es la relación que existe entre dos clases en donde los cambios sobre una impactan sobre la otra. El nivel de acoplamiento [33] determina que tan fuerte es el vínculo entre clases de un sistema. En términos prácticos, cuando al modificar una clase debemos modificar otra , esto significa que existe acople. Un alto acople perjudica la reutilización de las clases de forma aislada ya que una depende de la otra.

Por otro lado, cuando el acople es bajo, se incrementa la posibilidad de poder reusar las clases por separado. Los patrones de diseño OO promueven sistemas con bajo acople. Por ejemplo, el patrón diseño *Observer*[26] es bajo en acople debido a que el *sujeto* (rol *subject* del patrón) sólo conoce que el *observador* (rol *observer* del patrón) implementa una interfaz específica. En este caso nunca se conoce la interfaz concreta del *observador* sino que se sabe que tiene un contrato específico. Se pueden agregar *observadores*, eliminar o reemplazar *observadores* en cualquier momento sin necesidad de modificar el objeto *sujeto*. Cambios tanto al *sujeto* como al *observador* no van a implicar cambios al otro objeto.

2.2.3 Inversión de control - Definición

La técnica de Inversión de Control[34] (IoC, *Inversion of Control*) permite eliminar la creación de componentes y gestión de dependencias de componentes. La inversión de control usa exhaustivamente diseños basados en interfaces ya que el framework que implementa la inversión

de control configura objetos a partir de su contrato y no de su implementación. De esta forma se disminuye el acople de los componentes de la aplicación como se discutió anteriormente.

Supongamos el caso donde la clase *CarritoDeCompras* en un sitio de comercio electrónico colabora con una clase *MedioDePago* para alcanzar su objetivo. Del lado izquierdo de la Ilustración 17, la clase *CarritoDeCompras* obtiene un objeto *TarjetaMPImpl* (que implementa la interfaz *MedioDePago)* mediante la invocación del constructor de este último o utilizando algún objeto que implementa el patrón Factory[26]; en ambos casos gestionando las dependencias dentro de la clase. Por otro lado, utilizando IoC tal como se puede apreciar del lado derecho de la 17, la instancia TarjetaMPImpl es inyectada por algún proceso externo; usualmente implementado en un *framework*. La inyección de dependencias dinámicamente en tiempo de ejecución es también conocido bajo el concepto de Inyección de Dependencias (DI, *Dependency Injection*).

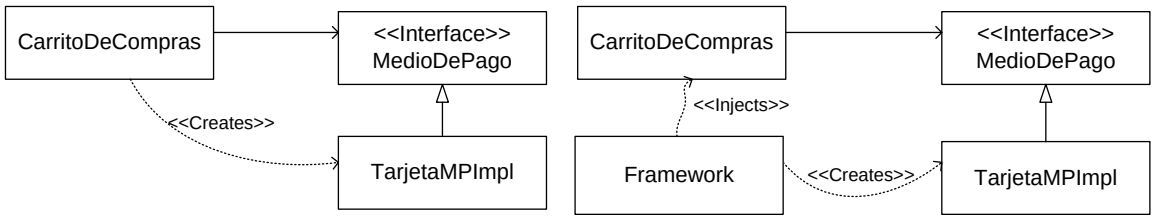


Ilustración 17 Comparación entre gestión de dependencias tradicionalicional e inversión de control

2.2.4 Concerns en Software

Un *concern*[35][36] representa un interés sobre el sistema, agrupando un conjunto de requerimientos o consideraciones específicas que deben ser alcanzadas para satisfacer los objetivos de un sistema. Por ejemplo, un sistema bancario puede estar compuesto por los siguientes *concerns*: administración de clientes y cuentas, computo de intereses, transacciones interbancarias, transacciones ATM, persistencia, etc.. Estos *concerns* son denominados *core concern* porque representan la funcionalidad principal del sistema.

El principio de separación de propósitos (*concerns*) propone encapsular características en entidades separadas para poder localizar cambios de estos y trabajarlos de forma aislada reduciendo su complejidad de diseño e implementación. El primer paso es descomponer el conjunto de requerimientos separándolos en *concern*.

Identificar y separar los *concerns* en un sistema es un ejercicio importante en el desarrollo de un sistema de software, sin importar la metodología utilizada.

2.2.4.1 Concern volátil

Un concern volátil es un conjunto coherente de requerimientos volátiles sobre un interés del sistema. Este tipo de concern posee las mismas características que las funcionalidades volátiles, ya que está definido a partir de estas funcionalidades, donde surge de forma inesperada, permanece en el sistema por un tiempo indeterminado y puede retirarse tanto por un evento de tiempo o de negocio específico.

2.2.4.2 Crosscutting concerns

Los *concerns* que entrecruzan otros *concerns* son denominados "*crosscutting concerns*". Éstos son responsables de producir representaciones entrelazadas; las cuales son difíciles de entender y mantener. Ejemplos de este *crosscutting concern* son los de remoting (lógica específica para hacer invocaciones remotas a objetos) y sincronización (lógica que permite ordenar el acceso a secciones críticas) de código que no puede ser encapsulado en una clase y en general se distribuye en varías clases.

La metodología de Programación Orientada a Objetos (POO) fue desarrollada como respuesta a la necesidad de modularizar los *concern*s de un sistema de software. La realidad es que aunque POO es bueno a la hora de modularizar "core *concerns*", ésta falla cuando se tratan de "*crosscutting concerns*". Los diseños OOP crean acople entre los concerns core y *crosscutting* que es indeseable, ya que la incorporación de nuevas funcionalidades o incluso ciertas modificaciones a la funcionalidad crosscutting requieren modificar módulos relevantes correspondientes al concern core. En AOSD [36] y AspectJ [35] se pueden encontrar descripciones claras e ilustrativas de la falencia de POO para aplicar separación de *concerns* diseños teniendo en cuenta más de un único criterio de separación.

Existen dos tipos de *crosscutting concern* dependiendo del código que estos producen: código *tangling* y código *scattered.*

Tangling

Cuando un componente OO implementa varios *concern*s simultáneamente con lógica de éstos entrelazada, nos encontramos con código *Tangling* (enredado o entrelazado). Un desarrollador usualmente considera *concerns* como: lógica de negocio, performance, sincronización, logging (o registro en bitácora), seguridad, etc. que son implementados en el mismo módulo.

En el Tabla 4 el pseudo código de ejemplo se puede apreciar cómo diferentes *crosscutting concerns* producen código *tangling* en una funcionalidad del sistema. *Concerns* de Traza (logging), Concurrencia, Seguridad y Transaccionalidad producen un código mezclado que entorpece la tarea del programador ya que el mismo debe prestar atención a diferentes aspectos en lugar de hacer foco en mantener la lógica de negocio.

```
public class ClaseEspecifica extends ClaseAbstracta {
        - atributos de la clase específica
        - variable necesaria para de traza en archivos log      ← Concern Traza
        - Semáforo de control de concurrencia                   ← Concern Concurrencia
public void operacion (<parámetros>) {
        - Código de validación de autorización                  ← Concern Seguridad
        - Bloqueo de objeto para sincronizar acceso a           ← Concern Concurrencia
        sección crítica
        - Comienzo de transacción                               ← Concern Transaccionalidad
        - Traza de inicio de operación                          ← Concern Traza
        - Realizar la operación de negocio esperada
        - Traza de fin de operación                             ← Concern Traza
        - Commit o rollback de transacción                      ← Concern Transaccionalidad
        - Desbloqueo de objeto para sincronización              ← Concern Concurrencia
}
...
}
```

Tabla 1 Pseudo código de un componente típico con código tangling

Scattered

Código *scattered* (esparcido o disperso) es representa escenarios cuando una idea es implementada en múltiples módulos. Por ejemplo, en un sistema que usa una Base de datos, el concern de performance puede afectar todos los módulos que acceden a la BD.

Es importante destacar que los *crosscutting* no son excluyentes con lo cual un *crosscutting concern* puede producir tanto código *tangling* como *scattered*. En la Ilustración 18 se puede apreciar como el mismo código utilizado para crear transacciones se ve replicado en varios artefactos de software en el sistema.

La Ilustración 18[3] muestra un típico ejemplo en el cual existe código de acceso a una base de datos repetido en diferentes lugares de una aplicación.

[3] Imagen tomada del libro Aspect-Oriented Analysis and Design: The Theme Approach [37]

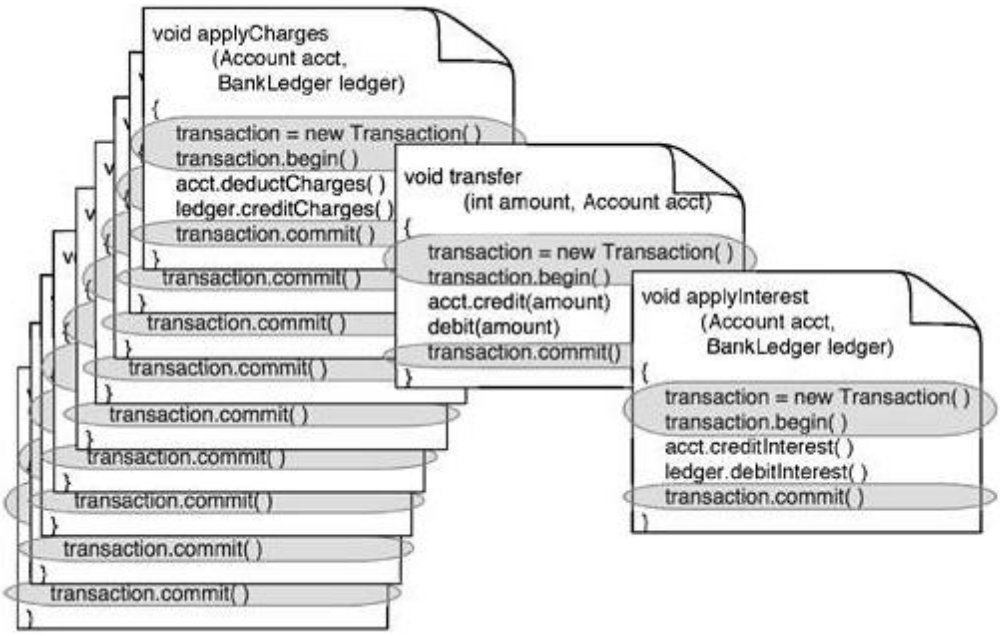


Ilustración 18 Código Scattered [37]

Un ejemplo de *crosscutting concern* es el sistema de logging[4] utilizado en Tomcat[5], donde dicha funcionalidad se encuentra esparcida a través de todo el sistema y no localizada en una única entidad del lenguaje. Se puede comprobar, por tanto, cómo en los sistemas OO de una cierta complejidad se encuentran a menudo con problemas de modularidad.

2.2.4.2.1 Problemas de modularización

Los códigos *tangling* y *scattering* juntos impactan en el diseño y desarrollo de software en las siguientes formas:

- ***Mala trazabilidad***: Implementaciones simultáneas de varios *concerns* incrementa el nivel de complejidad de las tareas de implementación de un *concern*; esto causa dificultad en el seguimiento de requerimientos desde su relevamiento a su implementación y viceversa.

- ***Poca productividad***: La implementación simultánea de múltiples *concerns* mueve el foco del *concern* core (el de la lógica de negocio) a los *concerns* secundarios. La falta de foco

[4] Logging se refiere al proceso de bitácora, en el cual se asientan los pasos realizados.

[5] Apache Tomcat es un servidor de aplicaciones. Es un contenedor de Servlets. Los Servlets es generar páginas Web de forma dinámica a partir de los parámetros de la petición que envíe el navegador Web. (http://tomcat.apache.org)

empeora la productividad porque los desarrolladores tiene que dejar de lado el objetivo primario para solucionar los *crosscutting concern*.

- ***Bajo reúso de código***: Si un módulo está implementando múltiples *concerns*, otros sistemas que requieren funcionalidad similar no van a ser capaces de usar fácilmente el módulo.

- ***Baja calidad***: el código *tangling* dificulta el examen de código y en consecuencia la localización de errores, y la corrección del código. Por ejemplo, revisar el código de un módulo que implementa varios *concerns* va a requerir la participación de un experto en cada uno de los *concerns*.

- ***Dificultad en la evolución***: Una perspectiva incompleta y limitada resulta en un diseño que solo alcanza los *concerns* actuales. Cuando un nuevo requerimiento llega, éste requiere una reimplementación. Esto se debe a que la implementación no esta modularizada e implica la modificación de varios módulos. Modificar cada módulo para tales cambios puede llevar a inconsistencias; lo cual requiere mucho esfuerzo en testeo para asegurar que los cambios no introducen errores.

2.2.5 Desarrollo Software Orientado a Aspectos

La comunidad de Desarrollo Software Orientado a Aspectos (AOSD, *Aspect Oriented Software Development*) ha estado estudiando durante más de una década cómo incrementar y mejorar la expresividad de los paradigmas Orientados a Objetos. Las principales propuestas de AOSD se basan en la idea de desarrollar un software de mayor calidad mediante la separación de *concerns*, especificando cada uno de ellos de forma separada y las relaciones existentes entre los mismos, para posteriormente, utilizando un mecanismo adecuado, componerlos formando un sistema completamente coherente.

Actualmente, POO constituye el paradigma dominante en el desarrollo de software. Se pueden encontrar distintas metodologías, herramientas de análisis y diseño, y lenguajes de programación OO. La programación OO ha hecho posible el desarrollo de sistemas y aplicaciones de una cierta complejidad sin dejar de lado el mantenimiento de un código comprensible y estructurado. Sin embargo, la OO aún presenta algunas limitaciones. Los investigadores OO ven como ciertos aspectos de los sistemas que implementan no pueden ser separados de una manera clara en objetos; aunque se pueda encontrar un diseño orientado a objetos, esté sufre de sobre-ingeniería. El código que implementa esos aspectos se encuentra esparcido y/o anidado en diferentes elementos o estructuras. Para los desarrolladores, resulta complicado centrarse en esos *concerns*, de manera que el mantenimiento y la adaptabilidad del software de cierto tamaño se convierte en un proceso de una complejidad importante.

Básicamente, AOSD reduce las principales limitaciones de la orientación a objetos con respecto a este problema de modularidad. Desde el punto de vista del desarrollador, es evidente que algunas de las características más importantes del software son la facilidad de comprensión de su código y la flexibilidad del mismo para adaptarse a las extensiones y cambios que puedan producirse en los requisitos para los que fue diseñado. Para conseguir que el software presente estas características, es importante conseguir mantener una buena modularidad del mismo.

2.2.5.1 Programación Orientada a Aspectos

Un aspecto es una unidad modular diseñada para implementar un *concern*. Una definición de aspectos puede contener algo de código y las instrucciones de dónde, cuándo, y cómo invocarlos. Dependiendo del lenguaje de aspectos, los aspectos pueden ser construidos jerárquicamente, y el lenguaje puede proveer mecanismos separados para definir un aspecto y especificar su interacción con el sistema subyacente.

Programación Orientada a Aspectos (AOP) complementa las metodologías existentes de programación como la programación orientada a objetos (POO) y la programación procedural, aumentando la capacidad de expresión permitiendo modularizar los *crosscutting concern*s. Los *crosscutting concern*s son modularizados por su rol definido en el sistema, implementando cada rol en su propio módulo y reduciendo el acoplamiento entre módulos.

Usualmente, con AOP, los core *concerns* son implementados usando la metodología base elegida, por ejemplo POO. Los Aspectos del sistema encapsulan los *crosscutting concerns*; estos estipulan como los módulos del sistema son ensamblados para conformar el sistema final.

En [35] se puede encontrar una interesante reflexión sobre la utilización de patrones de diseño OO para implementar *crosscutting concern* donde se destacan ventajas y desventajas de su utilización.

2.2.5.2 Metodología AOP - Descripción

De muchas formas, desarrollar un sistema usando AOP es similar a desarrollar un sistema usando otras metodologías: identificar los *concerns*, implementarlos, y formar el sistema final combinando éstos. Para esto se definen tres pasos:

- **Descomposición de Aspectos**: En este paso, se descomponen los requerimientos para identificar los *crosscutting* y los *core concerns*.

- **Implementación de *Concern*s**: Se implementa cada concern de manera independiente. De esta forma, los desarrolladores pueden implementar, por ejemplo, las unidades de lógica de negocio, de logueo, de persistencia y de autorización.

- **Recomposición de Aspectos**: Finalmente, se especifican las reglas de recomposición para los módulos, o aspectos. Este proceso de recomposición, conocido como *weaving*, usa esta información para componer el sistema final.

El cambio fundamental que AOP provee es la preservación de la independencia de los *concerns* cuando son implementados. La implementación puede ser fácilmente ensamblado al correspondiente concern, resultando en un sistema que es más simple, fácil implementar y más adaptable a los cambios.

2.2.5.3 Beneficios de AOP

Las críticas de AOP hablan de la dificultad de entenderlo. Y en efecto AOP toma un poco de tiempo, paciencia y práctica para aprender. Sin embargo, la razón principal, detrás de la dificultad, es la novedad de la metodología. AOP demanda pensar el diseño e implementación del sistema de una forma nueva.

Algunos beneficios son:

- ***Responsabilidades claras de los módulos***: AOP permite a un módulo responder sólo a sus requerimientos y no tener en cuenta requerimientos de otros *concerns* (*crosscutting concern*). Por ejemplo, un módulo que accede a la Base de Datos, ya no es responsable de gestionar una conexión, transacción, etc..
- ***Alta modularización***: AOP provee un mecanismo para obtener *concerns* con un mínimo acoplamiento entre ellos. Las implementaciones modularizadas dan como resultado sistemas fáciles de entender y mantener.
- ***Evolución simplificada***: AOP modulariza los aspectos y permite a los core *concerns* ignorar estos aspectos. Agregar una funcionalidad es ahora usando AOP una forma de incluir un nuevo aspecto y no requiere cambios en el modulo core. Además cuando agregamos un nuevo modulo core al sistema, los aspectos existentes lo cruzan, ayudando a una evolución coherente.
- ***Retraso de decisiones de diseño***: Los arquitectos pueden posponer la toma de decisiones de diseño para futuros requerimientos porque es posible implementarlos como aspectos separados. Ellos pueden enfocarse en los requerimientos base actuales del sistema. Los nuevos requerimientos de una naturaleza *crosscutting* pueden ser manejados con la creación de un nuevo aspecto.
- ***Mayor reúso de código***: La llave para un gran reúso de código es un bajo nivel de acople en la implementación. Porque AOP implementa cada aspecto como un módulo separado, cada módulo es menos acoplado que su implementación equivalente convencional.
- ***Rápida salida a producción***: Retrasar las decisiones de diseño permiten un ciclo de diseño más rápido. Una clara separación de responsabilidades permite una mejor asignación de desarrolladores especializados a módulos. Mayor reúso de código reduce el tiempo de desarrollo. Una fácil evolución permite una rápida respuesta a nuevos requerimientos. Todo esto conduce a un sistema que es fácil de desarrollar y distribuir.
- ***Reducción de costo de implementación***: AOP reduce los costos de implementar las características de *crosscutting*, ya que evita la modificación de los módulos cada uno por

separado. Además permite que cada desarrollador se enfoque en un concern de un módulo particular y haga uso de su especialidad, el costo de los requerimientos core es también reducido.

2.2.5.4 Aspectos tempranos

Las técnicas AOSD proveen medios semánticos para la identificación, modularización, representación y composición de *crosscutting concerns* tal como la seguridad, movilidad y restricciones de la aplicación. Estas propiedades tienen un efecto amplio sobre otros requerimientos o componentes de arquitectura.

AOSD ha proveído una perspectiva diferente complementaria del principio de separación de *concerns*. Como resultado, un número creciente de proyectos de software explícitamente consideran la separación de aspectos en la aplicación para alcanzar una mejor modularidad. Un análisis de los sistemas muestra que muy seguido los mismos aspectos son implementados una y otra vez. En el diseño de sistemas distribuidos, por ejemplo, *concerns* como sincronización, recuperación, seguridad, logging, y monitoreo son implementados; al parecer muchos de estos *concerns* son *crosscutting* y deberían ser implementados como aspectos. A pesar del hecho que muchos aspectos están presentes en varias aplicaciones ninguna metodología semántica parece haber sido propuesta para capturar y reutilizar estos aspectos. Los mismos aspectos son implementados para cada caso en particular o en el mejor de los casos son reutilizados oportunamente adaptando las especificaciones existentes de los aspectos.

La reutilización oportuna puede funcionar en un modo limitado para programadores individuales o pequeños grupos. Sin embargo, esta metodología no escala fácilmente en aplicaciones Web grandes y por lo tanto se requiere alguna tentativa semántica para la reutilización de software. La motivación generalmente reconocida para la reutilización sistemática son (válidas también para aspectos): disminuir el tiempo del ciclo de desarrollo, incrementar la calidad, y disminuir el costo de desarrollo.

Desafortunadamente, las metodologías AOSD convencionales se han enfocado principalmente en identificar aspectos a nivel de desarrollo (programación) y no ha prestado atención en el impacto de los *crosscutting concerns* en las primeras fases del desarrollo de software.

La iniciativa de Aspectos Tempranos (*Early Aspects)*[38] se concentra en el manejo de las propiedades *crosscutting* en las primeras etapas de la ingeniería de requerimientos y del diseño de la arquitectura del software.

Si los *Early Aspects* no son efectivamente modularizados, no es posible deducir sus efectos en el sistema. Además, la ausencia de modularización de tales propiedades puede resultar en una larga onda de efectos sobre otros requerimientos o componentes de arquitectura en su evolución.

Ha habido un trabajo significativo en la separación de *concerns* en las comunidades de ingeniería de requerimientos y diseño de arquitectura. Por ejemplo, *viewpoints*, *use case*, *goals* y modelos de

análisis de arquitectura. Sin embargo, estas técnicas no es concentran explícitamente en *crosscutting concerns*. El trabajo en Early Aspects complementa estas técnicas completándolas con medios semánticos para manejar tales *concerns*.

2.3 Herramientas de diseño

2.3.1 MATA – Descripción y ejemplo

MATA (*Modelling Aspects using a Transformation Approach*) soporta un amplio conjunto de tipos de composición en comparación a otras metodologías (por ej. casos de usos aspectuales de Jacobson [39]). Por ejemplo, un diagrama de secuencia de aspectos puede ser compuesto con un diagrama de secuencia base utilizando los conocidos fragmentos (constructores) *parallel*, *alternative*, y *loop* como parte de la regla de composición. Muchas de otras técnicas existentes se limitan a las estrategias *advice* de AspectJ[35] (*before*, *after* y *around*).

El mecanismo de composición de MATA está basado en transformaciones de grafos. Una transformación de grafo es una regla r: L→R donde L corresponde al grafo lado izquierdo (LHS) al grafo del lado derecho R (LHS). En MATA, la composición de un modelo base UML (Mb) con un modelo de un aspecto (Ma) que cruza (es *crosscutting*) al modelo base, es especificado con una regla de grafo, r: LHS → RHS:

- Un patrón es definido del lado izquierdo (LHS), capturando el conjunto de puntos en el modelo base donde los nuevos elementos deben ser agregados;
- El lado derecho (RHS) define aquellos elementos nuevos elementos y especifica como éstos deben ser agregados al modelo base.

MATA soporta composición de diferentes diagramas UML (diagramas clases, secuencia, actividad y estado). Las reglas de grafo son especificadas utilizando la sintaxis del diagrama UML concreto siendo este combinado con alguna extensión de MATA para lograr mayor expresividad en la composición. Los siguientes estereotipos UML son utilizados en reglas MATA presentes en diagramas:

- «create»: aplicado a cualquier elemento del modelo, especificando la creación de un elemento;
- «delete»: aplicado a cualquier elemento del modelo, especificando la eliminación de un elemento;
- «context»: utilizado para indicar elementos del diagrama que definen un contexto requerido es decir que su presencia es obligatoria para poder aplicar la regla.

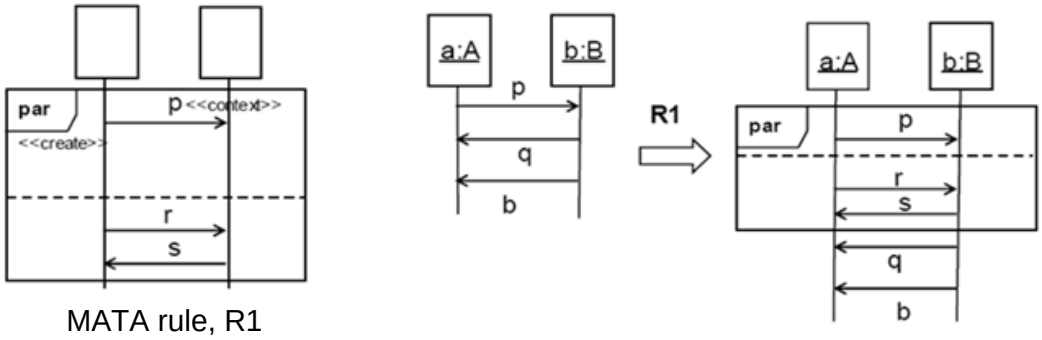


Ilustración 19 Ejemplo de regla MATA

La Ilustración 19 muestra un ejemplo de una regla MATA que especifica la incorporación de dos invocaciones de mensajes posteriormente a la invocación del mensaje *p* entre las instancias *a* y *b.* La regla R1 especifica que el comportamiento aspectual consiste en una interacción entre dos objetos que deben ser instanciados con elementos del modelo base; es decir, los objetos serán completados con aquellos objetos que en el diagrama base cumplan con las restricciones establecidas por los estereotipos. La regla dice que el fragmento *par* (que indica paralelismo) y los mensajes *r* y *s* deben ser creados; ellos definen un comportamiento aspectual que debe ser insertado en el modelo base. Sin embargo, dado que *p* es definido como "«context»", éste debe solaparse con un mensaje con el mismo nombre en el modelo base. El modelo compuesto resultante cuando se aplica la regla R1 se muestra en el lado izquierdo de la figura. Dado de que *q* y *b* no son parte de la regla, ellos aparecen luego del fragmento *par*.

2.3.2 Especificación de Patrón - Descripción y ejemplo

Una Especificación de Patrón, (PSs, *Pattern Specifications*) [40] son una forma de formalizar el reúso de modelos. La notación de los PS está basada en UML. Un PS describen un patrón de estructura o comportamiento definido sobre los roles que los participantes del patrón juegan. Los nombres del los roles van precedidos por una barra vertical ("|"). Una PS puede ser instanciada mediante la asignación de elementos de modelados concretos para representar esos roles. Un rol es una especialización de una metaclase UML restringida por las propiedades adicionales que cualquier elemento que satisface el role debe poseer. En la Ilustración 20 se puede ver la representación gráfica basada en una extensión UML. En esta figura se pude apreciar las tres partes principales definición, especificación estructura y especificación de comportamiento. En la sección estructural se definen los elementos requeridos para que el rol "concuerde" como también los nuevos elementos introducidos por el rol. La sección de comportamiento funciona de forma análoga pero a partir de métodos.

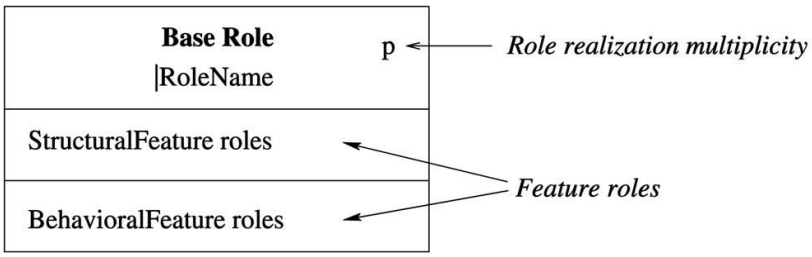


Ilustración 20 Estructura de un rol

En consecuencia, un rol especifica un subconjunto de instancias de la metaclase UML. Un modelo conforma a un PS si sus elementos que interpretan los roles del PS satisfacen las propiedades de los roles. En la Ilustración 21 se puede apreciar a la derecha un rol y a su izquierda una clase que lo conforma. En [40] se puede encontrar una explicación sobre PS detallada.

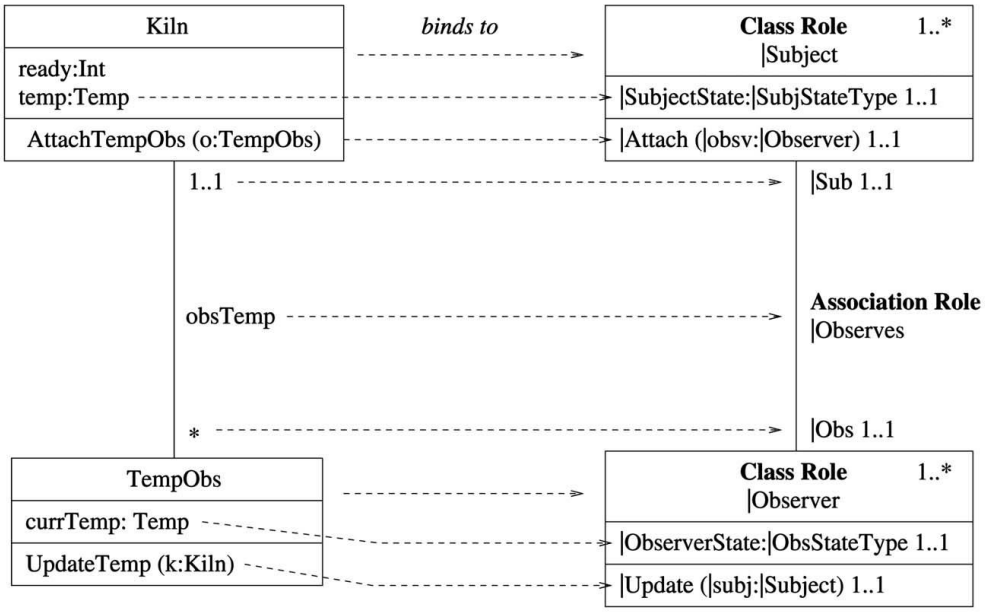


Ilustración 21 Un PS y una clase que lo conforma

Capítulo 3 Trabajos relacionados

Mientras que la evolución del software ha sido estudiada por años, la volatilidad[5] es un problema relativamente nuevo en la literatura. El impacto de requerimientos volátiles en el costo de un producto de software ha sido rigurosamente estudiado [41]. Actualmente es aceptado que el Software puede variar de formas imprevistas y que el impacto de estos cambios en los costos generales del Software es afectado por la complejidad creciente del ciclo de vida de la aplicación. Teniendo en cuenta los ejemplos presentados en la Sección 1.1, la funcionalidad del *concern Volver al Colegio* demanda la activación y desactivación de varios componentes de software y la ejecución de un número de actividades de desarrollo de software (análisis de requerimientos, diseño, implementación y testing) para introducir los cambios y luego removerlos cuando deja de ser necesario. En la mayoría de los casos, se requiere nuevamente le ejecución de tareas de codificación, testing e instalación de la aplicación para validar que no se hayan introducido errores por el apartamiento de la funcionalidad volátil.

3.1 Metodologías existentes

La primer metodología para lidiar con estos cambios de forma modular fue presentada en [5]; en este trabajo los autores proponen encapsular los *concern* volátiles durante la etapa de análisis de requerimientos, modelarlos como aspectos (tempranos) y seguir técnicas existentes de Desarrollo de Software Orientado a Aspectos (AOSD[6]) en el ciclo de vida general de la aplicación. Aunque la metodología es general, ésta carece de herramientas de especificación de aplicaciones Web Interactivas y en consecuencia ésta no cubre características específicas tal como el enriquecimiento de una página, el cambio temporal de una interfaz, o de navegación.

El trabajo propuesto en este documento también ha sido inspirado en las llamadas metodologías de separación de *concern* simétricas, tal como Theme/Doc[37]. En ésta técnica tanto los requerimientos core y volátiles son modelados utilizando los mismos conceptos, mientras que en una metodología asimétrica los *crosscutting concerns* son modelados utilizando una primitiva diferente: un aspecto.

Tal como será mostrado más adelante, la metodología clave que se propone en esta tesis para lidiar con funcionalidad volátil es conseguir una conexión y desconexión de nuevos requerimientos donde las mismas se incorporen fácilmente a un sistema y con la misma facilidad

[6] Se utilizará el acrónimo en inglés AOSD debido a su popularidad: Aspect Oriented Software Development (AOSD)

sean retirados. En el contexto de aplicaciones Web he encontrado tres metodologías para resolver este problema:

- Una metodología basada en componentes es presentado en [42] . El Web Composition Process Model es una metodología sistemática para incorporar evolución de requerimientos como un concepto principal en el ciclo de vida de una aplicación Web alentando el reúso mediante el llamado "Web Application Evolution Bus". A partir de contratos, se simplifica la adición de nuevas funcionalidades en componentes descriptos utilizando un lenguaje llamado Web Composition Mark-up Language. A diferencia de la metodología presentada en este trabajo, éste no permite lidiar con mecanismos de conexión y desconexión de nuevos componentes (volátiles); por otro lado, desde el punto de vista del nivel de abstracción, Web Composition Process Model no considera los problemas que surgen en las diferentes capas de una aplicación Web (modelo conceptual, navegacional y de interfaz) pero ataca principalmente aspectos de implementación.

- Otra tendencia destacable para lidiar con la evolución de forma transparente es la utilización de tecnologías Orientadas a Aspectos tal como se mencionó en secciones anteriores. En áreas especificas de aplicaciones Web, una interesante metodología puede ser leída en [43]. En este trabajo los autores proponen tratar de resolver evolución con Patrones de Diseño Orientado a Aspectos (Aspect-Oriented Design Patterns- AODP). La metodología es sólida y poderosa ya que los autores identifican un conjunto de tipos de cambios posibles en una aplicación Web y asocian un Patrón de Aspecto para resolver cada situación específica. Sin embargo, éste está dirigido a la implementación de funcionalidades más que a su diseño y en consecuencia éste pierde parte de su poder; sin embargo, como es discutido durante este trabajo, las ideas subyacentes detrás de la orientación a aspectos son conceptos claves para ser utilizados con la funcionalidades volátiles.

- Otro trabajo relacionado en la que se encontró problemas e ideas similares a las presentadas aquí es adaptación en aplicaciones Web. AMACONT[44] es un framework para el modelado Web con facilidades ortogonales que permite incorporar funcionalidad sin dependencias entre la adaptación y el componente adaptado. Este framework provee conceptos y herramientas para llevar a cabo a adaptación en aplicaciones Web implementando conceptos AOP. A partir de la utilización de adaptaciones orientadas a aspectos y basadas en semánticas para diferentes granularidades de adaptación, éste permite especificar cambios en modelos basados en componentes. El trabajo provee a un framework útil pero carece de medios para especificar aspectos de presentación así como también describir el ciclo de vida de las adaptaciones, o aspectos.

En [45] los autores usan una metodología orientada a aspectos para incorporar funcionalidad de navegación adaptativas en aplicaciones Web; la metodología, definida sobre el método UWE, soporta la mayoría de los bien conocidos patrones de navegación adaptativa de forma modular y

transparente. Utilizando aspectos, garantiza la activación y desactivación transparente aunque limitado a la capa de navegación. También, en [46] los autores proponen una metodología semántica orientada a aspectos para adaptar la cual combina la orientación a aspectos con el poder de los conceptos de la Web semántica en el contexto del método Hera [17]. En el contexto de Aplicaciones de Internet Ricas (RIAs), los autores de [47] proponen OOH4RIA, una extensión del método OO-H que permite definir adaptaciones de personalización para presentaciones RIA.

Siguiendo la línea de UWE, en [48] los autores presentan XML Publication Framework basado en la metodología UWE. El motor de generación de aplicaciones Web puede ser extendido con los conceptos presentados en esta tesis ya que el mismo transforma modelos en artefactos de Software; solo sería necesario incorporar filtros de transformación en la línea de procesamiento de Cocoon[49].

3.2 Desarrollo dirigido por modelos

Feature Oriented Programming[7] (FOP)[50] provee medios para sintetizar software por medio de la composición de funcionalidades llamadas en inglés *Feature*. Una *Feature* incrementa las funcionalidades de un sistema por medio de un pequeño refinamiento que introduce clases, recursos, dependencias etc. y tiene como objetivo ser reutilizada en varias aplicaciones. FOP y la metodología presentada en esta tesis comparten los mismos objetivos subyacentes: diseñar funcionalidades de forma modular donde las *features* son tratados como entidades de la aplicación. En [51] se presenta un framework para álgebra de *features* que permite diferentes alternativas de composición de *features*. Operaciones tal como la introducción y modificación de *features* son descriptas formalmente teniendo en cuenta propiedades algebraicas bien conocidas tal como asociatividad, conmutatividad, idempotencia, entre otras. Dado que éste es un framework de *features* general, éste no se instancia en un único dominio especifico, por lo tanto se dejan detalles de implementación sin cubrir. Cuando FOP es combinado con el paradigma Model Driven Development (MDD) se obtiene como resultado un nuevo paradigma llamado Featured Oriented Model Driven Development (FOMDD) en el cual modelos son definidos en lugar de artefactos específicos a la plataforma. Por ejemplo, en [52], Trujillo et al. presentan una metodología FOMDD para la producción en línea de *portlet*[53], donde los modelos de los *portlets* son definidos con transformaciones que introducen funcionalidades. La metodología no toma en cuenta el aspecto de volatilidad de las funcionalidades y por lo tanto hace que la composición de *features* sea permanente.

Utilizando *UML Package Merging* [54], es posible diseñar modelos sin acople y de forma trasparente, que luego de su diseño son combinados produciendo un modelo complejo comprendiendo *concerns* diferentes; los elementos UML tal como paquetes y clases son combinados, o entretejidos. Solo algunas plataformas permiten la traducción directa de éste tipo

[7] Programación orientada a funcionalidades.

conceptos de composición de *concerns* tal como las clases parciales en C#[55] brindando la posibilidad de definir diferentes facetas de una clase en diferentes archivos, *Introduction* del lenguaje AspectJ (aspectos) permitiendo especificar los nuevos elementos de la faceta en un archivo (aspect) por separado [35], y FUJI actuando como compilador FOP combinando diferentes *features*[56]. Si la plataforma subyacente no soporte la composición o tejido de clases, el programador debe tomar decisiones que no van a ser trazables debido a que no existe una relación isomorfa entre la implementación y los modelos que permita esto.

3.3 Tecnologías para separación de concerns

La composición de interfaces ha sido estudiada en [57] utilizando operadores de algebra de árbol con el lenguaje UsiXML de descripción [58]. En el trabajo anterior una herramienta visual llamada *ComposiXML* fue desarrollada para dar soporte a estas ideas. La intención de esa investigación es ayudar al reúso de los componentes de interfaz de usuario. La metodología presentada en esta tesis utiliza especificaciones y transformaciones XML entre otras herramientas conceptuales en lugar de algebra de árbol, enfocándose en integración transparente de interfaces core y volátiles.

Hasta ahora no hay antecedentes de metodologías que permitan la composición de modelos de interfaces de forma transparente sin acople. Mientras tanto, en el campo de la tecnología de XML, el proyecto de AspectXML [59] (aún en etapa de investigación) ha portado algunos conceptos de la orientación a aspectos, tal como *pointcut* y *advice* de AspectJ a la tecnología XML.

En [60] se discute sobre un framework J2EE llamado AspectJ2EE que incorpora aspectos en componentes Enterprise Java Beans (EJB). AspectJ2EE puede ser utilizado para incorporar *concerns* volátiles en aplicaciones J2EE solo en la capa de modelo ya que aspectos en la capa de navegación e interfaz no son mencionados en el trabajo. El weaving del aspecto es realizado en la etapa de instalación (*deploy*) mientras que en esta tesis se propone realizarlo en tiempo de ejecución con el objetivo de lidiar con funcionalidades volátiles que solo afectan dinámicamente algunas instancias de una clase.

En [61] los autores presentan una herramienta llamada AWAC basada en A-OOH (extensión de adaptación de OO-H [62]) para la generación automática de aplicaciones Web adaptativas. Una arquitectura basada en reglas es presentada en [63]; por medio de la utilización de reglas Evento-Condición-Acción (ECA), un motor de reglas procesa los eventos generados desde la Web disparando cambios sobre el modelo de la aplicación para lograr comportamiento adaptativo. A pesar de ser soluciones similares, los enfoques anteriores son adecuados para problemas de adaptatividad, mientras que el presentado en este trabajo se enfoca en un espectro más amplio de patrones de cambios y, por otro lado, contempla la incorporación y eliminación de estos cambios de forma no planificada.

A pesar de que la mejora no intrusiva de los modelos conceptuales y navegacionales ha sido tratada en la literatura, no hay antecedentes de alguna metodología que soporte composición no

intrusiva [36] de modelos de diseños de interfaces para poder implementar separación de *concerns* a nivel de interfaz.

Capítulo 4 Caracterización de funcionalidad volátil

La incorporación de funcionalidades volátiles en aplicaciones Web puede ser pensada como un caso de evolución de software Web y en consecuencia resuelta usando técnicas como *WebComposition Process Model* [64], o con técnicas más generales tal como refactoring[16] y patterns[8][26]. A partir de la motivación de la Sección 1, se destaca la necesidad de una metodología específica para el tratamiento de las funcionalidades volátiles ya que la misma naturaleza volátil de estas funcionalidades es la que distinguen a ellas de otras evoluciones de requerimientos.

Una funcionalidad volátil puede utilizar diferentes *patrones de volatilidad*. Muy frecuentemente, las funcionalidades volátiles siguen un ciclo de vida cíclico: son agregadas a la aplicación, eliminadas luego de cierto tiempo para reaparecer de nuevo más adelante, en intervalos regulares de tiempo. Este es el caso de ofertas especiales correspondientes a eventos tal como: el "*Día de la Madre"* o "*Navidad"* en un sitio Web de comercio electrónico, donde año tras año se repite el ciclo. Otras veces, funcionalidades volátiles no siguen un patrón repetitivo, pero necesitan ser activado oportunamente cuando un evento particular e impredecible ocurre y luego ser desactivado. Este es el caso de una recaudación de fondos luego de un desastre natural tal como un terremoto, tsunami, incendio, etc. Otro de este tipo de funcionalidad volátil con un patrón incierto es aquel que corresponde a las funcionalidades Beta, es decir, funcionalidades que son puestas a prueba y si no son adoptadas son retiradas (evento de desactivación).

En ambos tipos de volatilidad, una técnica que permita conectar y desconectar (activar y desactivar) de forma simple y fluida de funcionalidades deseada es tan útil como recomendado.

En las investigaciones realizadas [65], [66] se apuntó a desarrollar una metodología dirigida por modelos [67] para lidiar con funcionalidades volátiles que haga posible:

(i) tratar a las funcionalidades volátiles de forma similar a las funcionalidades core[9] manteniendo su diseño e implementación separado de la las funcionalidades core (aplicación web que recibirá la funcionalidad volátil),

[8] Se referenciará durante esta tesis a los patrones de diseño de la programación orientada a objetos con su nombre en inglés.

[9] Son aquellas funcionalidades que son inherentes a la aplicación también referidas conocidas como funcionalidades base o núcleo.

(ii) y automatizar su activación/desactivación de acuerdo al patrón de volatilidad definido.

Para alcanzar los objetivos enumerados anteriormente, primero se trabajó en la identificación de las características principales de las funcionalidades volátiles analizando varios ejemplos tomados de sitio Web reales y su solución. De este análisis, encontramos que una funcionalidad volátil puede ser caracterizada como mínimo en tres dimensiones: *Propósito*, *Alcance*, y *Patrón de volatilidad*.

El *propósito (Intent)* de una funcionalidad volátil puede ser identificado respondiendo las siguientes preguntas:

- ¿Cuáles características (navegación, elementos visuales, lógica, etc.) de la aplicación Web es impactada por la funcionalidad volátil?
- Por cada característica, ¿Cuáles tipos de objetos (o clases) de la aplicación tienen que ser agregados o modificados y cómo?

A partir de las dimensiones identificadas para una aplicación Web por la mayoría de los métodos de ingeniería Web [18], una funcionalidad volátil puede esparcir cambios en las diferentes capas, o modelos: contenido, navegación, e interfaz de usuario (o presentación) y comportamiento. Respectivamente, los tipos de objetos de la aplicación pueden ser: tipos de contenido, vistas de contenidos, nodos navegacionales, interfaces de usuario, widgets de interacción, operaciones de usuario, flujos de trabajo de procesos de negocios (también conocidos como workflows), actividades de procesos de negocios, y reglas de negocios. Los nombres adoptados para los aspectos mencionados (también conocido como *concern* o capa) y tipos de objeto (también conocido como conceptos de modelado) pueden variar dependiendo de la metodología de ingeniería Web.

El *Alcance (Extent)* de una funcionalidad volátil identifica el conjunto de objetos de la aplicación (instancias de un tipo identificadas por el *Propósito*) que éste impacta. El *Alcance* de una funcionalidad volátil es de hecho determinado respondiendo las siguientes preguntas:

- ¿Qué objetos de la aplicación tienen que ser incorporados o modificados determinados por el *Propósito*?
- ¿Afecta la funcionalidad volátil a todas las instancias de los objetos de negocios de la aplicación?
- ¿Afecta esta funcionalidad volátil a solo algunas instancias especificas de objetos de negocio? ¿A cuáles?

Por cada uno de los tipos de objetos de la aplicación en el *Propósito*, una forma de determinar las instancias comprometidas tiene que ser especificada.

El *Patrón de volatilidad*, como ya fue mencionado, permite describir el ciclo de vida de las funcionalidades volátiles, esto es, las reglas que gobiernan su activación/desactivación a lo largo del tiempo. De hecho, está dimensión distingue a una funcionalidad volátil de un requerimiento evolutivo. A pesar de que tanto un requerimiento evolutivo como una funcionalidad volátil pueden ser imprevistos en el comienzo del desarrollo de la aplicación, una funcionalidad volátil tiene un *Patrón de volatilidad* bien definido que describe su ciclo de vida. Esta dimensión de la caracterización puede ser bien definida respondiendo las siguientes preguntas:

- ¿Cuándo debe ser activada la funcionalidad volátil? Por ejemplo, luego de que un evento específico haya ocurrido, en una fecha determinada, en un día específico de la semana, a una hora particular del día, cuando ciertas reglas de negocio son satisfecha, etc.
- ¿Cuánto tiempo estará activa? Cierto número de días, horas o minutos.
- ¿Cuándo deberá ser desactivada? Cuando cierto tiempo ha pasado desde su activación, cuando una regla de negocios especifica es satisfecha o cuando se llega a una fecha y hora especifica entre otros.
- ¿Aparecerá nuevamente en el futuro? Si es así, ¿Cuál es el patrón de repetición?

Por cada una de las características y sub características descriptas anteriormente, se identificó un conjunto de posibles, pero no exhaustivo, valores que éste puede asumir. Este análisis puede ser realizado durante la etapa de relevamiento de requerimientos, cuando se detecta que una funcionalidad es volátil; por ejemplo, cuando se diseña un sitio de comercio electrónico se define que existirán ofertas especiales durante ciertas fechas ("*Navidad"*, "*San Valentín*", etc.). En otras situaciones, como fue indicado, una funcionalidad volátil puede ser también completamente inesperada. Por ejemplo, se solicita que nuevas funcionalidades tal como el etiquetado de productos [68] para permitir a los usuarios evaluar estas características de forma temporal y estudiar si la adopción de la funcionalidad es la deseada y ,si no lo fuera, se la retirara. En el último caso, la identificación de todas las características de esta funcionalidad volátil ayuda a hacer su incorporación en el sistema más fácil.

En general, se obtuvo una caracterización de marco de trabajo para funcionalidades volátiles reportada en Tabla 1 [66].

Tabla 1 Características de funcionalidades volátiles con valores de ejemplo.

Característica		**Posible valor**
Propósito	Aspectos de la aplicación afectados / tipos de objetos.	• Contenidos o tipos de contenidos, vista de contenidos, accesos a estructuras de contenidos, etc. • Navegación o nodos de navegación, enlaces (también conocidos como links) de navegación, etc. • Interfaces de usuario (o presentación) o páginas, look and feel de las páginas, widgets de interacción definidos dentro de las páginas, etc. • Comportamiento u operaciones de usuario, flujos de trabajo de procesos de negocios, actividades de procesos de negocios, reglas de negocio, etc.
	Tipo de intervención posible	• Agregar nuevos tipos de objetos de navegación • Modificar tipos de objetos existentes en la aplicación
Alcance	Objetos de la aplicación afectados	• Todas las instancias de los tipos de objetos identificados en el *Propósito* • Sólo alguno de ellos, seleccionados dinámicamente (utilizando una consulta),etc.
	Objetos agregados	• Cuales nuevos objetos de los tipos identificados en el Propósito tienen que ser creados.
Patrón de volatilidad	Cuándo	• En alguna fecha en especial (Por ej. Navidad) • En algún día específico del año/mes/semana • En alguna hora fija del día, o día de la semana, etc. • Cuando ocurre algún evento (Catástrofe) • De acuerdo a alguna regla de negocio. Por ejemplo se desea promover algún disco de música, se desea premiar al centésimo cliente del día, etc.
	Por cuánto tiempo	• Por un periodo de tiempo específico. (Por ej. 15 días) • Hasta cierta fecha. (Por Ej., 1 de marzo). • Dependiendo de una regla de negocio. (Por ej. no más de x CDs vendidos) • En base a una intervención humana, en otras palabra alguna acción manual.
	Patrón de repetición	• Cada lunes, cada primer Domingo de mes, a las 8:00am de cada día.

Es interesante destacar algunas características particulares de las funcionalidades volátiles mostradas en la Tabla 1 que se reflejan el resultado del análisis realizado en sitios Web exitosos tal como Amazon.com[6].

Se destacan los aspectos de *Propósito*, *Alcance* y *Patrón de Volatilidad*. Donde en primer lugar, la volatilidad que siempre surge a nivel de requerimientos, puede afectar cualquier tipo *concern* de aplicación (contenido, navegación, interfaz o comportamiento); en algunos casos, funcionalidades volátiles pueden extenderse en diferentes *concern*, es decir, un nuevo contenido o link implica modificar la capa de navegación y la de interfaz. En segundo lugar, las incorporaciones pueden ser muy irregulares, esto es, sólo una o algunas instancias específicas de un tipo de objeto de la aplicación (tipo de contenido, vista de contenido, nodos navegacionales, caminos navegacionales, interface de usuarios, widget de interacción, etc.) pueden estar afectadas. Finalmente, inserciones volátiles pueden ser *crosscutting* con las funcionalidades core: en el Capítulo 9 se describe una característica volátil que ofrece a los clientes un descuento de envío a domicilio cuando se realiza la compra de un lector de libros electrónicos (por ej. Kindle de Amazon) con un producto escolar; esta característica volátil (temporariamente) modifica las reglas de negocio básicas en las tarifas de envío para un producto específico, en consecuencia modificando de forma trasversal el modelo de negocio principal de la aplicación de comercio electrónico. Las características de esta funcionalidad hacen a éste un claro caso de funcionalidad volátil ofrecida por el sitio de comercio electrónico Web Amazon.com. Su *Propósito* se extiende sobre la navegación, presentación y comportamiento de la aplicación y comprende la creación de nuevos contenidos, links de navegación y páginas así como también la modificación de otros objetos existentes en la aplicación Web.

El marco de trabajo presentado en ésta sección, adicionalmente para permitir la caracterización de funcionalidad volátil, representa la base del lenguaje especifico de dominio [69] que se presentará en el Capítulo 10 y que se adopta para configurar el automático activación y desactivación de las funcionalidades volátiles.

Capítulo 5 Metodología de desarrollo de funcionalidad volátil

En la mayoría de las metodologías de diseño Web, tal como UWE, WebML, Hera, OOWS[70] o OOHDM (ver [17] para una descripción y ejemplo de cada una de las metodologías), una aplicación Web es diseñada con un proceso iterativo comprometiendo como mínimo modelado conceptual y navegacional. Algunos métodos también incluyen metodologías específicas para el diseño de procesos de negocios [71] así como también técnicas para el relevamiento de requerimientos y especificación de interfaz de usuario. De acuerdo con el estado de arte de las técnicas de ingeniería Web dirigida por modelos [17], la mayoría de estos métodos de diseño producen un modelo independiente de la implementación que pueden ser luego mapeados a diferentes plataformas para su ejecución. En esta tesis se presentará una metodología para el diseño de funcionalidades volátiles que abarca las principales etapas del desarrollo de Software: relevamiento de requerimientos, su validación, y modelado conceptual, navegacional y de interfaz de los requerimientos. La metodología propuesta utiliza OOHDM como método base de diseño pero puede ser adaptada a otros métodos de diseño de aplicaciones Web ya que los conceptos subyacentes son compartidos entre las metodologías de diseño Web. Aunque algunos comentarios sobre requerimientos volátiles son presentados a continuación, una discusión completa sobre este aspecto puede ser encontrada en [5].

Dado que la mayoría de los problemas descriptos hasta ahora aplican a todas las metodologías de desarrollo, primero se describirá la filosofía subyacente de la solución técnica de tal forma que ésta pueda ser reutilizada; luego, se concentrará en los siguientes capítulos en los modelo de diseño de OOHDM y será brevemente discutido como cada parte de la metodología puede ser adaptado a otros métodos de diseño.

La metodología está basada en la idea de que incluso la funcionalidad volátil más simple (por ej. un video que está disponible por un período de tiempo) debe ser considerada una funcionalidad de hecho y, como tal, debe ser diseñada como tal. Al mismo tiempo, su diseño e implementación tiene que ser realizado por separado y tanto como sea posible desacoplar de aquellas funcionalidades core y estables.

En esta tesis, se utilizará OOHDM como metodología base completándola con herramientas conceptuales que permitan la incorporación de requerimientos volátiles. OOHDM ha sido presentado en la Sección 2.1.1, como resumen se puede decir que la metodología requiere el diseño de la aplicación Web en tres capas o niveles: conceptual, navegacional, y de interfaz. El modelado conceptual se realiza utilizando el paradigma de Orientación a Objetos y se utiliza diagramas UML para su representación gráfica. El modelo navegacional define nodos como

vistas de las clases del modelo conceptual y *links* que establecen la forma en que la aplicación será navegada; este diagrama utiliza una notación gráfica de OOHDM. Y por último el modelo de interfaz establece de qué forma será percibida la información por el usuario utilizando diagramas *ADV*, *ADV-Charts* y *Diagramas de configuración*.

En la metodología propuesta a lo largo de esta tesis, las funcionalidades volátiles pueden ser nuevos comportamientos que son agregados al Modelo Conceptual (y que pueden englobar varias clases) o Modelos de Navegación estables y probados, conteniendo nuevos nodos, links, e incluso relaciones con clases conceptuales. Cada funcionalidad volátil es tratada como un subsistema autocontenido y modelado utilizando la metodología OOHDM. La notación es similar a metodologías simétricas para la separación de *concerns* tal como la descripta en la metodología Theme/Doc [37].

En la Ilustración 22 se puede apreciar del lado izquierdo como los modelos core (conceptual, navegacional e interfaz) son empaquetados por capas mientras que, como si fuese un espejo del lado derecho, las funcionalidades volátiles son empaquetadas simétricamente por capas. Para poder combinar estos diseños se utilizan mecanismos de extensión específicos para cada capa de la aplicación Web. A lo largo de ésta tesis se discutirán cada uno de ellos.

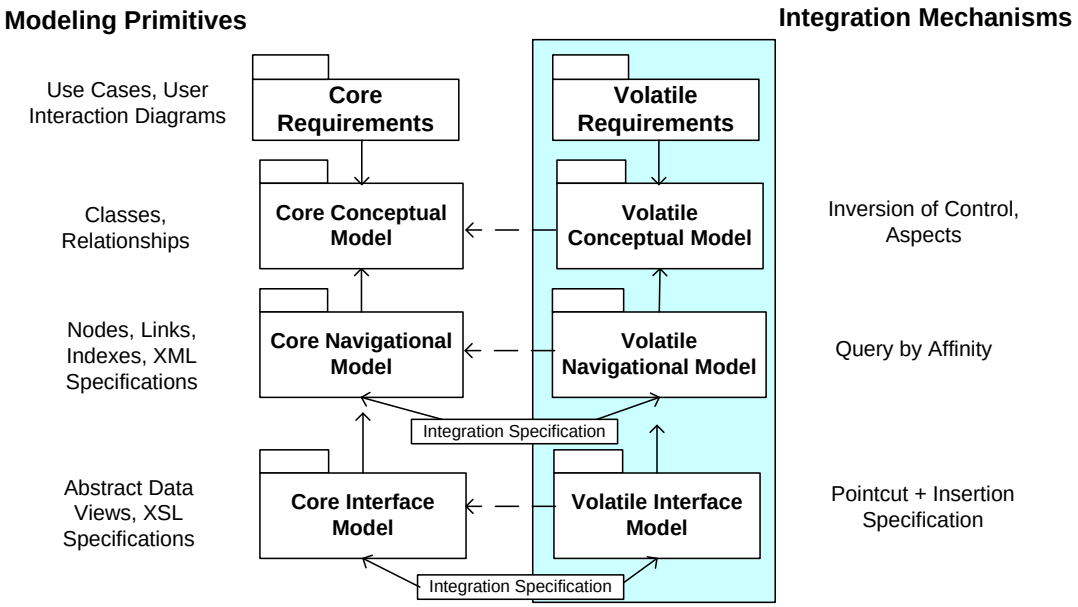


Ilustración 22 Esquema de separación de *concerns* por capas

Basado en las ideas anteriores, la guía de desarrollo que se encuentra esquematizada en la Ilustración 23.

En el diagrama se pueden apreciar flechas desde los modelos (conceptual, navegacional e interfaz) volátiles hacia su contraparte core describiendo dependencias entre el modelo volátil y el

modelo core. Por ejemplo, en el modelo conceptual volátil se puede indicar, mediante una relación UML del diagrama de clases, que una de las nuevas clases posee una referencia a una clase core (definida en el modelo core), o, por otro lado, una definición de comportamiento de una interfaz volátil puede hacer referencia a una interfaz core en caso de que exista una dependencia; un pop-up volátil dependerá de una interfaz principal para funcionar.

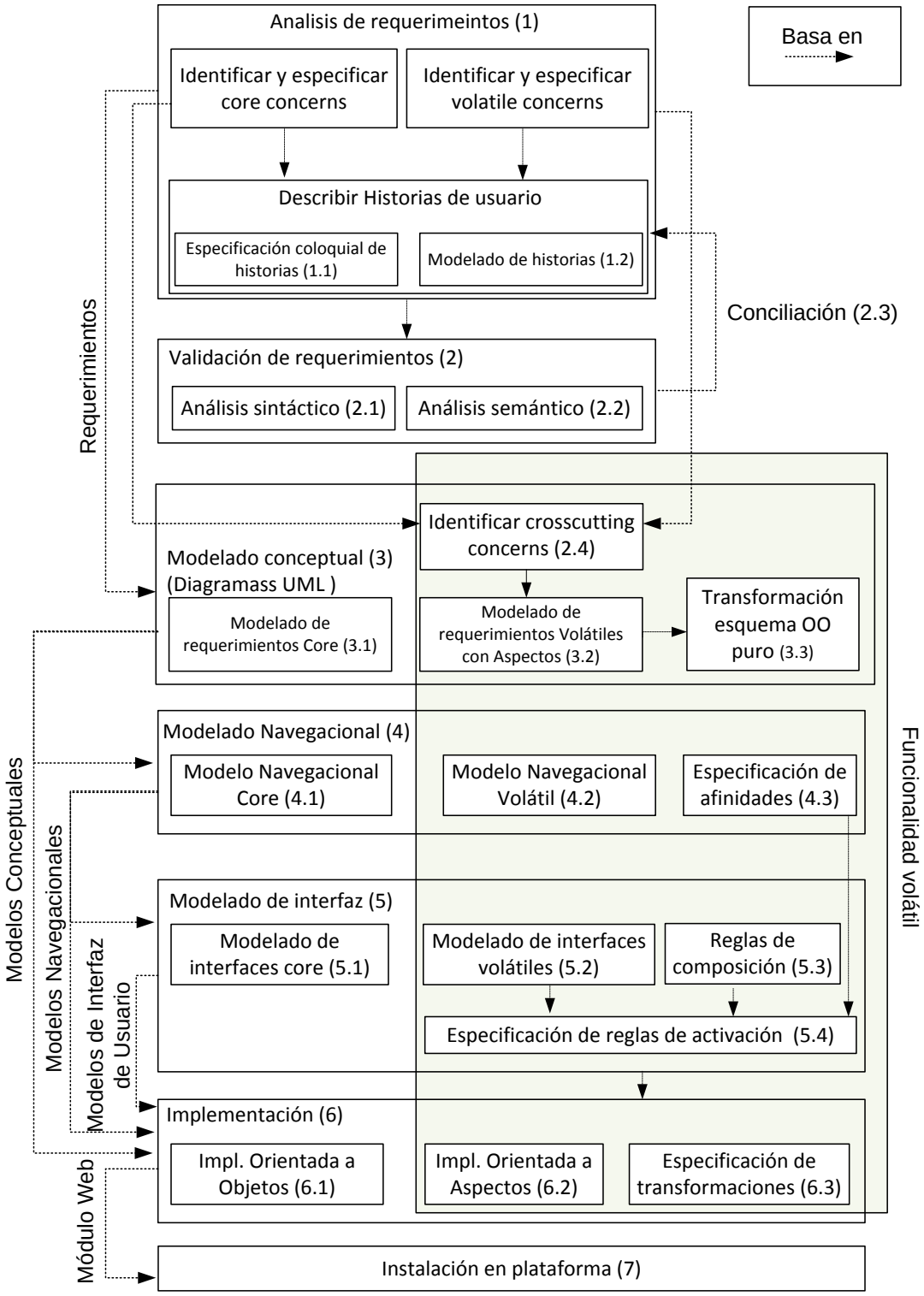


Ilustración 23 Esquema de la metodología

La metodología tiene dos grupos de tareas: una correspondiente a la funcionalidad core y otra correspondiente a la funcionalidad volátil (recuadrado en la imagen). En ambos casos existe una fase inicial donde se realiza el relevamiento de requerimientos (paso 1) siguiendo técnicas tradicionales tal como Joint Application Development, Casos de Uso o con técnicas más novedosas como Historias de Usuarios[10] (paso 1.1) de los métodos ágiles. Para el relevamiento de requerimientos de interacción se utiliza WebSpec (paso 1.2). La novedosa incorporación de la utilización de WebSpec permitirá, tal como será descripto en el Capítulo 6 , la validación de inconsistencia de requerimientos en las etapas tempranas del desarrollo de software. A grandes rasgos, la validación se desdobla en dos actividades: validación sintáctica (paso 2.1) y validación semántica (paso 2.2). Tareas de conciliación de requerimientos (2.3) pueden ser requeridas para corregir las inconsistencias detectadas.

Luego, la metodología propone el diseño de los concerns core de forma tradicional siguiendo la metodología OOHDM produciendo modelos conceptual, navegacional e interfaz. Como se indicó anteriormente, los requerimientos core son modelados independientemente. El primer paso es su modelado conceptual (paso 3.1), luego su modelado navegacional (paso 4.1) y, finalmente, el modelado de las interfaces (5.1). La etapa de implementación (6.1) utiliza plataformas Orientadas a Objetos donde la traducción de los modelos citados resulta directa.

Finalmente los componentes son implementados (codificados) en la plataforma seleccionada e instalados en el servidor de aplicaciones Web.

Hasta aquí, solo se presentó de forma resumida la metodología OOHDM, esta tesis extiende la metodología anterior con las siguientes tareas para el modelado de funcionalidad estrictamente volátil. La funcionalidad volátil es desacoplada de la funcionalidad core por medio de la incorporación de una capa de diseño para la funcionalidad volátil (capa volátil) la cual comprende un modelo conceptual, un modelo navegacional y un modelo de interfaz. Las tareas son:

- Los requerimientos volátiles son analizados utilizando un conjunto de herramientas conceptuales que permitirán identificar el impacto de su incorporación en los diferentes *concern* de la aplicación. De este análisis se detectarán crosscutting concerns que servirán de suma utilidad para el diseñador.

- Los modelos conceptuales de los requerimientos volátiles son diseñados (paso 3.2) utilizando la misma notación que la utilizada para modelar requerimientos core (por ejemplo: diagrama de casos de uso, diagrama de clases UML, diagrama de secuencia UML, y diagramas de interacción de usuarios, etc.) pero utilizando las extensiones de MATA y PS (Sección 7.4) descriptas en la secciones 2.3.1 y 2.3.2 respectivamente. La novedosa incorporación de PS permite modelar cambios volátiles que afectan la estructura de las clases UML mientras que la utilización de MATA permite diseñar los

[10] También conocido como *User Stories.*

cambios de comportamiento de los métodos de las clases. Notar que los requerimientos volátiles no son integrados dentro del modelo de requerimiento core dejando en consecuencia su integración para futuras actividades de diseño.

- Previamente a las tareas de desarrollo, cuando la plataforma destino no soporte la implementación de aspectos de forma nativa, se debe transformar los modelos MATAs (paso 3.3) utilizando un conjunto de heurísticas definidas que principalmente guían la migración de conceptos AOP a construcciones similares utilizando patrones POO (Sección 7.4.2); tal como *Decorator* y *Command* así como también *Inversión de relación*.
    - Nuevo comportamiento es modelados como objetos en el modelo conceptual volátil; ellos son considerados como una combinación de *Commands* y *Decorators* [26] de las clases core.
    - La *inversión de relación* es utilizada para eliminar el conocimiento por parte de los componentes core de los elementos de la funcionalidad volátil. En lugar de hacer a las clases conceptuales core consientes de las nuevas funcionalidades, la relación de conocimiento es invertida.
- Nodos y links pertenecientes al modelo de navegación volátil (paso 4.2) pueden o no tener relaciones con el modelo navegacional core (4.1). El modelo navegacional core es también inconsciente a las clases navegacionales volátiles. Esto es, no hay links u otra referencia desde el modelo navegacional core al navegacional volátil.
    - Una definición llamada *Afinidad* es utilizada para especificar la conexión entre los nodos core y volátiles (4.3). La integración se obtiene en tiempo de ejecución. En otras metodologías dirigidas por modelo, la integración puede ser realizada durante la transformación del modelo core mediante la implementación de las correspondientes transformaciones.
- Las interfaces de usuario correspondiente a cada *concern* (core y volátil) es diseñada (e implementada) separadamente; el diseño de interfaz de las clases core (paso 5.1), descripto en OOHDM utilizando ADV, es inconsciente acerca de las interfaces del *concern* volátil (paso 5.2). Como en la capa navegacional este principio es independiente de la metodología de diseño.
    - Las interfaces core y volátiles (en las capas de ADV e implementación) son combinadas mediante la ejecución de la *Especificación de Integración* (paso 5.3). En esta tesis se propondrá además, para aquellas interfaces orientadas a objetos, el diseño de las mismas utilizando MATA para encapsular los comportamientos *crosscutting*.

- Una vez que los requerimientos volátiles fueron diseñados, se deberá definir el ciclo de vida del mismo mediante las *Reglas de Producción* (paso 5.4). Estas reglas, determinarán cuándo la funcionalidad estará disponible y cuando dejará de estarlo en base a eventos de negocios o fechas determinadas. Estas reglas son diseñadas al finalizar la etapa de diseño dado que las *Reglas de Producción* poseen referencias a *Especificación de Integración* y Afinidad *Navigational*, esta es diseñada casi al final del método.

- La implementación requiere el desarrollo de los artefactos de software para cada uno de los *concern* core y volátiles (pasos 6.1 y 6.2 respectivamente) diseñados en las etapas anteriores de ésta guía. Además, se deberán mapear cada uno de las configuraciones de integración definidas en cada capa. Por ejemplo, en el caso de que se requiera una *Especificación de Integración*, se deberá traducirla a su representación análoga de un documento XSLT (transformación). Este documento, inyecta los cambios estructurales y de comportamiento a las interfaces Web concretas. El framework Cazon, que será presentado en esta tesis, soporta este mapeo de forma semi automática ahorrando gran parte del tiempo y esfuerzo dedicado a esta tarea.

- Finalmente, los artefactos obtenidos deberán ser instalados[11] en el servidor de aplicaciones (paso 7) el cual deberá ser capaz de interpretar las configuraciones de integración (*Especificación de Integración* y *Reglas de Producción*) de forma automática en la plataforma para combinar los *concern* core y volátiles. De esta forma, se obtendrá la aplicación final (paso 8).

En los siguientes capítulos se explicará cómo estos principios han sido puestos en práctica con la metodología OOHDM. Se va a enfocar principalmente en aquellas cuestiones que son específicas de aplicaciones Web (por ejemplo: contenido, navegación, y presentación) e ignorado cambios de granularidad fina en reglas de negocio que aplican solo al modelo conceptual y no impactan en otros modelos ya que está fuera del foco de la tesis; sin embargo, este tipo de requerimiento volátil ha sido estudiado en detalle en la literatura (ver por ejemplo [5]) y puede ser atacarse tanto utilizando aspectos o Decoradores de objetos [26].

La metodología propuesta soporta la incorporación de *features* (también conocido como, composición de módulos monótono [25]) por medio de diferentes tecnologías:

(i) patrones de programación orientados a objetos son adoptados para modelar *features* volátiles en el nivel del modelo conceptual,

(ii) afinidades son utilizadas para enriquecer nodos en el modelo navegacional,

[11] El proceso de instalación de artefactos en servidores es denominado *deploy* en ingles.

(iii) y transformaciones XSL son utilizadas en los modelos de interfaz para agregar nuevas estructuras a las vistas. La eliminación de funcionalidad (también conocido como, composición de módulos no monótono) y propiedades de composición (por ej. conmutatividad y asociatividad [51]) no están cubiertos en esta tesis.

Capítulo 6 Análisis de requerimientos

En este capítulo se discutirá la problemática de los conflictos en la etapa de relevamiento de requerimiento para brinda soporte a aquellas funcionalidades volátiles que puedan contradecir funcionalidad core generando ambigüedad en la misma. En general durante toda la sección se hará referencia a requerimientos conflictivos sin discriminar si son o no volátiles ya que su calidad de volátil no modifica su tratamiento.

En la Ilustración 24 podemos apreciar las tareas específicas al análisis de requerimientos de la metodología donde introduce la especificación de requerimientos utilizando Historias de Usuarios (paso 1.1) y su modelado con WebSpec (paso 1.2). Estos requerimientos serán analizados sintácticamente (paso 2.1) y semánticamente (paso 2.2) en búsquedas de conflictos. En el caso de que se detecte alguno, diferentes tareas de conciliación deberán ser llevadas a cabo (paso 2.3).

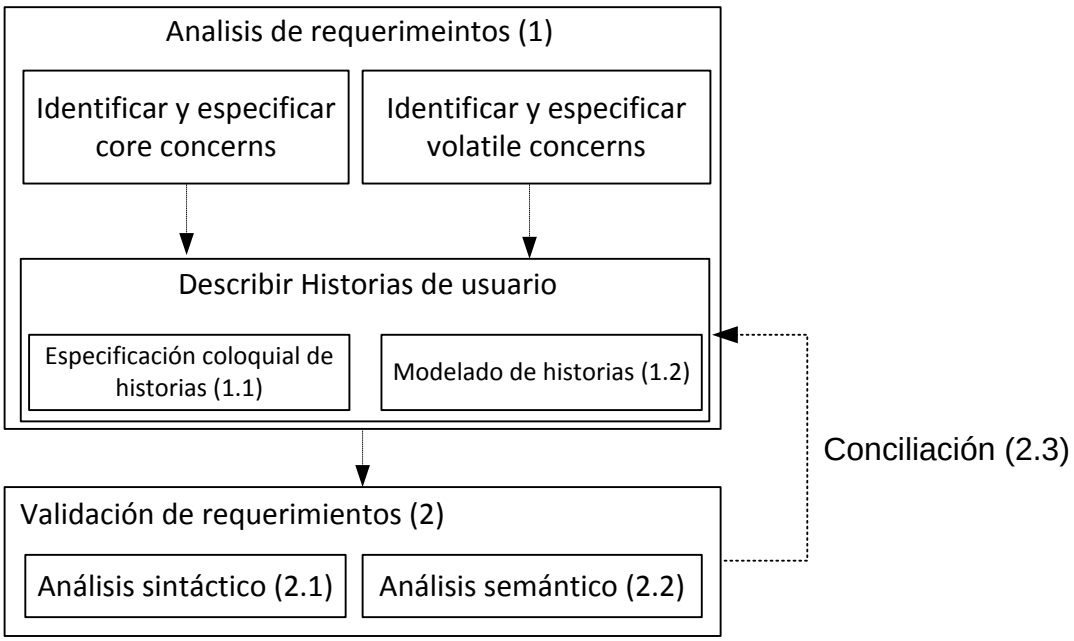


Ilustración 24 Tareas especificas de análisis de requerimiento extraidas del esquema general del metodología

Además de discutir la problemática de conflicto, En este capítulo se mostrará como especificar requerimientos utilizando Historias de Usuarios[72] (*User Histories*).

El análisis de requerimiento de aplicaciones Web implica adquirir un entendimiento de las necesidades de todos aquellos interesados en el sistema; aquellos que están interesados en el mismo negocio corporativo. La mayoría de las veces, los requerimientos son acordados por los interesados de tal forma que la semántica y el significado de cada término o concepto utilizado es bien entendido. Sin embargo, cuando existen diferentes puntos de vista [73] del mismo concepto de negocio, ambigüedades y/o inconsistencias pueden surgir, siendo estas perjudiciales para la Especificación de Requerimientos de Software[12] (ERS). Tradicionalmente, las tareas de conciliación son realizadas utilizando técnicas y herramientas basadas principalmente en reuniones; con el objetivo de eliminar ambigüedad e inconsistencias en los requerimientos. Cuando la inconsistencia en requerimiento no es detectada a tiempo, ellos pueden convertirse en defectos en el software; siendo éste una de las razones más severas de que los proyectos superen el costo estimado [74][75] ya que estos defectos deben ser resueltos en las etapas finales del proceso de desarrollo de software. En este contexto, el esfuerzo para corregir las fallas es varios órdenes de magnitud mayor que corregir los requerimientos en las etapas tempranas del desarrollo del software[75].

Las inconsistencias pueden surgir desde nuevos requerimientos, que introducen nuevas funcionalidades o mejoras a la aplicación, o, incluso, desde requerimientos existentes que cambian durante el proceso de desarrollo. Por ejemplo, un sitio de comercio electrónico que provee un servicio pago de entrega a domicilio de productos, del tipo que ha sido utilizado en los ejemplos hasta el momento, puede planear una promoción para "*Navidad"*, donde algunos productos tienen el servicio de envío sin costo por un período de tiempo; mientras que el resto de los productos mantienen el costo usual del servicio. Este nuevo requerimiento introduce cambios que son percibidos por el usuario porque él puede ver banners promociónales en diferentes páginas. Es destacable que algunos de los requerimientos originales correspondientes al (*Concern*) "Envío a domicilio" son redefinidos por las excepciones del servicio de envío para determinados productos que introduce la nueva funcionalidad, introduciendo ambigüedades: ¿Qué productos tienen la promoción de *envío gratuito*? ¿Cuánto tiempo estará la promoción disponible?

En este capítulo se presenta una técnica de validación y detección de inconsistencias basada en el modelado de los requerimientos de aplicaciones Web, particularmente para aquellos que se presentan durante la navegación e interacción; dos aspectos claves de aplicaciones Web. En consecuencia, Los ejemplos que ilustran la técnica serán modelados utilizando el meta-modelo de WebSpec[31], donde las mismas ideas pueden ser fácilmente utilizadas con otros meta-modelos similares como WebRE[76] o Molic [77]. Utilizando esta técnica se reduce el riesgo de errores y sobre costos en proyectos causados por inconsistencias detectadas en las etapas finales del proceso de desarrollo de Software.

[12]En ingles es conocido bajo el nombre: Software Requirement Specification (SRS)

Las mayores contribuciones de esta sección son tres: una caracterización de inconsistencia de requerimientos de aplicaciones Web dependiendo en la taxonomía del conflictos; una técnica modular para detectar inconsistencia que puede complementar fácilmente cualquier proceso de ingeniería de aplicaciones Web sin importar su estilo: ágil o unificado; y un conjunto de ejemplos que ilustran la técnica.

6.1 Caracterización de conflictos en requerimientos de aplicaciones Web

Durante la especificación de requerimientos, existen casos donde dos o más escenarios que reflejan la misma lógica de negocio difieren sutilmente uno de otro produciendo una inconsistencia. Cuando estas inconsistencias están basadas en comportamientos contradictorios, nos encontramos con un conflicto de requerimientos[78]. Los conflictos están caracterizados por tener diferencias en las características, conflictos lógicos (lo que se espera) o temporales (cuando se espera) entre acciones, o incluso diferencias en terminología que crea ambigüedad. Mientras más dinámicos sean los requerimientos de la aplicación mayor será la posibilidad de encontrar un conflicto.

En este análisis, se enfatizará en la navegación e interacción de usuario de aplicaciones Web que no son cubiertas en las caracterizaciones tradicionales de los conflictos de requerimientos[78]. Por ello, se proveerá una interpretación de de cada tipo de conflicto en el dominio de las aplicaciones Web, utilizando ejemplos simples pero ilustrativos. Se utilizará el meta-modelo de WebSpec para especificar requerimientos en los ejemplos.

6.1.1 Conflictos estructurales: Definición

Los conflictos estructurales corresponden una diferencia entre lo datos esperados por los diferentes usuarios (interesados en el sistema) para ser presentados en una página Web. Un usuario puede exigir que un dato sea mostrado en una página Web que contradice con otro requerimiento de otro usuario. Por ejemplo, un usuario espera una descripción del contenido de un producto como un elemento de interfaz de solo lectura tal como una etiqueta (*Label*), mientras que otro espera el contenido como una lista de paquetes con una descripción general contradiciendo al primer requisito. En la Ilustración 25 se puede ver una *interacción* utilizando WebSpec de un *Producto* (será referenciado como Producto^{v1}) mientras que en la Ilustración 26 se puede ver una interacción de un *Producto* (será referenciado como Producto^{v2}) que difiere del primero en algunos elementos produciendo un conflicto estructural.

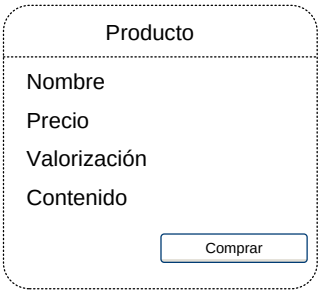


Ilustración 25 Vista de *Producto* donde la descripción del contenido es solo una Etiqueta.

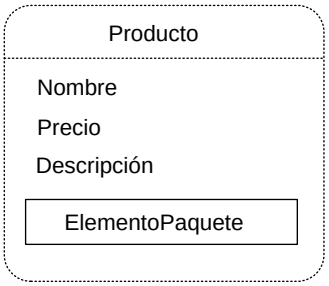


Ilustración 26 Vista de *Producto* donde el contenido del paquete es un listado.

Las diferencias estructurales pueden ser calculadas mediante la operación de conjuntos de diferencia simétrica. A continuación se muestra el cálculo del mismo:

diferencia = ($Producto^{v1}$ – $Producto^{v2}$) U ($Producto^{v2}$ – $Producto^{v1}$) = {Valorización,Contenido, Descripción, ElementoPaquete, Comprar}

6.1.2 Conflictos navegacionales: Definición

Dos requisitos de una aplicación Web pueden contradecir la forma en las páginas son visitadas a través de la navegación de enlaces, o *links*. Por ejemplo, teniendo una página origen pero dos destinos visitados con el mismo evento, es decir, realizando un clic sobre el link nos envía a dos páginas diferentes. Los nodos destinos son diferentes pero el evento que dispara la navegación y las condiciones necesarias son las mismas produciendo una ambigüedad de este requerimiento.

En términos de WebSpec, un conflicto navegacional para una secuencia de navegación dada compuesta por *interacciones* y *navegaciones*, existiendo dos alternativas de navegación disparadas por el mismo evento. Por ejemplo, una definida con WebSpec *navegación* puede definir que tras realizar un clic sobre el botón *Comprar* en la *interacción Producto*, un *CarritoDeCompras* es presentado. Por otro lado, la misma *navegación* tiene como destino la *interacción MétodoDePago*, la cual permite seleccionar un método de pago en lugar de presentar el *CarritoDeCompras*. La Ilustración 27 muestra como mediante un clic en el botón *Comprar* se navega al carrito de comprar mientras que en la Ilustración 28 realizando la misma secuencia de paso se accede a la *interacción* donde se selecciona el método de pago.

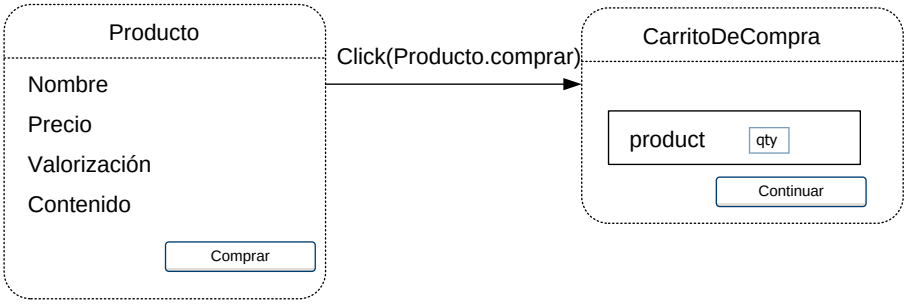


Ilustración 27 Especificación de navegación desde un *Producto* a un *Carrito de compras* disparado por un click al botón *Comprar*.

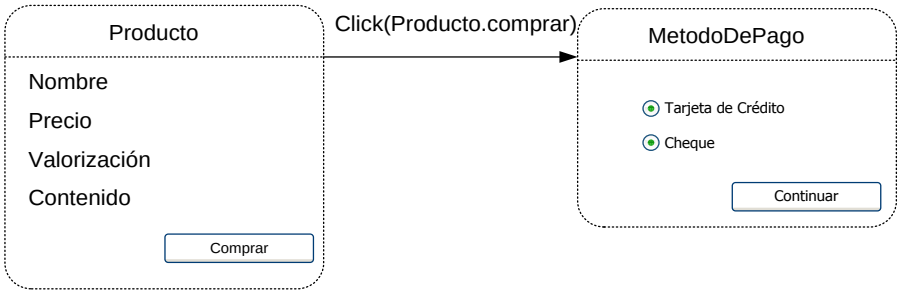


Ilustración 28 Especificación de navegación desde un Producto a la selección de método de pago disparado por un click al botón Comprar.

6.1.3 Conflicto semántico: Definición

Un conflicto semántico ocurre cuando el mismo objeto del mundo real es descrito con diferentes términos. Esta situación puede generar un falso positivo en el proceso de detección de conflictos, debido a que un conflicto puede no ser detectado y nuevos términos son incorporados dentro del espacio del sistema incrementando así su complejidad. Como consecuencia, el mismo objeto de dominio de negocio es modelado en dos entidades utilizando diferente terminología. Por ejemplo, un sitio de comercio electrónico puede equivocadamente definir dos entidades que representan el mismo concepto: *Bien* y *Producto*. En la Ilustración 29 se presenta la *interacción* donde se presenta los datos del *Producto* y en la Ilustración 30 los datos del *Bien*. La información de los objetos es la misma pero modelada utilizando WebSpec con diferentes nombres. Por ejemplo, en la primer interacción referencia al objeto por su "Nombre" mientras que en el segundo caso por un "Identificador".

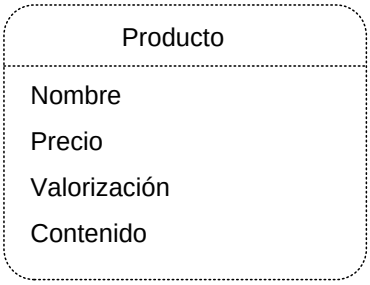


Ilustración 29 Interacción que presenta los detalles de un Producto

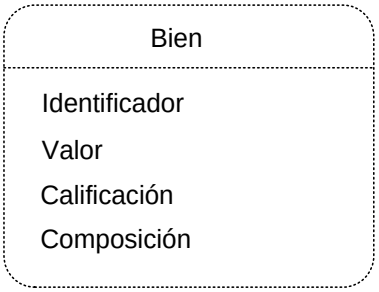


Ilustración 30 Interacción que presenta detalles de un Bien

6.2 Detectando y corrigiendo conflictos

A continuación se presenta una técnica que ayuda a detectar conflictos chequeando la existencia de conflictos con falsos positivos y falsos negativos. La técnica es parte de la metodología general presentada en esta tesis que puede ser incorporada fácilmente en otras metodologías de diseño de aplicaciones Web. En la Ilustración 24 se explota la actividad simplificada en el paso 1 y 2 de la Ilustración 23.

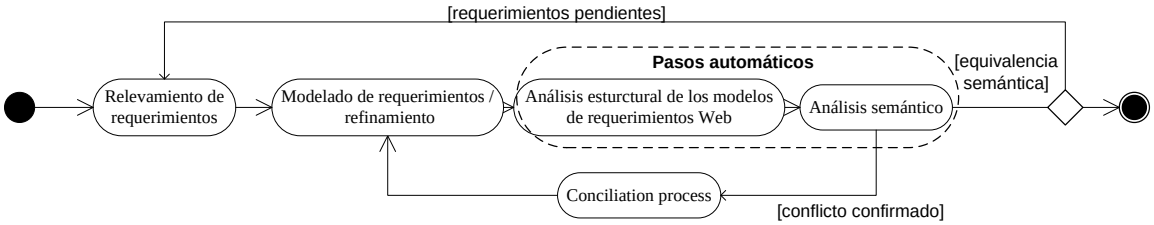


Ilustración 31 Proceso general para la detección de conflictos en requerimientos

En la Ilustración 31, los pasos que pueden ser implementados para ser automatizados son agrupados con bordes punteados y el resto de los pasos son aplicados manualmente.

A continuación se describen cada uno de los pasos de la figura (nótese que los pasos 1 y 2 ya fueron referenciados en la metodología principal pero a fines prácticos serán repasados):

A. **Relevamiento de requerimientos**: Utilizando técnicas de relevamiento de requerimientos bien conocidas tal como reuniones, encuestas, Joint Application Development[79]), etc. Una especificación de requerimientos de software (usualmente en lenguaje natural) es producida. En el caso de un proceso de desarrollo subyacente ágil, una descripción breve es usualmente utilizada con las *historias de usuarios* [72]; los *casos de uso* son comunes en un estilo unificado[80] (unified process).

B. **Modelado de requerimientos**: Los requerimientos en aplicaciones Web son formalizados utilizando un Lenguaje Específico de Dominio[81] (DSL, *Domain Specific Language*) tal como WebSpec, WebRE o Molic. Esta formalización es esencial durante el proceso de validación con stakeholders. Por medio de la utilización de un DSL de requerimiento, el proceso de validación puede ser automatizado.

C. **Control de consistencia (no presente en el diagrama)**: por cada nuevo requerimiento se debe verificar su consistencia con respecto a los requerimientos originales ejecutando los siguientes pasos:

   i. **Análisis estructural de los modelos de requerimientos Web**: utilizando una comparación algebraica de los modelos, se pueden detectar conflictos estructurales y navegacionales. Además, los caminos navegacionales son evaluados para controlar su consistencia.

   ii. **Análisis semántico**: Los conflictos candidatos (detectados en el paso C.i) son analizados y las equivalencias semánticas son detectadas. Dos modelos son equivalentes semánticamente si a pesar de tener una sintaxis diferente, expresan lo mismo. Para cada conflicto candidato, tanto el nuevo requerimiento y el requerimiento original son traducidos a una forma mínima que utiliza utilizando constructores simples y disminuyendo el nivel de refinamiento. Finalmente, son comparados y, en el caso de que sean iguales, se detecta un conflicto semántico. De esta forma, se destaca cuál es el propósito del escenario abstrayéndose de la metáfora de comunicación (*Navegación* y *Comportamiento rico* definidas en el meta-modelo de WebSpec) así como también de los widgets utilizados para presentar la información. En consecuencia se obtiene un diagrama simplificado útil para entender el propósito del escenario en lugar de perderse en detalles que no contribuyen a su semántica.

   iii. **Proceso de conciliación**: una vez que la existencia de conflictos es confirmada, se debe comenzar a conciliar requerimientos. El proceso demanda el establecimiento de un canal de comunicación entre todos los usuarios relacionados al conflicto. Todos aquellos que definan o se vean comprometido tanto por el nuevo requerimiento que introduce el conflicto como el conjunto de usuarios relacionados con el requerimiento original.

   iv. **Refinamiento**: Una vez conciliado los requerimientos, ajustes y refinamientos deben ser realizados para remover los conflictos detectados en el modelo del nuevo requerimiento y alcanzar el estado consistente con respeto a los requerimientos originales. Finalmente, se reinicia el proceso para verificar que los otros requerimientos no introduzcan nuevas inconsistencias.

El proceso es aplicado iterativamente cada vez que un requerimiento es detectado. El requerimiento entrante es validado con cada uno de los requerimientos ya consolidados del espacio del sistema.

6.3 Relevamiento de requerimientos y modelado de requerimientos

Con el objetivo de describir clara y efectivamente los pasos A y B mencionado anteriormente, se utilizará como ejemplo funcional el desarrollo y extensión de un sitio de comercio electrónico. En las siguientes secciones, los ejemplos harán foco en dos conjuntos de requerimientos: aquellos relevados en la primera versión de la aplicación (originales) y aquellos que surgieron de forma inesperada durante la evolución de la aplicación; requerimientos volátiles (requerimientos que dan origen a funcionalidad volátil). El primer conjunto comprende los requerimientos "Mostrar información del producto" y "Venta de Juegos". El segundo está compuesto por "Mostrar resumen de producto", "Venta de teléfonos móviles" y "Venta de libros electrónicos". En la Ilustración 32 se muestran las historias de usuarios [72] derivadas de los requerimientos anteriores. Los diagramas de WebSpec correspondientes serán mostrados en los subsiguientes pasos.

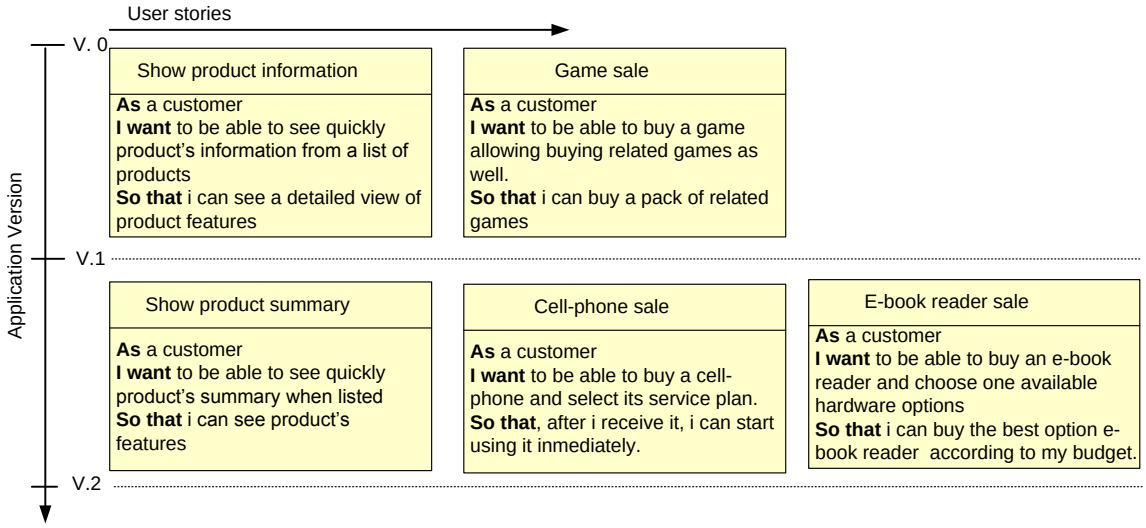


Ilustración 32 Historias de usuarios a partir de los requerimientos obtenidos

6.4 Detectando conflictos sintácticos

La actividad inicial relacionada al paso *C.i* corresponde a la detección de conflictos denominados candidatos. Un conflicto candidato surge cuando un conjunto de diferencias sintácticas entre modelos no es vacía. Estas diferencias pueden ser consecuencia de ausencia de un elemento en un modelo pero presente en otro, la utilización de dos widgets diferentes para describir la misma

información, y, finalmente, una diferencia en la configuración de un elemento presente en ambos modelos tal como los valores de las propiedades de dicho elemento. Esta situación puede surgir cuando dos clientes diferentes tienen diferentes vistas o perspectivas de una funcionalidad, o cuando un requerimiento contradice uno originario. Como consecuencia de utilizar una herramienta formal para describir requerimientos, la tarea de detección puede ser implementada razonando sobre la especificación del requerimiento; en este caso un modelo WebSpec. Para esta tesis, se desarrolló una extensión una extensión a la herramienta WebSpec[82] utilizando sentencias OCL[83].

La detección de conflictos estructurales puede ser implementada mediante una operación de comparación entre *interacciones* con el objetivo de detectar ausencia de elementos o diferencias en la constitución de un elemento. Puesto que las *interacciones* WebSpec son contenedores de widgets, se puede aplicar operaciones de diferencia entre conjuntos para la detección de inconsistencias. Por ejemplo, una versión de interacción de *Producto* llamada *Producto* de la Ilustración 25 tiene las etiquetas *Nombre*, *Valorización* y *Contenido*, y un botón *Comprar* y, por otro lado, una versión diferente de la Ilustración 26 comprende las etiquetas *Nombre*, *Precio*, y *Descripción*, y una lista de *ElementoPaquete*. Luego de aplicar la operación de diferencia simétrica, los siguientes widgets difieren entre sí: *Valorización Contenido*, *Agregar*, *Descripción* y una lista de *ElementoPaquete*.

Se debe notar que, en la operación de comparación, dos elementos son iguales si y solo si ellos tienen el mismo nombre, tienen el mismo tipo de widgets y configuración compatible.

Para detectar conflictos navegacionales, se debe analizar la existencia de dos o más navegaciones salientes de un nodo dado que tengan eventos idénticos que los disparen pero tengan destinos diferentes. La tarea es bastante sencilla; dado que las navegaciones son descriptas por una guarda (que sirven con precondiciones necesarias para poder evaluar las acciones) y un conjunto de acciones que disparan la navegación dispara, las *navegaciones* para una *interacción* dada deben ser comparadas entre ellas teniendo en cuenta las guardas y sus conjuntos de acciones. El desafío principal de este procedimiento es validar cuándo el conjunto de acciones que corresponde a las navegaciones son semánticamente equivalentes considerando que las acciones pueden ser sintácticamente diferentes.

6.5 Análisis semántico

En el paso *C.ii*, se toma el resultado del análisis estructural de modelos de requerimientos utilizando WebSpec (una lista de conflictos candidatos) que deberán ser analizados con el objetivo de detectar falsos positivos; en otras palabras, conflictos tentativos que en realidad no son conflictos debido a que las especificaciones describen el mismo requerimiento. Los casos falsos positivos solo pueden ser identificados con un previo análisis semántico.

Esta problemática ya ha sido estudiada en [84][85] donde modelos son analizados con el propósito de exponer sus objetivos subyacentes. Cuando los objetivos son diferentes, nos encontramos con un conflicto confirmado.

En esta tesis se utilizará una técnica propuesta en [84] que propone tener una vista semántica adicional para los requerimientos complementando así la vista sintáctica existente. Para obtener la vista semántica, los modelos de requerimientos son simplificados obteniendo representaciones refinadas formadas sólo con elementos simples y relevantes.

Esta técnica está conformada por dos partes: un meta-modelo llamado vista semántica, definido como un conjunto reducido del DSL de requerimientos utilizado (en este caso WebSpec), y una transformación que tiene como parámetro el modelo de requerimiento y como salida un modelo semántico. En ésta transformación se toman elementos desde el modelo y los lleva a la vista semántica.

Los modelos comprometidos en un conflicto (el original y el nuevo donde se introduce el conflicto en cuestión) son transformados a la vista semántica donde los modelos obtenidos son finalmente comparados sintácticamente; es decir elemento contra elemento y las propiedades de cada uno contra las otras. Para cada conflicto detectado en el paso C.i, esta metodología ayuda a detectar falso positivos porque las construcciones semánticamente equivalentes implican que diferentes modelos representan el mismo requerimiento. Complementariamente, los modelos son comparados cuando no hay conflictos sintácticos para traer a la luz falsos negativos.

Se utilizará como vista semántica un meta-modelo WebSepc reducido donde las jerarquías de *Transition* y *Container* fueron retiradas. La jerarquía de *Transition* está formada por dos especializaciones – *Navigation* y *RichBehaviour* – que fueron removidas con el objetivo de permitir enfocarse en determinar cuál es el propósito de la interacción independientemente del patrón de interacción utilizados: navegación tradicional o interacción RIA. Cuándo los contenedores (*Container*) no tienen el mismo nombre, ellos son removidos para reducir la complejidad de composición y evitar agregaciones de objetos innecesarios. En la Ilustración 33 se distinguen las clases eliminadas del meta-modelo de WebSpec con un recuadro de líneas punteadas rojas. El meta-modelo obtenido se denomina versión semántica; en este caso, versión semántica de WebSpec.

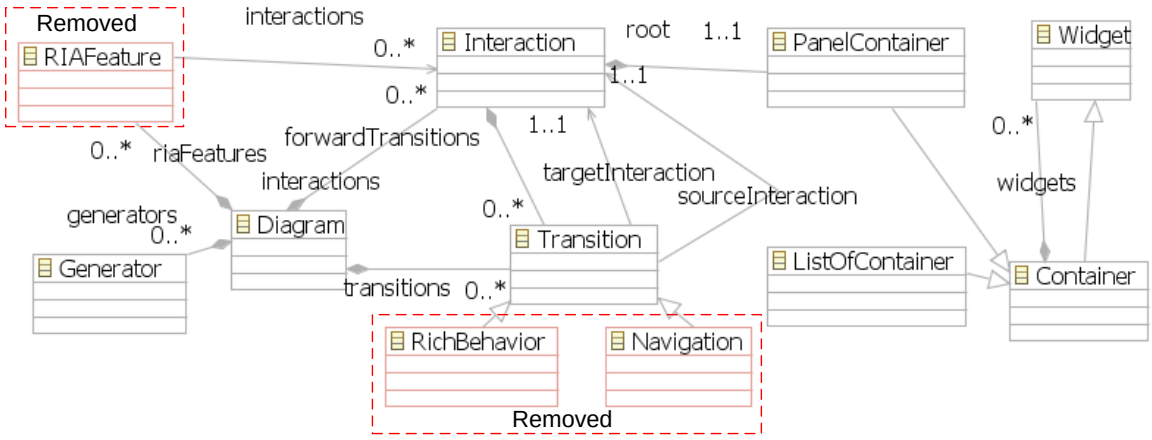


Ilustración 33 Versión semántica del Meta-modelo WebSpec

Finalmente una transformación de un modelo debe convertir un modelo WebSpec en un modelo semántico para proveer un entendimiento simplificado del mismo.

Si otro meta-modelo de requerimientos es utilizado tal como WebRE, un conjunto diferente de reglas de transformación debe ser definida donde cada una debe incrementar el nivel de abstracción de tal forma que el propósito del modelo sea enfatizado.

Reglas base definidas en base al meta-modelo WebSpec que serán utilizadas por la transformación son:

- Los elementos de tipo *TextField* deshabilitados son transformados en widgets de tipo *Label*. Los *TextFields* poseen al menos dos estados: habilitado y deshabilitados (conocidos también como estados de escritura/solo-lectura). El primero permite ingresar datos a los usuarios mientras que el segundo no. Cuando los *TextFields* son deshabilitado, ellos cumplen una función de solo lectura tal como los *Labels*, por ello son reemplazados por etiquetas (*Label*) simples.

- Los elementos de tipo *Links* son transformados en elementos de tipo *Button*. Los Links y Buttons (Botones) son usualmente utilizados para disparar una acción. En consecuencia, se normalizan ambos a un solo elemento: un *Button*.

- Elementos de tipo *Links* y *Buttons* no referenciados en guardas (como por ejemplo en acciones de transiciones) son removidos para refinar el modelo ya que estos elementos no proveen ninguna funcionalidad al estar huérfanos.

- Las transiciones, *Navigations y RichBehaviour*, son simplificados dentro de una abstracción general de *Transición*. Esta regla permite destacar en el requerimiento las estructuras de datos utilizadas en lugar de la forma en que ésta es accedida. Finalmente,

aquellas acciones que disparan la interacción en las transiciones *(Navigations y RichBehavior*) son eliminadas ya que al normalizarse la transición en un concepto más abstracto no son necesarios.

Para detectar si un conflicto identificado en la etapa de análisis sintáctico es de hecho un conflicto, la transformación es aplicada sobre los requerimientos que participan en la contradicción produciendo un modelo que respeta la versión semántica del meta-modelo. Si las transformaciones producen el mismo modelo semántico, no existe conflicto semántico.

Por otro lado, los falsos negativos, conflictos que no son detectados en el análisis sintáctico, son detectados llevando los modelos de los requerimientos (que no están en conflictos) a la vista semántica. De esta forma, mediante las reglas de eliminación de especializaciones, por ejemplo el caso de *Navegation* y *RichBehaviour* en WebSpec, se podrían detectar conflictos navegacionales que a simple vista no son conflictos ya que ellos son indicados con dos especializaciones diferentes de una *Transition*.

El siguiente ejemplo tiene el propósito de ilustrar como conflictos semánticos son detectados; en particular un caso de falso negativo. En la Ilustración 34 y Ilustración 35 requerimientos llamados "Mostrar la información de un producto" y "Mostrar resumen de producto" representan la misma idea de interacción pero utiliza dos patrones de interacción diferentes: navegación Web tradicional y patrón RIA *Hover-Reveal* [86].

En la transformación, un conjunto de reglas estrictamente relacionadas al meta-modelo de requerimientos Web utilizado son aplicadas sobre el modelo de entrada obteniendo la ya comentada vista semántica. Estas reglas están basadas en heurísticas definidas por el ingeniero de requerimientos. El es responsable de detectar patrones de ambigüedad y mejorar iterativamente el conjunto de reglas a partir de las lecciones aprendidas de la aplicación de las mismas.

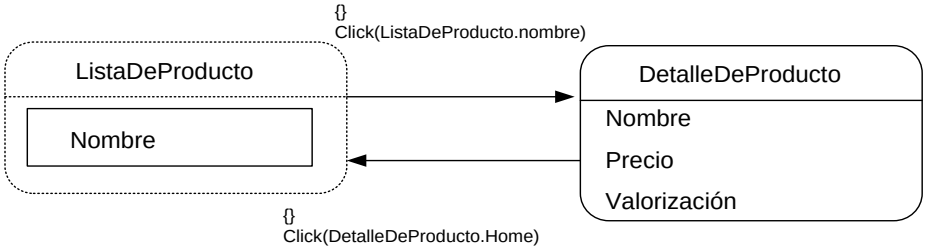


Ilustración 34 Especificación de un requerimiento con navegación convencional.

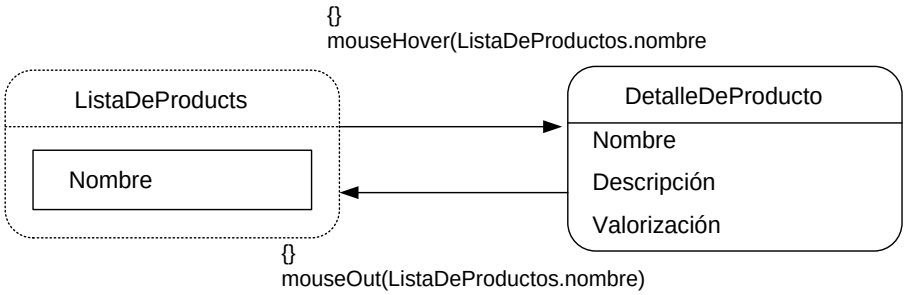


Ilustración 35 Interacción basada con características RIA.

La Ilustración 34 define que luego de realizar un *click* sobre el nombre de un producto, el link es navegado y un detalle del producto es mostrado. Por el otro lado, en la Ilustración 35, cuando el puntero del mouse es ubicado sobre el nombre del producto, un detalle del producto surge. Es destacable que ambos modelos de requerimientos tienen el mismo propósito pero son descriptos utilizando diferentes constructores de WebSpec.

El resultado de aplicar la transformación para ambos modelos conflictivos es un par de diagramas normalizados que deben ser sintácticamente comparados con el objetivo de detectar diferencias. En la Ilustración 36 y Ilustración 37 se muestra el resultado de aplicar la transformación a los ejemplos presentados en las Ilustración 34 y Ilustración 35 respectivamente donde dos transiciones de diferentes tipos (*Navigation* y *RichBehavior*) fueron normalizadas a una *Transition* abstracta.

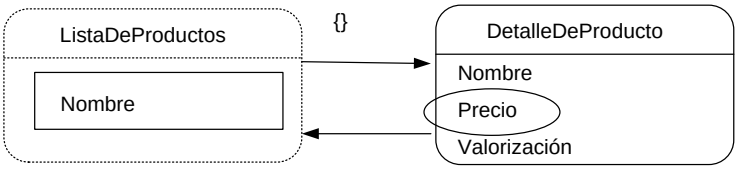


Ilustración 36 Requerimiento de navegación convencional normalizada en la vista semántica.

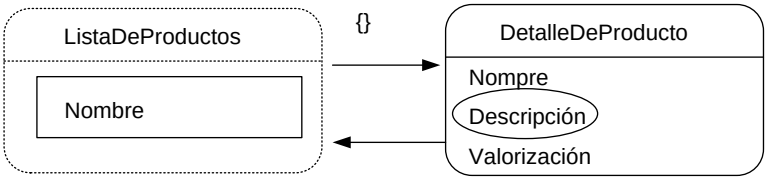


Ilustración 37 Requerimiento con comportamiento Rico normalizado a una vista semántica.

Finalmente un conflicto semántico es detectado debido a que ambos modelos no son sintácticamente iguales en la vista semántica ya que las etiquetas de *Precio* y *Descripción* no

están presentes en ambas interacciones *DetalleDeProducto* (destacado con una elipse en Ilustración 36 e Ilustración 37).

6.6 Proceso de conciliación

Hasta ahora, se ha mostrado como detectar conflictos que deben ser resueltos para mantener el SRS correcto y completo. A continuación se presentará un conjunto de heurísticas que ayudaran a resolver conflictos estructurales y navegacionales correspondiente al paso *C.iii*.

En el caso de conflictos estructurales, la ausencia de un widget dado en un modelo pero presente en otro, se puede optar por una posición optimista entendiendo que la mejor solución es incluir el elemento como una nueva mejora cuando éste no está presente. Esta idea tiene origen desde el hecho de que nuevos requerimientos pueden mejorar funcionalidades definidas en otros requerimientos; en consecuencia los widgets del nuevo requerimiento pueden enriquecer interacciones existentes. Si bien se pueden definir muchas estrategias, en este caso se optó por la adopción de la optimista.

A la izquierda de la Ilustración 38 se muestran nuevos requerimientos identificados como tres *user stories* diferentes. En la derecha de la figura se muestran cómo, a medida que nuevos requerimientos arriban, se constituye un modelo agregado con el modelo anterior incluyendo las mejoras introducidas por una *user story*. La versión 1 del modelo consolidado se constituye del *user story* 1 por ser la primer versión; la versión dos del modelo incorpora la interacción *C*, elementos introducidos en la interacción A^2, y una la navegación *n2*; y por último la versión 3 del modelo consolidado no incorpora ningún elemento ya que simplemente es una proyección del modelo consolidado (todos los elementos de la user story 3 ya fueron definidos anteriormente).

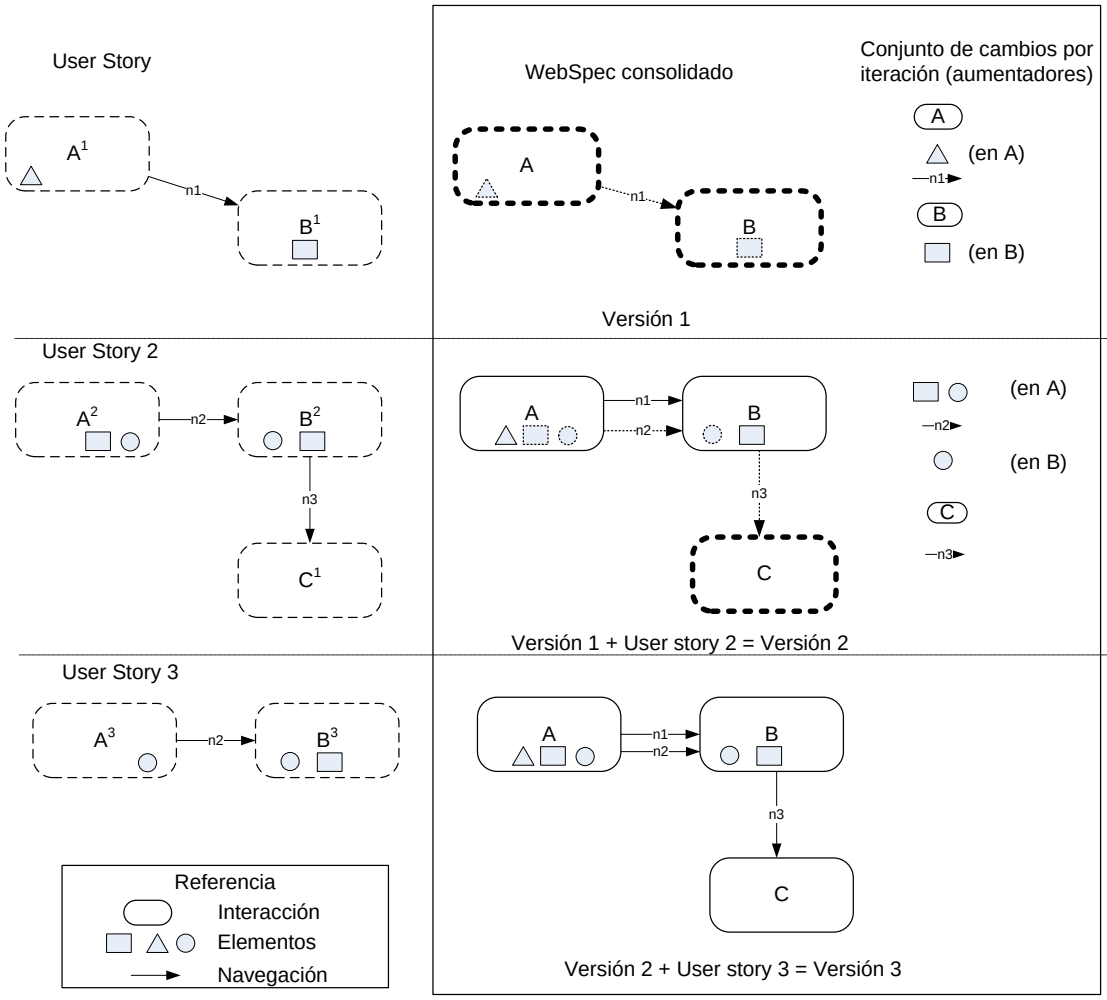


Ilustración 38 Evolución de modelos por nuevos requerimientos

Por otro lado, la incompatibilidad de tipos de widgets demanda un análisis más profundo para entender el contexto de la diferencia. Por ejemplo, si nos encontramos con un widget de tipo *Combo* y en otra interacción disponemos de un Widget de tipo *Field* con el mismo nombre, la diferencia debe ser estudiada.

Los conflictos navegacionales expresan ambigüedad en la forma en que la aplicación es navegada, teniendo dos destinos diferentes (*Interactions* de WebSpec) en navegaciones (*Navigation*) lanzadas por el mismo evento. Esta situación es naturalmente resuelta enriqueciendo los escenarios WebSpec de tal forma que el conflicto es disuelto por el incremento en el nivel de detalle. Dado que en esta tesis estamos utilizando WebSpec como una herramienta de modelado de requerimientos, existen dos estrategias posibles para desambiguar: agregar cláusulas de

precondición o extender el escenario con más navegaciones. En el último caso, es posible enriquecer el contexto básico de una navegación (su propia definición y clausulas que la disparan) teniendo en cuenta de qué forma se accedió a la interacción que la dispara.

Otra situación de conflictos se da cuando diferentes clientes (o interesados en el sistema) puede proveen especificaciones sutilmente diferentes para el mismo objetivo de la aplicación. Sin embargo, existen escenarios donde se es más propenso a enfrentar inconsistencias tal como la presencia de objetos de negocios con jerarquía; objetos que disponen de una clasificación tal como *Vehículo* con *Auto* y *Ciclomotor*. En la etapa de recolección de requerimientos, las jerarquías de objetos de negocios pueden no ser claramente detectadas y definidas, y, como consecuencia, varios objetos de negocio que son estructuralmente diferentes son referenciados con el mismo nombre.

A continuación se presentan tres requerimientos Web que convergen en un conflicto navegacional en el contexto de la aplicación Web de comercio electrónico que hemos utilizado hasta el momento:

- El gerente de ventas de video juegos requiere que desde la página de un producto video juego se debería poder acceder a paquete de juegos relacionados al video juego actual. Esto ayudaría a vender la última versión del juego con una versión más vieja.

- Un vendedor de lectores portables de libros electrónicos (e-book) solicita que para la venta de este producto, las alternativas de configuración del mismo tal como hardware (Wi-Fi, 3G, Memoria, etc.) y software deben ser descriptos.

- Finalmente, un gerente de la sección de venta de teléfonos móviles especifica que después de seleccionar un teléfono móvil, el cliente debe elegir entre una variedad de planes antes de que el producto personalizado sea agregado al carrito de compras.

Desde estos requerimientos se obtiene, a primera vista, una especificación correcta en lenguaje natural donde no hay ambigüedades. Cuando se modelan estas necesidades se encuentra que todos ellos tienen el mismo objetivo: "vender un producto". Luego de analizar los modelos, se detecta un conflicto navegación. Aplicando la técnica de validación de requerimientos definida en este capítulo, podemos detectar las inconsistencias navegacionales que deben ser resueltas: bajo el mismo camino – accediendo a la *Lista de Productos*, realizando un click en el link *Nombre* y accediendo el *Producto* – tres interacciones diferentes pueden ser accedidas tal como se puede apreciar en la Ilustración 39 .

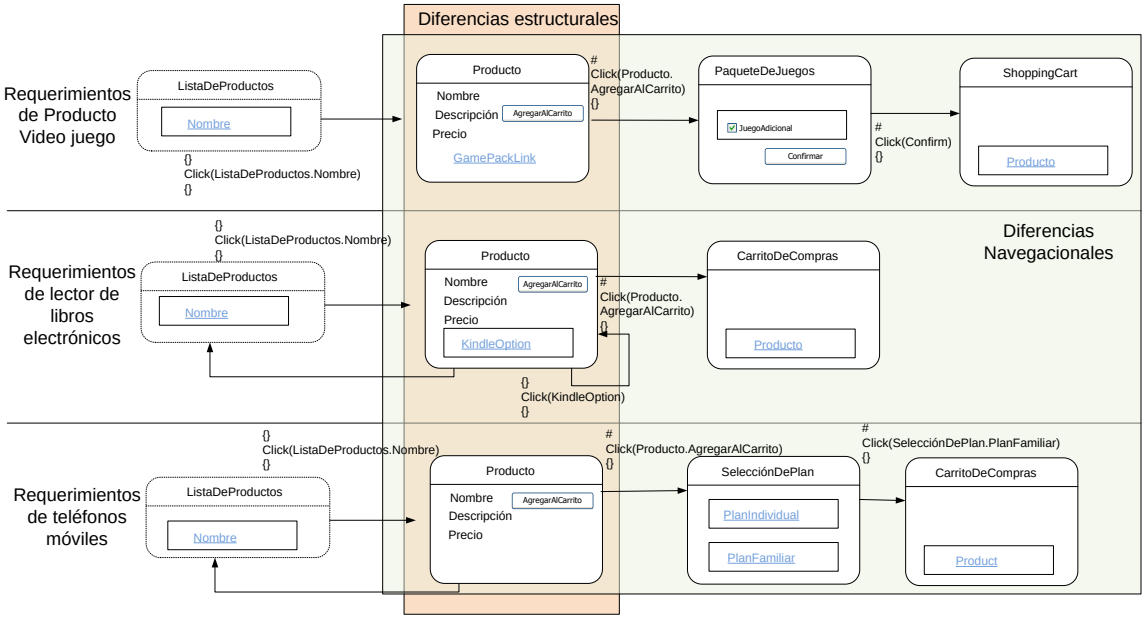


Ilustración 39 Modelos para la venta de un producto con conflictos navegacionales

Con el objetivo de solucionar este conflicto, se ha optado por enriquecer escenarios introduciendo interacciones de inicio (aquellas donde se inicia el caso de uso). En la Ilustración 40 se muestran las tres nuevas interacciones y navegaciones que desambiguan el conflicto.

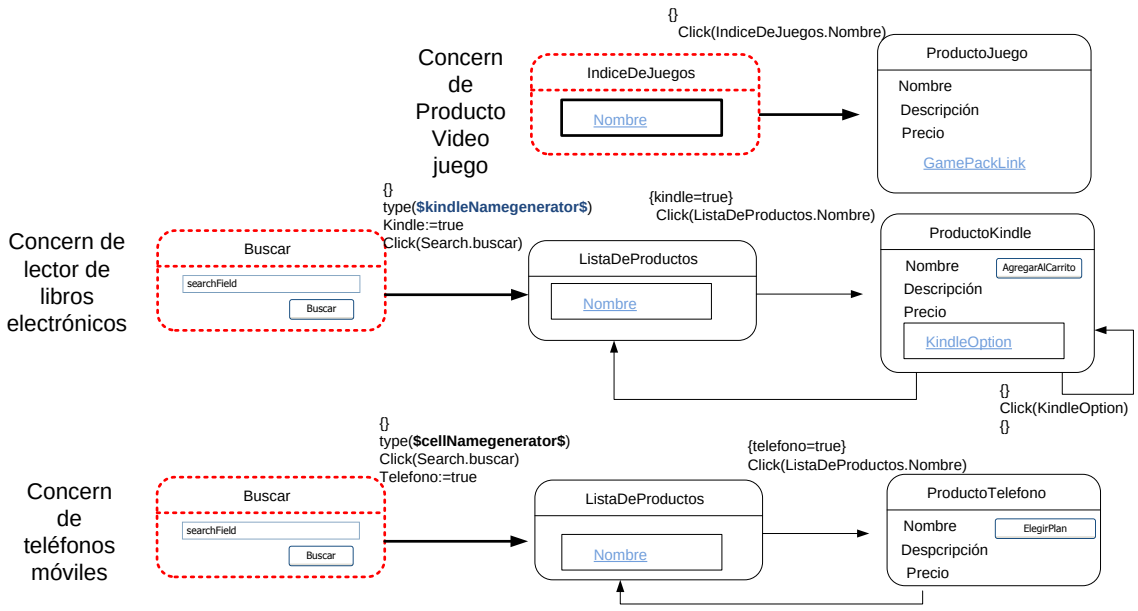


Ilustración 40 Modelos consolidados luegos de ajustar su definición

En el caso del *concern* de video juegos, una interacción particular que juega el rol de índice es provista. Por otro lado, cuando se estudió la venta del lector de libros electrónicos y los teléfonos móviles con los interesados, se detectó que cada tipo de producto es accedido por una búsqueda de producto previa. Por lo tanto, fue acordado enriquecer los escenarios con una interacción que permite buscar un producto donde un generador con un nombre de producto específico fue usado para cada tipo (*kindeNameGenerator* y *cellNameGenerator*). Por otro lado, para resolver el conflicto estructural de la interacción Producto, se definió nuevas interacciones para describir las tres *interacciones* correspondientes a los productos manipulados: *ProductoJuego*, *ProductoKindle* y *ProductoTelefono*.

6.7 Herramienta CASE de soporte

Como parte del trabajo de esta tesis se ha extendido la herramienta WebSpec[82] con un soporte de razonamiento que permite detectar inconsistencias en el proceso de modelado de requerimientos. La herramienta provee un motor de chequeo basado en el sistema de consulta de Eclipse EMF OCL [83]. Por medio de la ejecución de consultas OCL que implementan las reglas definidas en este capítulo para la detección de inconsistencias sobre diagramas, se detectan tanto inconsistencias estructurales como navegacionales. La herramienta automatiza el análisis estructural de los modelos de requerimientos Web. Su principal propósito de asistir al analista de requerimientos en las actividades de modelado, gestión, y control de consistencia de los requerimientos.

La herramienta genera un reporte que lista los conflictos detectados y los widgets comprometidos por los conflictos. Luego, a partir de las inconsistencia detectadas, se presenta una lista de refactoring candidatos que resuelven las inconsistencias. Estos refactoring pueden ser de aplicación automática o semiautomática cuando requieren parámetros de entrada para configurar su ejecución. Es importante destacar que muchos de los problemas no triviales y por ende solo alguno de ellos dispondrán de refactorings que lo resuelva. El analista decidiría cuál de los refactoring es la mejor opción para ser aplicado y más tarde la herramienta aplicaría los refactorings seleccionados sobre el diagrama WebSpec.

La Ilustración 41 muestra un requerimiento para permitir comprar un producto; los productos elegidos por el usuario pueden ser vistos en el *Carrito de Compras*. Por otro lado, la Ilustración 42 muestra un requerimiento inesperado introducido por el periodo de *Navidad* que indica que se debe mostrar una página de publicidad (volátil) como resultado de seleccionar un producto. Está página nueva corresponde a una funcionalidad volátil ya que surge de forma inesperada y permanecerá en el sistema hasta que las ventas del periodo hayan disminuido.

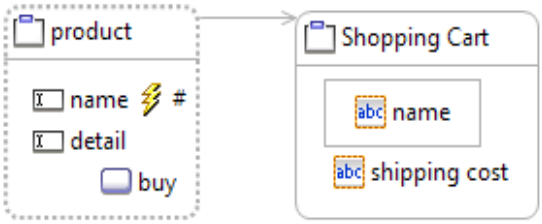


Ilustración 41 Requerimiento de compra de producto

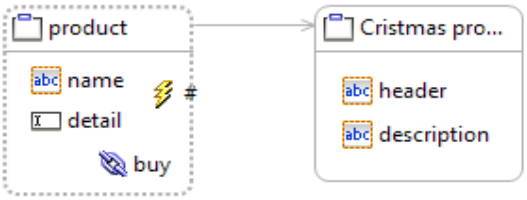


Ilustración 42 Página de publicidad volátil por navidad

Cuando se analiza la consistencia de los modelos, la herramienta WebSpec detecta y reporta conflictos como se puede apreciar en la Ilustración 43; en este caso conflictos estructurales y navegacionales. El conflicto estructural surge del hecho de que el concepto *name* está definido con dos widgets diferentes (*TextField* y *Label*) y el conflicto navegacional surge debido a que el mismo evento dispara una navegación a dos widgets diferentes. Finalmente, cuando las inconsistencias son detectadas, se presenta una lista de refactoring candidatos tal como se puede apreciar en la Ilustración 44.

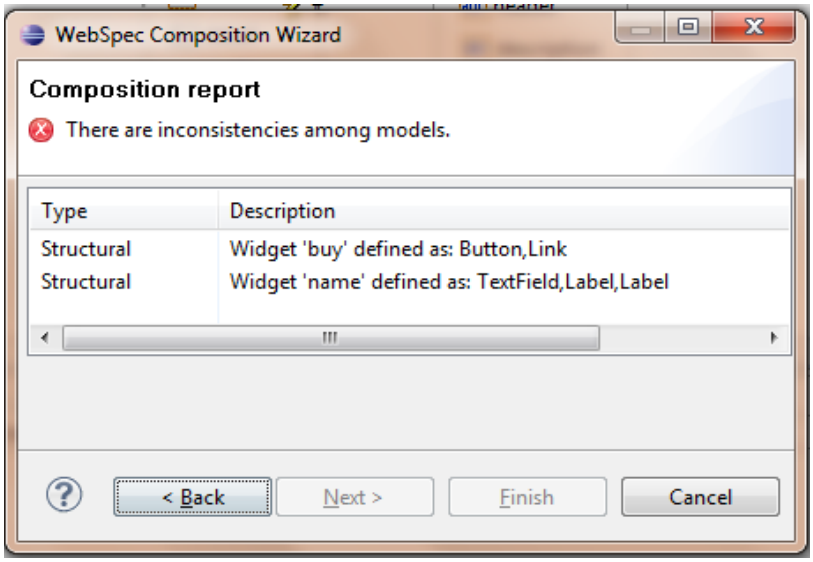


Ilustración 43 Reporte de inconsistencia de la herramienta WebSpec.

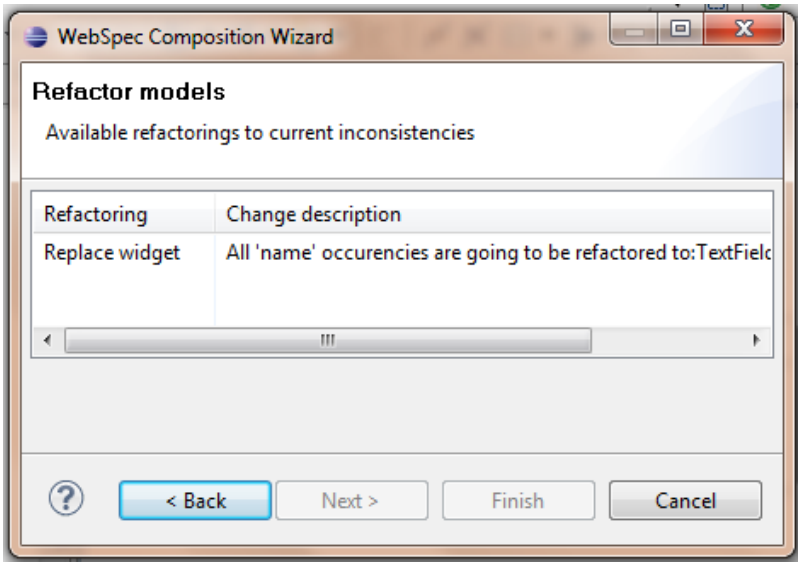


Ilustración 44 Lista de refactorings posibles para las inconsistencias activas.

En resumen, la herramienta provee facilidades para el modelado y control de consistencia de requerimientos Web utilizando el meta-modelo WebSpec. De esta forma, el control de consistencia se ve automatizado disminuyendo los tiempos necesarios para el análisis manual de estos modelos ahorrando el tiempo y esfuerzo necesario resolver las fallas del sistema detectadas en las etapas finales del proceso de desarrollo de Software.

Capítulo 7 Modelo Conceptual

El modelo conceptual de una aplicación Web (también conocido como modelo de aplicación, dominio, o contenido) está enfocado en definir los conceptos de una aplicación y sus atributos así como también el comportamiento asociado a estos. Cuando éste es definido utilizando la metodología OOHDM (u otra tal como UWE), éste es un modelo orientado a objetos descripto con UML y comprende clases, atributos y métodos de éstas, y asociaciones entre clases.

La funcionalidad volátil pude suponer nuevas clases de contenido (por ej. una clase que modela el tipo de contenido Video), o la modificación (volátil) de la estructura una clase existente o del comportamiento de la aplicación.

En este capítulo se discutirán las tareas correspondientes al paso 3 del enfoque general de la metodología presentada en el capítulo 5. En la Ilustración 45 se presenta un extracto del esquema general.

Dado que el modelado conceptual de requerimientos core (paso 3.1) , tal como OOHDM establece, utilizando diagramas UML (Sección 2.1.1.2), o base, no es el foco de esta tesis no se discutirá en profundidad las decisiones de diseño para hacer hincapié principalmente en cómo diseñar los requerimientos volátiles utilizando una metodología orientada aspecto con MATA (paso 3.2). Para ejemplificar el procedimiento, se tomará los ejemplos presentados en el Capítulo 1 donde una aplicación Web con características GIS debe implementar un requerimiento volátil. Posteriormente, se discutirá cómo implementar los conceptos AOP utilizados en el paso 3.2 para que puedan ser implementados en una plataforma OO (sin soporte AOP); paso 3.3.

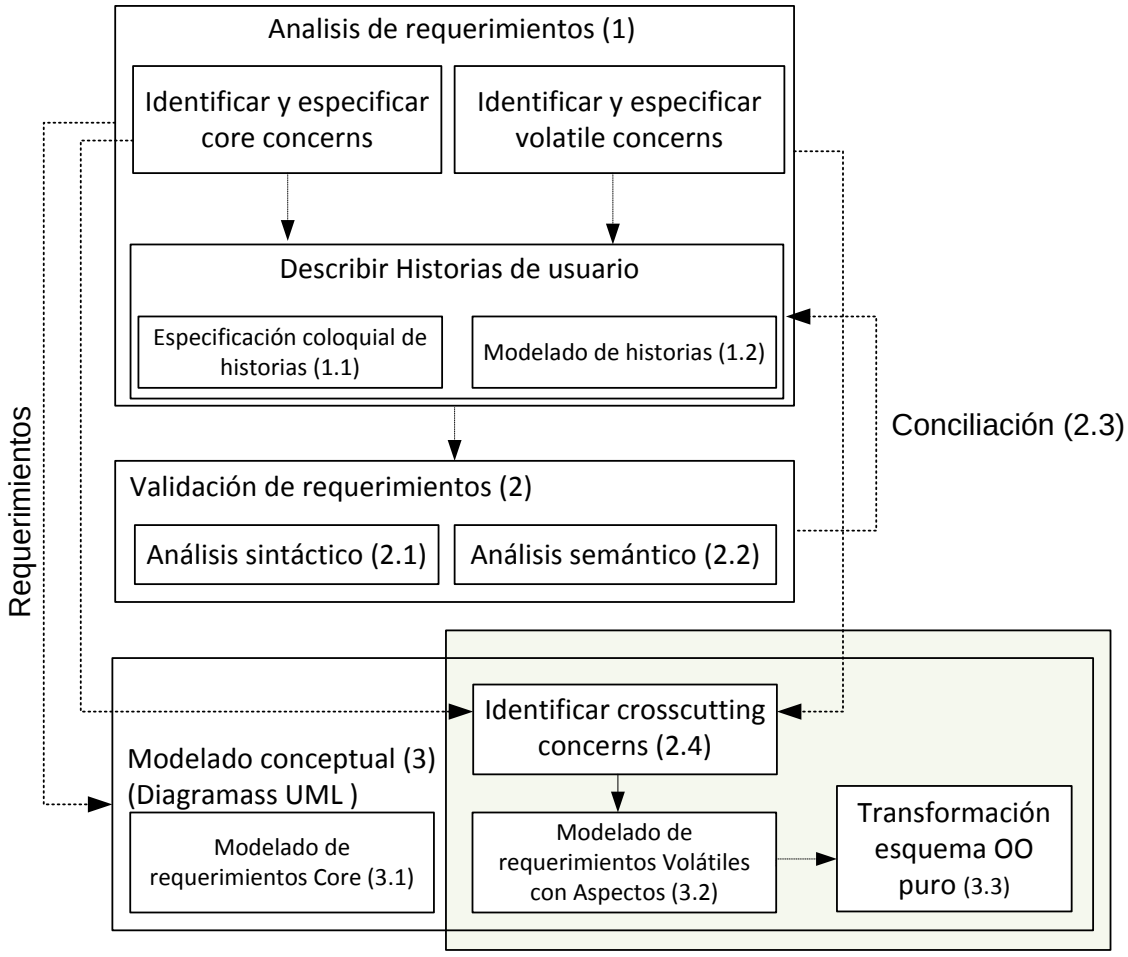


Ilustración 45 Tareas de la metodología especificas de la fase de Modelado Conceptual

7.1 Requerimientos del ejemplo guía

En este caso se utilizara el ejemplo introducido en la Sección 1.1.2. Este es un subsistema del sitio de comercio electrónico para la venta de productos online que hemos utilizado hasta el momento. Este permite gestionar el envió de productos adquiridos por los cliente a domicilio. A continuación se presentan algunos de sus requerimientos. Para mantener el ejemplo simple no se realizarán los casos de usos, historias de usuario o diagramas WebSpec; ya que estos fueron discutidos en el Capítulo 6 . Por lo tanto, solo se hará hincapié en la lista de requerimientos.

Se puede observar en las tablas Tabla 2 y Tabla 3 dos conjuntos de requerimientos, agrupados por *concern*, core y volátiles respectivamente. Los requerimientos son volátiles debido a que éstos se encuentran en un período de prueba (los denominados hasta el momento como Beta) y no se conoce si serán finalmente adoptados. Es importante destacar que los concern volátiles no necesariamente deben ser disjuntos con respecto al concern core; es decir, podría ocurrir el caso

en el que un requerimiento volátil pertenezca al concern core. Por ejemplo, si se desea agregar un código QR o de barras a la etiqueta generada por el requerimiento R2, el requerimiento sería parte del mismo concern *Delivery*.

Tabla 2 Requerimientos core

Concern	Requerimientos	Descripción
Delivery	R1	Un empleado podrá registrar un pedido de servicio de envió de productos.
	R2	Cada paquete, una etiqueta con el identificador univoco (ID) es asignado.
	R3	Utilizando el identificador de paquete, un usuario podrá conocer el estado actual del servicio de envío contratado.
Route Planning	R4	Los clientes podrán visualizar las oficinas de la empresa en un mapa.
	R5	Una hoja de ruta es elaborada para el chofer de la camioneta o el ciclista, que describe la secuencia de calles que ella/el deben seguir para la entrega.
	R6	El chofer de la camioneta podrá visualizar la hoja de ruta como una secuencia de calles en un mapa.
Office	R7	Nuevas oficinas podrán ser registradas en el sistema.
	R8	La información de las oficinas podrá ser editada.
Transportation	R9	El sistema gestiona la información de los vehículos utilizados en el transporte de los paquetes tal como camionetas y bicicletas.

Tabla 3 Requerimientos volátiles

Concern	Requerimiento	Descripción
Delivery tracking	R10	Un cliente utilizando el ID de traza emitido para un paquete podrá consultar la ubicación de ésta última en tiempo real.
	R11	La ubicación del paquete es visualizada en el tablero de control de la compañía.
Blocked streets	R12	Para optimizar el servicio, el cálculo de la hoja de ruta debe tener en cuenta las calles bloqueadas para evitar planes obsoletos u erróneos.
	R13	Las interfaces de usuario y el reporte de hora de ruta (impresión) debe describir explícitamente y sin ambigüedad las calles bloqueadas.
Commentable	R14	Los empleados podrán agregar comentarios a los destinos sobre los mapas para generar y compartir conocimiento.
	R15	Los empleados deben poseer herramientas de edición para editar los comentarios.

City transportation policy	R16	Dependiendo del vehículo utilizado, camioneta o bicicleta, para el envío se debe ajustar la hoja de ruta debido a las restricciones de cada camino disponible (por ej. En rutas no podrán transitar bicicletas y en peatonales no podrán transitar camionetas). Por lo tanto, la hoja de ruta debe tomar en cuenta estas características del camino para determinar el mejor trazado.
	R17	La interfaz de usuario y el reporte generado debe describir en detalle las restricciones de las rutas para dejar en claro al empleado que realiza la entrega cuáles son las calles inhabilitadas y porqué.

7.2 Identificar crosscutting concerns

Como resultado de los pasos anteriores establecidos por la metodología, se dispone de casos de uso y su diagramas WebSpec para los requerimientos donde ellos han sido validado en pos de mantener la consistencia de la aplicación. Como hemos adelantado anteriormente, esta metodología incorpora el método de *Early Aspects* para la detección temprana de *crosscutting concerns* aprovechando sus ventajas discutidas en la Sección 2.2.5.4. Para ello, previamente al diseño del modelo conceptual, se ha incluido una variante del proceso de análisis de *crosscuting concerns* presentado en [87]. En el proceso propuesto en esta tesis, se detecta posibles *crosscutting* estudiando ocurrencias de *concern* en cada caso de uso o historias de usuarios, y se construye una matriz que describe las relaciones *crosscutting* [88] entre los casos de uso y concerns definidos en la etapa de relevamiento de requerimientos.

La diferencia entre la metodología actual y la propuesta en [87], es que, en primer lugar, la propuesta actual utiliza WebSpec como meta-modelo de requerimientos de interacción mientras que en [87] se utiliza UID. En segundo lugar, la actual propuesta, en lugar de utilizar las interacciones (elementos análogos a los *Navigational Units* de los UID), solo referencia a los diagramas WebSpec sirviendo como una herramienta conceptual que asiste a los analistas en el proceso de detección y análisis de comportamientos que pueden ser encapsulados en aspectos.

		Casos de Uso				
		Caso de uso$_1$	Caso de uso$_2$	....	Caso de uso$J$	
Concern	Concern$_1$	v_{11}	v_{21}	…	v_{i1}	C | S
	Concern$_2$	v_{12}	v_{22}	…	v_{j2}	C | S
	Concern$_3$	v_{13}	v_{23}	…	v_{j3}	C | S
	…	…	…	…	…	C | S
	Concern$_i$	v_{1i}	v_{2i}	…	v_{ji}	C | S
Cuenta de concern por Caso de Uso		$\sum_{k=0}^{i} v_{1k}$	$\sum_{k=0}^{i} v_{2k}$	$\sum_{k=0}^{i} v_{...}$	$\sum_{k=0}^{i} v_{jk}$	

Tabla 4 Matríz de crosscutting

Por cada caso de uso o historia de usuario, el analista debe identificar la presencia de requerimientos que causan comportamiento *tangled* asignando un valor (0 o 1) en la celda *joinpoint* como se muestra en la Tabla 4. La celda *joinpoint* v_{ji} especifica que el $concern_i$ afecta al caso de uso_j. El comportamiento *tangling* (ver Sección 2.2.4.2) es detectado cuando la suma de los pesos es mayor a uno. Por otro lado, cuando un *concern* está presente en varios casos de uso, es probable que estemos presenciando un comportamiento *scattered* (ver Sección 2.2.4.2). Esta información es marcada con una S (para el caso de *scattered*) y una C (para el caso de Core), como se puede ver en la última columna de la Tabla 4.

El número de *concern* que cruzan un *caso de uso* dado es una métrica que podemos utilizar para ayudar a detectar artefactos de Software que pueden ser modelados como una jerarquía de aspectos. En esta jerarquía, un aspecto abstracto define los elementos comunes tal como la definición de pointcut y comportamiento compartido (variables de instancias y métodos) por todos los *crosscutting concerns* y cada especificación de aspecto encapsula comportamiento específico en su *advice*. Por ejemplo, supongamos que dos *concern* diferentes tal como "Calles bloqueadas por manifestaciones" y "Soporte de líneas de ferri", dado que ambos *concern* alteran el algoritmo de búsqueda en una aplicación Web GIS para reducir el conjunto de calles disponibles y agregar rutas navegables por los ferries respectivamente. En ambos casos, se incorpora comportamiento en el mismo punto de la aplicación donde se realiza el cómputo de rutas disponibles adyacentes a un punto dado ya que se deben filtrar caminos bloqueados por manifestantes o agregar rutas de feries en base a los horarios de actividad. Cuando este escenario es detectado de forma temprana, los diseñadores van a poder intentar modelar un aspecto abstracto que defina un *pointcut* común (en este caso, el lugar específico del objeto Nodo donde el *advice* debe ejecutarse), rutinas compartidas, y extender el aspecto abstracto con especializaciones conteniendo comportamiento especifico para aumentar el conjunto de rutas disponibles para el caso del concern de "Soporte líneas de Ferri" y reducir el conjunto de calles disponibles para el caso del concern "Calles bloqueadas por manifestaciones".

Por cada caso de uso, es posible detectar requerimientos que causan comportamiento *scattered*. En la Tabla 5, Delivery Tracking está presente en "Query package status" (R3) y en "View transportation information" (R9), significando que los dos últimos son alterados por la misma nueva lógica. Es importante destacar que la Tabla 5 solo muestra un conjunto reducido de casos de uso y c*oncerns* para facilitar la legibilidad de la tabla.

		Casos de uso					
		Request delivery route plan	Delivery Request	Query package status	Fill delivery papers	View transportation information	
Concern	Delivery	1	0	0	0	0	C
	Route planning	0	1	0	0	0	C
	Delivery tracking	0	0	1	0	1	S
	Blocked streets	1	0	1	0	0	S
	Cuenta de concern por Caso de Uso	*2*	*1*	*2*	*0*	*1*	

Tabla 5 Análisis de concern por caso de uso

Los valores obtenidos en la tabla serán utilizados en la Sección 7.4.2 para modelar generalizaciones de aspectos complejas.

7.3 Modelado de funcionalidad Core Orientado a Objetos

En esta sección se describe el paso 3.1 del metodología general, los requerimientos core son modelados utilizando diagramas de clase y secuencia UML. En la Ilustración 46 se presenta, a la derecha, un diagrama de clase correspondiente a las principales entidades de negocio (*Client*, *Package* y *Employee*) y, a la izquierda, las clases correspondientes los *concern* de "Delivery" y "Route Planning". Este último define clases llamadas *Resolver* que encapsulan la lógica necesaria para resolver la hoja de ruta (utilizando el patrón *Strategy*[26]) y la clase *RoadSegmentNode* que define la información de un segmento de calle (nombre de calle, ciudad, etc.) y el rango que cubre (por ejemplo, direcciones entre número 100 y 150). En esta sección no se discutirán las decisiones de diseño debido a que no es el foco de interés de la tesis.

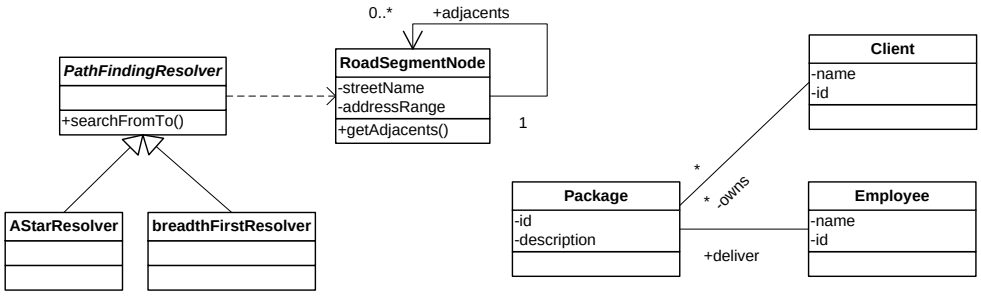


Ilustración 46 Modelo conceptual para requerimientos base

La funcionalidad de la búsqueda de caminos es descripta en el diagrama de secuencia de la Ilustración 47. Los diagramas de secuencia UML serán utilizados para completar las características estructurales de los diagramas de clase con características de comportamiento de los objetos definidos en éste. En este diagrama se mostrará excepcionalmente objetos

correspondientes a la interfaz de usuario solo para mostrar cómo se comportan los objetos del Modelo Conceptual diseñados en esta Sección.

El requisito de una hoja de ruta pasa por la capa de la interfaz Web (debido a R6) hasta llegar al objeto *PathFindingResolver* que colabora con un conjunto de objetos *RoadSegmentNodes* para calcular la ruta esperada. Los objetos correspondientes a la capa navegacional (*RoutePlannerController*) y presentación (*PlannerView*) fueron incorporados para enriquecer el ejemplo.

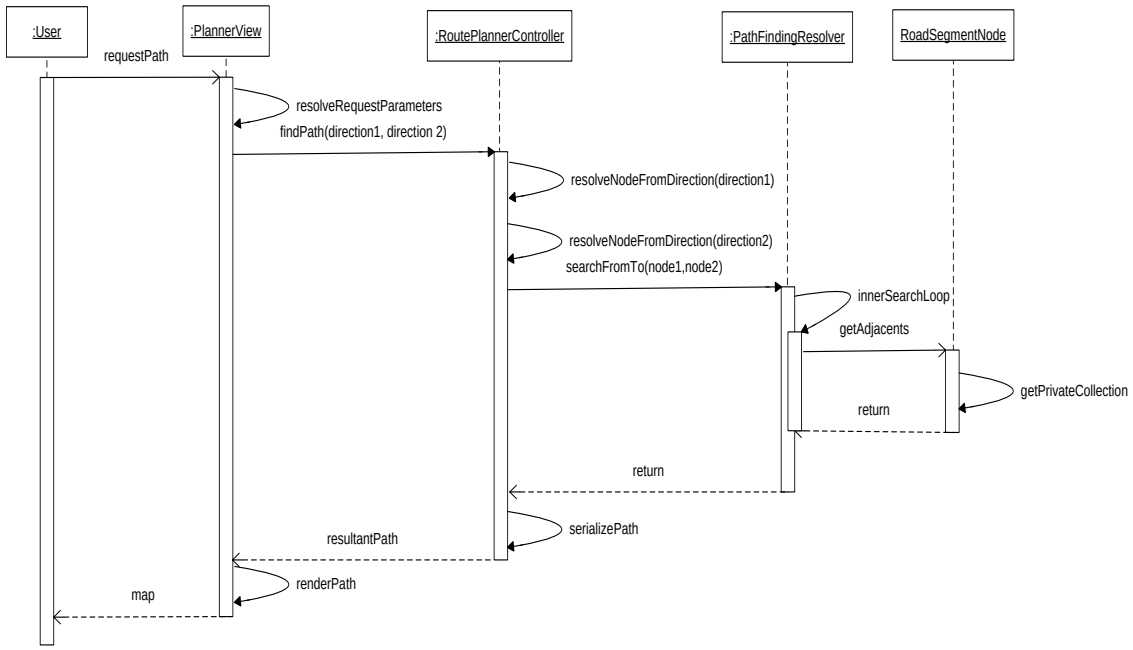


Ilustración 47 Diagrama de secuencia de la funcionalidad de búsqueda de caminos

Por simplicidad las Ilustración 46 y Ilustración 47 especifican solo clases y comportamientos relevantes, que van a ser discutidos en profundidad en las próximas secciones.

7.4 Modelado conceptual Orientado a Aspectos

En la Sección 7.3 se indicó cómo modelar requerimientos core utilizando diagramas UML (clase y secuencia). Estos diagramas resultan convenientes en un escenario estable, sin embargo, en el tipo escenario que proponen las funcionalidades volátiles, estos diagramas por si solos no permiten diseñar funcionalidades sin requerir modificar los modelos conceptuales de la funcionalidad core; es decir, de forma no intrusiva. En esta sección, se discutirá como diseñar modelos conceptuales sin modificar los modelos core utilizando Orientación a Aspectos.

7.4.1 Modelado de funcionalidad volátil con MATA

Como principal requerimiento del *concern* de *calles bloqueadas* (*Blocked Streets*) establece que el cálculo de hoja de ruta debe tener en cuenta las calles inaccesibles, de ahora en adelante *calles bloqueadas*, para evitar informar al conductor rutas erróneas. El cálculo de hoja de ruta está basado en un algoritmo de búsqueda de caminos [15] que trabaja sobre un grafo, donde los nodos representan la red de calles. El algoritmo visita cada nodo en el grafo preguntando por sus nodos adyacentes los cuales son utilizados para obtener la mejor opción; en este caso el camino más corto.

El aumento o reducción del conjunto de nodos adyacentes a un nodo dado puede ser realizado por medio de la encapsulación de este comportamiento especial con las técnicas orientadas a aspectos. En este caso en particular, un aspecto es definido para modularizar el aumento o reducción del conjunto de objetos que son procesados, nodos del grafo de calles, por el algoritmo de búsqueda como se puede ver en la Ilustración 48. Un actor realiza la solicitud de la lista de nodos adyacentes, el cual es anotado con el estereotipo MATA *context* (“<<context>>”), funcionando como *pointcut* del aspecto. En el advice del aspecto, los nodos adyacentes son computados invocando el mensaje original *getAdjacents* (retornando todos los nodos adyacentes) de forma implícita mediante el operador “<<Any>>”; este operador permite cualquier secuencia de mensajes del modelo base (Ilustración 47). Luego, las *calles bloqueadas* son filtradas utilizando el método *filterBlockedStreets* introducido por el aspecto. Este método utiliza el servicio de calles cortadas (clase *BlockeStreetService* incorporada por el concern volátil), el cual determina si un segmento dado está bloqueado utilizando el método *isBlocked*. Finalmente, la hoja de ruta es retornada al actor que la requirió sin que éste haya notado la presencia del aspecto.

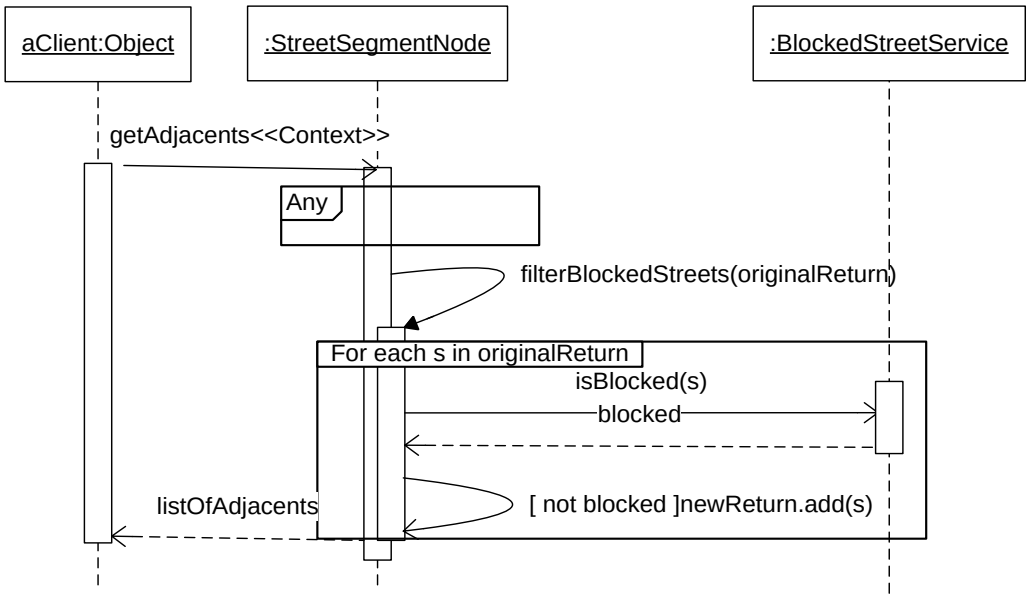


Ilustración 48 Diseño del aspecto que filtra calles cortadas

7.4.2 Diseñando generalizaciones de aspectos

Como fue descripto en Tabla 5, varios *concern* pueden cruzar el mismo caso de uso generando código *tangled*. Por ejemplo, caso de uso "Request delivery route plan" es afectado por algunos requisitos del *concern* "Blocked streets". Además, un nuevo *concern* "City transportation policies" (no presente en la tabla) también cruza el caso de uso. Este último permite restringir rutas de transporte utilizadas para el transporte del paquete; por ejemplo, bicicletas pueden ser utilizadas para envíos en zonas céntricas de la ciudad debido a que es fácil transitar calles angostas mientras tanto para largas distancias se debe priorizar el uso de camionetas. El *concern* "City transportation policies" va a tener un patrón de solución similar al diseñado para "Blocked streets" ilustrado en la Ilustración 48 . Es importante notar que el diseñador puede tomar ventaja de este hecho para producir un diseño de aspecto mejor y con mayor reusabilidad. La definición del *pointcut* para el filtrado de las adyacencias de un nodo, y eventualmente el *advice*, pueden ser generalizados en un aspecto abstracto. Adicionalmente, el aspecto abstracto puede también definir e introducir métodos compartidos por aspectos concretos. Un diseño más avanzado puede ser obtenido modelando los elementos *advice* de un aspecto como un patrón *Template Method* [26] para especificar la lógica compartida de filtrado. En la Ilustración 50, un aspecto abstracto llamado *AbstractFilteringAspect* es modelado implementando el patrón *template method* definiendo en este el comportamiento compartido para el filtrado de nodos. La especificación de

comportamiento es alcanzada implementando el método *processItem*. En la Ilustración 49 se muestra como diferentes aspectos reutilizan la lógica del aspecto abstracto.

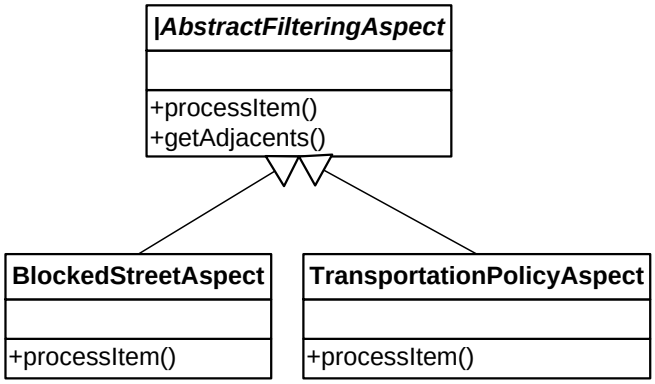


Ilustración 49 Jerarquía de aspectos para el filtrado de adyacencias

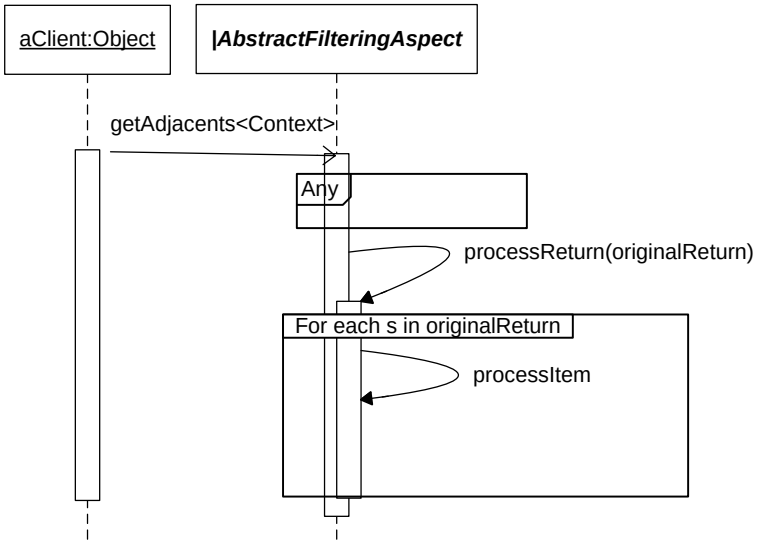


Ilustración 50 Advice abstracto para manipulación de adyacentes

7.4.3 Implementación AOP de diagramas MATA y PS

Esta sección presenta cómo implementar el ejemplo de *concern* de *calles bloqueadas* modelado en la Sección 7.4.1.

Los diagramas MATA y PS descriptos a lo largo de la Sección 7.4 puede ser implementado en cualquier plataforma que soporte conceptos AOP ya que su mapeo es directo en aquellas plataformas que provean construcciones de *pointcut*, *advice* y *joinpoint*. Continuando con el ejemplo correspondiente a la aplicación Web que permite la búsqueda de caminos utilizando el algoritmo A* (descripto inicialmente en la Sección 1.1.2 y diseñado a lo largo de este capítulo). Este algoritmo resuelve el problema del camino menos costoso desde un punto dado a un destino.

La funcionalidad base de búsqueda de caminos se implementó utilizando la librería PyRoute [89], el cual consume información de un servidor de cartografía llamado OpenStreetMap [90]. Este servidor es una implementación de código abierto donde su grupo de desarrollo mantiene además una vasta cartografía mundial; que además es abierta.

Siguiendo los pasos establecidos por la metodología (definidos en el Capítulo 5), los modelos aspectuales descriptos anteriormente deberán ser implementados utilizando un lenguaje orientado a aspectos.

La correcta implementación del requerimiento volátil R12 requiere la implementación del aspecto que introduce la noción de calles bloqueadas en las búsquedas de ruta y el controlador que maneja la presentación de estos nuevos conceptos de forma adecuada. El modelado e implementación de las clases necesarias para la adecuada presentación de los conceptos de calle bloqueada al usuario será discutido en las secciones siguientes.

El algoritmo de búsqueda de caminos funciona sobre un grafo dirigido [15] donde cada *nodo* es visitado para alcanzar un *nodo* destino. Cuando el destino no es alcanzado, el algoritmo visita los *nodos* adyacentes del *nodo* que visita en ese momento repitiendo el proceso una y otra vez hasta que el destino es alcanzado. En este contexto, el aspecto que encapsula el *concern* de “Blocked streets”, agrega el comportamiento necesario para filtrar calles bloqueadas posteriormente a la búsqueda de *nodos* adyacentes realizada por el algoritmo original; es decir se delega al algoritmo original para buscar adyacentes y posteriormente se retiran aquellos *bloqueados*. Esta solución es alcanzada decorando el mensaje *#getAdjacents* de la clase *RoadSegmentNode* (retorna todos los *nodos* adyacentes del *nodo* actual). En consecuencia, cuando el algoritmo resuelve los adjacentes para en algún *nodo* particular, éste controla si el próximo *nodo* corresponde a una calle bloqueada una vez que el algoritmo original resolvió todos los adyacentes. Para esto, el aspecto que encapsula partes de los requisitos del concern “Blocked street” (ya que no encapsula las características de presentación) construye una colección que incluye todos los posibles *nodos* destino (utilizando el algoritmo original #getAdjacents) que no han sido aun caminados para alcanzar el destino, cuando la colección de *nodos* adyacentes es retornado por el algoritmo original, el aspecto filtra los *nodos* bloqueado utilizando un servicio de calles bloqueadas (denominado *blokedService* incorporado por el concern volátil para identificar calles cortadas). Luego el aspecto que decora el comportamiento original, retorna la nueva colección y el algoritmo disparador de la búsqueda continua normalmente.

```
(1)class AdjacentsAs(Aspect):
(2)     def __init__(self):
(3)         self.blockedService = resolveBlockedStreetService()
(4)     def atReturn(self, cd):
(5)         originalReturn = cd.returned
(6)         newReturn = []
(7)         for el in originalReturn:
(8)             if not (self.blockedService.isBlocked(el[0])):
(9)                 newReturn.append(el)
(10)        cd.change()
(11)        return(newReturn)
```

Código 1 Aspecto que introduce el concepto de una calle cortada en un algoritmo A*

El artefacto que implementa la funcionalidad volátil es un aspecto implementado utilizando el lenguaje de programación Python. Este encapsula la lógica que filtra las calles bloqueadas y puede ser apreciado en el Código 1. En la línea 3 se resuelve el servicio encargado de determinar sí un *nodo* del grafo corresponde a una calle bloqueada. Las líneas 4-11 definen el cuerpo, o *advice*, que deberá ser ejecutado ante cada invocación del mensaje *#getAdjacents*. En la línea 7-9 se filtra la colección retornada, delegando al *servicio de calles bloqueadas* la identificación de una calle bloqueada por cada elemento de la colección. La línea 10 utiliza el mensaje *#change* para alterar el mensaje original retornado por la captura del mensaje en el *pointcut*. Finalmente, la línea 11 retorna una nueva colección sin los segmentos de calles cortadas. Este aspecto has introducido exitosamente los conceptos de calle bloqueada en el algoritmo, de forma transparentes y sin acople; ya que el algoritmo original no fue modificado.

7.5 Solución Orientado a Objetos sin soporte de aspectos

Muchas veces la plataforma destino no fue concebida teniendo en cuenta conceptos orientación a aspectos necesarios para implementar de forma directa los modelos diseñados en las secciones anteriores; es decir, conceptos tal como *pointcut* y *advice*. Por ende, esta brecha tecnológica debe ser cubierta con conceptos provistos por la programación orientada a objetos.

Para cubrir esta brecha conceptual por la ausencia de conceptos AOP, como paso posterior al diseño de una funcionalidad volátil (utilizando UML y MATA en el paso 3.2 de la metodología) se sugiere la migración de estos conceptos desarrollados en términos de patrones *Commands* [26] y *Decorators* [26] para las nuevas operaciones o modificaciones de comportamiento, y una técnica definida en esta tesis llamada *Inversión de relación* para implementar las relaciones volátiles. La lógica introducida en uno o más objetos por una funcionalidad volátil es encapsulada en un *Command* porque éste simboliza un comportamiento de aplicación en una clase; en lugar de un método. Este puede también ser considerado como un *Decorator* debido a que este permite agregar nuevas características (propiedades y comportamientos) a una aplicación de una forma no intrusiva; es decir, sin necesidad de modificar la clase decorada.

La misma solución puede ser aplicada cuando un proceso de negocio es modificado alterándolo en groso modo; cuando, por ejemplo, una nueva actividad o tarea es agregado al mismo. Es

posible en OOHDM ya que las actividades son modeladas como objetos de hecho en los esquemas conceptuales y navegacionales (ver[91]).

7.5.1 Reglas de transformación de modelos AOP a modelos OOP

A continuación se discutirá brevemente las reglas de regresión del nivel de abstracción de modelos AOP a modelos POO. Estas reglas o heurísticas tomarán modelos elaborados con conceptos AOP y los llevarán a un modelo OOP puro.

7.5.1.1 Nuevas Clases

Las nuevas clases no son un problema ya que no alteran el modelo core. Estas clases se mantienen tal como fueron definidas y son agrupadas en el paquete correspondiente a la funcionalidad volátil. En la Ilustración 51, se puede apreciar el paquete volátil introducido ante la necesidad de gestionar temporalmente y sin un periodo definido, Videos multimedia en la aplicación Web.

A la izquierda el Modelo Conceptual core donde un *CD* conoce a todos sus *Autores*, y, a la derecha, una nueva clase volátil que modela las características de un Video (nombre, descripción y longitud) y define el comportamiento de reproducción habitual (método *play*).

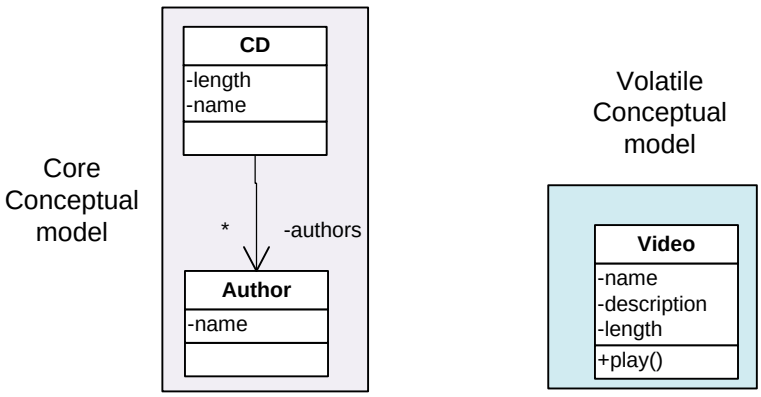


Ilustración 51Clase volátil nueva

7.5.1.2 Nuevo comportamiento

El nuevo comportamiento es modelados como objetos principales en el modelo conceptual volátil; ellos son considerados como una combinación de *Commands* y *Decorators* [26] que colaboran con clases del modelo conceptual core. Esta estrategia aplica también a las pequeñas variantes de reglas de negocio (tal como la adición de un descuento de presión hasta un producto). En este caso, la utilización de un *Decorator* es frecuente cuando el cambio es a nivel de método que a nivel de clases. Notar que esta estrategia puede ser aplicada a varios métodos en una clase correspondiente a un modelo orientado a objetos. En métodos que están basados en constructores

para el modelado de datos, tal como WebML, agregar nueva funcionalidad (volátil) es simple dado que se utiliza un lenguaje para tipo de entidades. Alternamente, tal como fue discutido en el capítulo de trabajos relacionados, existe una extensión de WebML que permite introducir conceptos de la orientación a orientada a aspectos [36].

Supongamos que se requiere implementar una funcionalidad volátil, esta funcionalidad introduce una operación y un atributo volátil a un objeto de negocio, en este caso un CD, tal como se puede apreciar del la Ilustración 52. El rol "|CD" especifica la incorporación del atributo "aVolatileAttribute" y el método "aVolatileOperation".

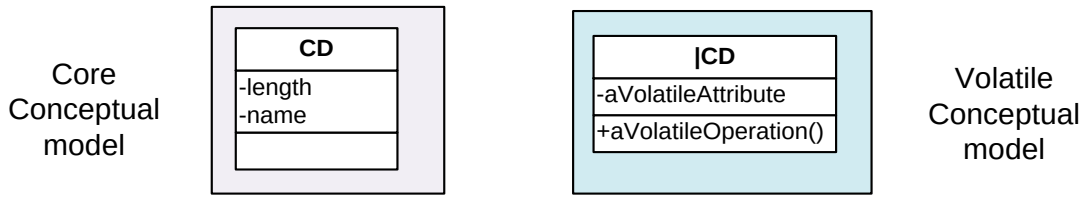


Ilustración 52 Modelo Conceptual AO utilizando MATA para incorporar un método volátil

En el caso de que la plataforma subyacente no soporte de forma nativas conceptos AOP, a continuación se discutirá como utilizar patrones de diseño de POO utilizando los patrones de diseño *Decorator* y *Command*.

7.5.1.2.1 Uso del Patrón Decorador para nuevo comportamiento

El patrón *Decorador (Decorator)* [26] puede ser utilizado para extender (decorar) la funcionalidad en tiempo de ejecución de una instancia específica de una clase sin tener en cuenta todas las instancias de una clase. Un objeto *decorador* (nombre del rol en el patrón) envuelve otro objeto para hacer de intermediario entre este último y los objetos con el que éste colaboraba. De esta forma, el objeto *decorator*, captura aquellas invocaciones que les son de interés para agregar el comportamiento que fue indicado en el diseño (que puede tener en cuenta el comportamiento original) y delega aquellas que no le son de interés al objeto decorado. En [26] se podrá encontrar una discusión exhaustiva de las ventajas y desventajas del patrón así como también las diferentes alternativas de diseño para su implementación.

En la Ilustración 53 se puede apreciar la implementación de la solución al diseño de la Ilustración 52 donde se utilizó el patrón *Decorator*. Los pasos requeridos para la traducción del modelo AOP al modelo OOP son:

1. En el caso de que no exista ninguna otra funcionalidad volátil sobre una *Clase*, subclasificar la clase destinataria del nuevo método (o decoración de alguno método existente). Se sugiere nombrar a esta nueva clase con el nombre de la superclase teniendo

como postfijo la palabra "Decorator". Por ejemplo, en la ilustración, la clase *CD* fue subclasificada en *CDDecorator*.

2. La clase *Decorator* debe poseer una variable de instancia (*component*) a la clase decorada y debe implementar cada mensaje de la clase decorada de tal forma que las invocaciones sean delegada a la instancia decorada utilizando la variable anteriormente definida. Además, deberá definir un mensaje constructor que tome como parámetro la clase decorada para su correcta inicialización.

3. Extender la Clase *Decorator* para implementar el decorador concreto a la funcionalidad volátil. En esta clase se deberán implementar aquellos nuevos mensajes incorporados por la funcionalidad volátil o reescribir los mensajes que posean nuevos comportamientos. Por ejemplo, *CDDecoratorVolatileOperation*. La jerarquía puede ser evitada implementando el nuevo comportamiento como un *Proxy* en Smalltalk (ver la explicación del patrón en [26] para más detalle).

En la Ilustración 53 se puede apreciar el resultado de aplicar la técnica anterior dando como resultado un paquete volátil que incluye la clase *CDDecorator* y *CDDecoratorVolatileOperation*. La primer clase es la definición Abstracta del decorador definiendo los recursos compartidos por todos los posibles decoradores (referencia al objeto decorado y métodos delegadores), y la segunda clase corresponde al decorador concreto que agrega la nueva operación *aVolatileOperation* al objeto decorado.

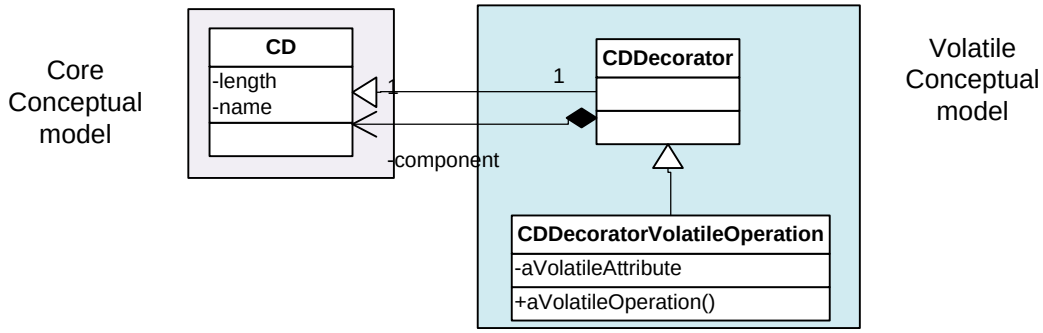


Ilustración 53 Modelo Conceptual OO utilizando un patrón Decorator para implementar una operación volátil

7.5.1.3 Uso del Patrón Comando para nuevo comportamiento

El patrón Command[26] es un patrón de diseño en la que un objeto es utilizado para representar y encapsular toda la información necesaria para realizar una operación. Las clases que implementan este patrón, exponen un método con el que se ejecuta la lógica requerida.

Este patón puede ser utilizado para encapsular la lógica correspondiente a la funcionalidad volátil. Para ello, a continuación se presentan un conjunto de pasos que permiten transformar el modelo AOP en un modelo OOP. Los pasos son:

1. Por cada nueva operación definida en el rol del diagrama MATA para un objeto de negocio, en el ejemplo *aVolatileOperation* definido por el rol *"|CD"*, se define una clase que implementa el patrón *Command*; por ejemplo, *VolatileOperationCommand*. Es decir, implementa un método, por ejemplo *perform*, que encapsula la lógica volátil.

2. La clase que implementa el patrón *Command*, deberá poseer las variables de instancia y conocimiento necesarias para ejecutar la lógica de la funcionalidad volátil. Por ejemplo, en Ilustración 54, existe una relación *parameter*, utilizado para manipular el objeto CD. Esta variable de instancia reemplaza la relación de conocimiento por defecto que se establece entre un método y la clase que lo implementa (por ejemplo, en Smaltalk, es *self* y, en Java, *this*).

En Ilustración 54 se puede apreciar el resultado de implementar la nueva funcionalidad definida en la Ilustración 52 utilizando el patrón *Command*.

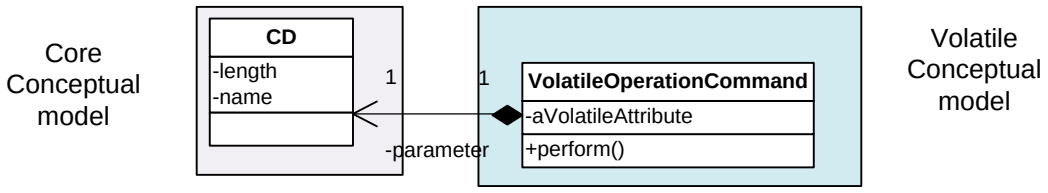


Ilustración 54 Diseño conceptual OO utilizando un patrón Command para implementar una operación volátil

7.5.1.4 Inversión de la relación

La *inversión de la relación* es utilizada para eliminar el conocimiento por parte de los componentes core de los elementos de la funcionalidad volátil. En lugar de hacer a las clases conceptuales core consientes de las nuevas funcionalidades, la relación de conocimiento es invertida. Nuevas clases conocen las clases base (core) sobre las cuales ellas están definidas. Las clases core, en consecuencia, no tienen conocimiento sobre ninguna funcionalidad volátil. Por ejemplo, si se desea relacionar un *Producto* con sus *Compradores* en un sitio de comercio electrónico donde el *Producto* es una clase Core y un comprador es una clase *volátil*, se define una relación entre *Comprador* y *Producto* con base en Comprador llamado *producto adquirido*.

Supongamos el caso donde un nuevo requerimiento volátil requiere la incorporación de videos promocionales de ciertas pistas de un CD de música. En este caso, una relación debe ser establecida entre el objeto del modelo core CD y el objeto del *concern* volátil *Video*; este último modela el video comprendiendo nombre, descripción y longitud del mismo entre otras

propiedades. En la Ilustración 55 se puede apreciar el modelo aspectual utilizando MATA donde existe un role para el componente CD y una relación en rojo llamada "hasVideo".

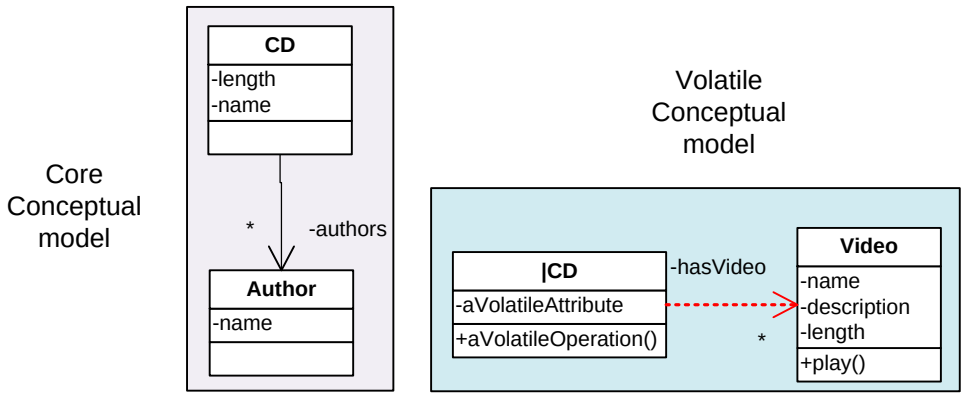


Ilustración 55 Modelo Conceptual OA de la funcionalidad volátil *Video*

Para resolver éste problema se deben analizar los distintos tipos de relaciones en las que la entidad core es propietaria ya que en las relaciones opuestas no producen problema alguno. Los tipos a analizar son:

Muchos a muchos: en este caso, arbitrariamente, el rol de la relación de conocimiento de la clase core para que sólo quede el conocimiento desde la clase volátil a la core. Toda la lógica correspondiente a esta relación deberá ser ajustada en base a consultas (*queries*) sobre objetos volátiles. Por ejemplo, para conocer los videos (clase *Video*) de un *CD* se deberá consultar todos los objetos de tipo *Video* que conocen un *CD* dado.

Uno a muchos: En este caso, se propone una transformación sobre la nueva relación introducida por la funcionalidad volátil que posee los siguientes pasos:

1. En primer lugar, si la relación es unidireccional, se la debe convertir en bidireccional. Para ello, se aplica el refactoring "Change Unidirectional Association to Bidirectional" [16].

2. Luego, se introduce una clase que representa una relación N-aria entre las dos entidades comprometidas con el mismo nombre de la relación y que posee referencias a las clases comprometidas. Esto se lleva a cabo implementando la variante 2 del patrón "Defining N-ary Relations on the Semantic Web: Use With Individuals"[92]. Por ejemplo, teniendo en cuenta la Ilustración 55, surge la nueva clase *HasVideo* con la referencias a las clases *CD* y *Video*. Cabe destacar que la entidad propietaria de la relación pierde a la misma y ésta pasa a ser propiedad de la Clase que implementa la relación N-aria.

3. Finalmente, si el diseñador lo ve conveniente, se deben componer la clase que representa la relación N-aria con la clase volátil utilizando el refactoring "Inline Class" [16].

En base la transformación anterior, en una plataforma de desarrollo que no brinda soporte de aspectos se deberá utilizar inversión de relación para que la relación natural de un "*CD* conoce sus *Video*" (*hasVideo* en la imagen) deberá ser reemplazada por su equivalente "un *Video* conoce su *CD*" (*videoOf*). En la Ilustración 56 se puede apreciar el resultado de aplicar los pasos 1 al 3 donde la relación invertida es identificada con una línea de puntos.

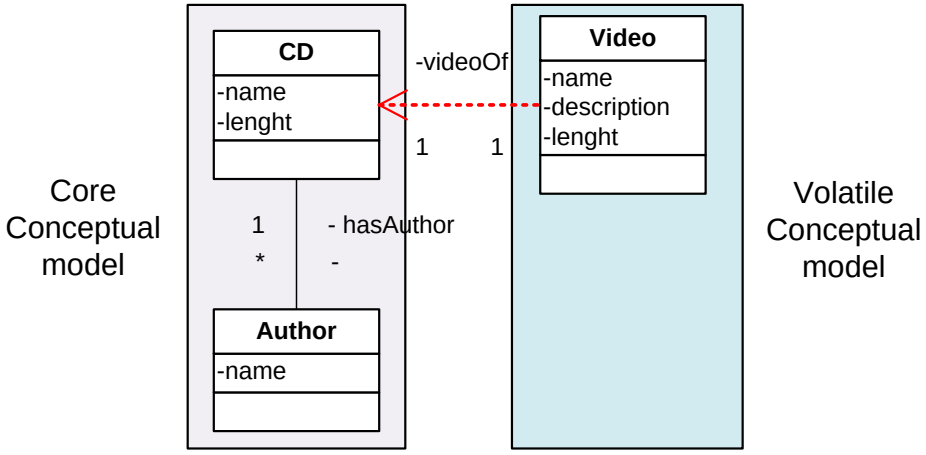


Ilustración 56 Ejemplo de inversión de relación en Modelo Conceptual

7.5.2 Discusión sobre las reglas de transformación

Es importante destacar que en procedimiento discutido en esta sección se está perdiendo los *pointcuts* definidos en los modelos de diagrama de secuencia MATA. La pérdida del control de los *pointcuts* obliga a que la instanciación de los *Commands* y *Decorators* se gestionada de forma ad-hoc. Por experiencia en las pruebas de campo realizadas, la manipulación de objetos conceptuales volátiles es realizada solo en la capa navegacional volátil ya que el modelo core conceptual y navegacional no conocen ni conocerán de la existencia de los artefactos volátiles. En consecuencia, se deberá tomar el trabajo de implementar la gestión de los objetos que implementan la funcionalidad volátil, *Commands* y *Decorators*, en la capa de navegación.

En este caso, se toma la libertad de perder los pointcuts ya que es una penalización de utilizar este descenso de abstracción.

La decisión de la estrategia para implementar nuevos o mejoras de comportamiento, *Commands* y *Decorators*, de la funcionalidad volátil dependerá del diseñador dependiendo del contexto.

Se puede resaltar las siguientes características de cada solución:

- En casos donde se incorpora nueva funcionalidad, se recomienda la utilización del patrón C*ommand* ya que es una solución simple que no requiere definir más clases que el solo comando.

- En caso donde se modifica funcionalidad se recomienda la utilización del patrón *Decorator* ya que permite modificar el comportamiento de la clase decorada sin que los clientes cambien de contrato. Es importante destacar que el desafío de esta solución es gestionar de forma ad-hoc cómo instanciar estos decoradores. En Smalltalk las referencias hacia un objeto core pueden ser fácilmente reemplazables por su decorador mediante el mensaje *#become* [93].

Capítulo 8 Modelo navegacional

A continuación se discutirá cómo diseñar los cambios navegacionales que produce una funcionalidad volátil. En Ilustración 57 se muestran las tareas específicas correspondientes al diseño del Modelo Navegacional de funcionalidades volátiles. A partir de los Modelos conceptuales obtenidos en la etapa "Modelado conceptual" (3) se realizan los Modelos Navegacionales (4). La tarea de Modelado Navegacional Core corresponde a la tarea indicada por OOHDM y ya discutida en la Sección 2.1.1.3. En este capítulo, se presentará dos tareas de la metodología propuesta en esta tesis necesarias diseñar las características navegacionales introducidas por una funcionalidad volátil.

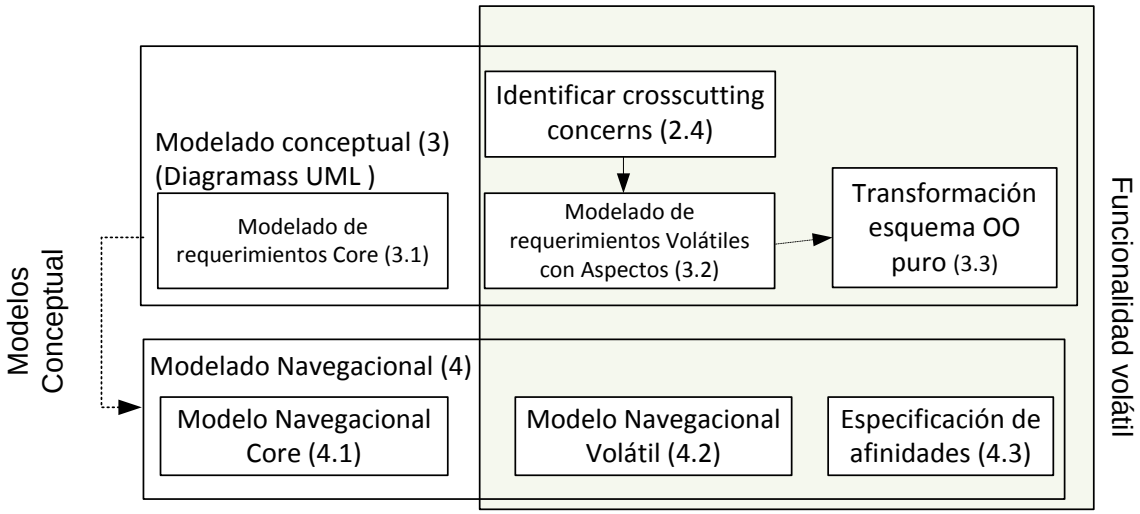


Ilustración 57 Tareas de la metodología especificas de la fase de Modelado Navegacional

El modelo navegacional de aplicaciones Web tiene como objetivo de definir vistas, acceso a estructuras y caminos a contenidos para facilitar su fácil acceso y navegación por los usuarios. La mayoría de los métodos de ingeniería Web basan sus modelos navegacionales en dos primitivas de modelado, llamadas *Nodo* y *Link*. OOHDM no es la excepción a esto, ya que éste define *nodos* como vistas lógicas sobre clases del modelo de la aplicación y *links* como la implementación hipermedial de las asociaciones del modelo de aplicación. Pueden existir *links* que mejoren la navegación de la aplicación sin necesidad de que exista una contra parte en el Modelo Conceptual; por ejemplo, el link que va desde cualquier Nodo Navegacional al Nodo Principal de la aplicación Web.

En el Modelo Navegacional, los Nodos core y volátiles se relacionan utilizando una *afinidad* la cual indica, por ejemplo, si la atributos de un Nodo definidos por la funcionalidad volátil son "insertados" en el nodo core o si ellos están conectadas con un hipervínculo. Como se mostro en la Ilustración 1 del Capítulo 1 se puede apreciar un nuevo link desde un nodo core al nodo navegacional volátil que presenta la información del *concern Volver a la Escuela* (Ilustración 2 del Capítulo 1). En la Ilustración 58, se puede apreciar del lado izquierdo el modelo core con los Nodos de navegación *HomePageNode y ProductNode*. El primero corresponde a la página principal donde existe un índice de productos tal como se puede apreciar en la Ilustración 1. El segundo es la vista del modelo conceptual de un producto y se puede ver su interfaz en la Ilustración 2. Del lado derecho de la ilustración, podemos apreciar el Modelo Navegacional volátil que define un Nodo índice para listar los productos "escolares" y una extensión del Nodo *ProductNode* que le permite a este último gestionar una promoción.

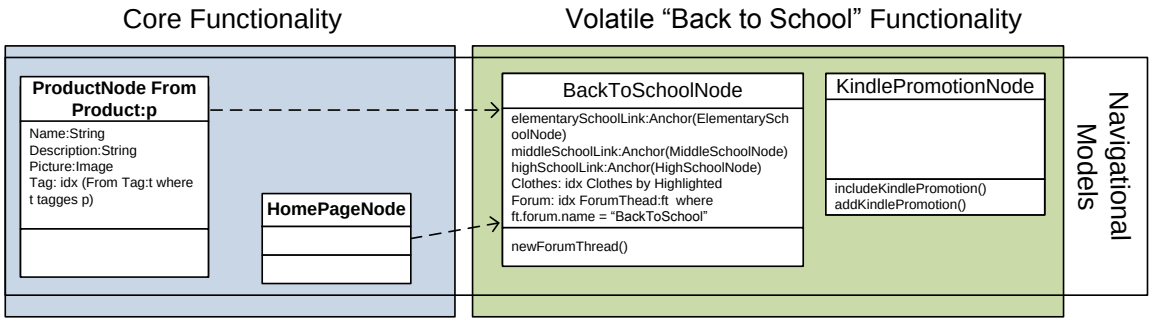


Ilustración 58 Modelo Navegacional Core y Modelo Navegacional volatil de la funcionalidad "Volver a la escuela"

Esta especificación también incluye una consulta indicando cuáles nodo core va a tener la extensión. Esta consulta respeta el lenguaje de consulta (query) OOHDM y determina cuáles Nodos son afectados por la funcionalidad volátil. El nombre fue inspirado desde [94].

Es posible utilizar una o más afinidades para la misma funcionalidad volátil. Esto es, la misma funcionalidad volátil puede ser incorporada en diferentes partes de la aplicación siguiendo diferentes reglas.

8.1 Definición de afinidad

Las afinidades de una funcionalidad volátil son especificadas con el mismo lenguaje de consulta utilizado en OOHDM para definir nodos [17]. El lenguaje está basado en consultas sobre objetos. Utilizando este lenguaje de consulta, la definición de una *afinidad* asume la forma del Código 2.

```
AFFINITY: AffinityName
FROM C1…Ci
WHERE Predicate
INTEGRATION: Extension | Linkage((V1…Vi))
```

Código 2 Definición de *Afinidad*

En el Código 2, el parámetro *AffinityName* corresponde al nombre asociado a la afinidad (*AFFINITY*) donde la secuencia $C_1…C_i$ denota las clases core envueltas en la consulta. El parámetro *Predicate* del campo *WHERE* es la expresión lógica definida en termino de propiedades de un modelo de objetos el cual determina las instancias de los nodos core $C_1…C_i$ que serán afectadas por la funcionalidad volátil. Por otro lado, la estrategia de integración de nodos core y volátiles se indica en el campo *INTEGRATION* utilizando las palabras clave *Extension / Linkage.* Estas indican la forma en la que la funcionalidad volátil es compuesta en los nodos core combinando los nodos volátiles $V_1…V_i$. Una *Extension* indica que los nodos core son mejorados incorporando la nueva información (y operaciones). En una composición de tipo *Linkage*, los nodos core "solo" permiten navegación hacia los nodos volátiles $V_1…V_i$ los cuales realmente contienen la funcionalidades volátiles. En el caso de la integración *Linkage*, se puede también especificar características adicionales tal como atributos o *anchors* que tienen que ser agregados para extender el nodo (por ejemplo cuando se desea mejorar la navegación).

Los requerimientos relacionados con el *concern* volátil de *Volver a la Escuela* hace necesario introducir un nuevo Nodo navegacional llamado *BackToSchoolNode* (como se puede ver en el Modelo Navegacional volátil de la Ilustración 58) el cual actúa como un índice de categorías de productos relacionados a la escuela (ver el resultado en la Ilustración 2). Para que el nuevo nodo volátil sea fácilmente accesible a los usuarios, se necesita definir como mínimo dos *afinidades* del tipo *linkage*: la primera va a ligar la página principal del sitio hacia el índice de catálogo y el segundo va conectar cada nodo de producto etiquetado como "school" al índice. En la Ilustración 1 del Capítulo 1 se puede ver un ejemplo de este nuevo link "School supplies" señalado con una elipse. La especificación de estas dos afinidades pueden verse en el Código 3 y Código 4.

```
AFFINITY: Back to School affinity - Home Page
FROM: HomePageNode
INTEGRATION: Linkage (BackToSchoolNode)
```

Código 3 Definicón de *Afinidad Linkate* desde el Nodo principal al Nodo de "Volver a la escuela"

```
AFFINITY: Back to school affinity - Generic link

FROM: ProductNode

WHERE: hasTag('School')

INTEGRATION: Linkage (BackToSchoolNode)
```

Código 4 Definicón de *Afinidad Linkate* desde el Nodo "escolar" al Nodo de "Volver a la escuela"

En la *Afinidad* del Código 3, la página principal de Amazon.com denominada *HomePageNode* se enriquece con un link hacia el nodo navegacional *BackToSchoolNode* definido en el paquete de la funcionalidad volátil de la Ilustración 58. Mientras que en la *Afinidad* del Código 4, se agrega el link a todas las instancias del nodos que presentan información de productos, denominados *ProductNode*, siempre y cuando estén etiquetados como "escolar" indicado con el predicado *hasTag('school')*.

Por último, para incorporar las características del nodo *ShippingPromotionNode* al nodo *ProductNode* que permitan gestionar una promoción a nivel navegacional se debe especificar una *Afinidad* del tipo *extensión*. Esta afinidad complementa los métodos del nodo *ProductNode* con los métodos del nodo *ShippingPromotionNode* mostrados en la ilustración que serán invocados por el comportamiento volátil de la interfaz. La afinidad que realiza lo descripto puede ser apreciada en Código 5.

```
AFFINITY: Back to school affinity .Shipping Promotion

FROM: ProductNode

WHERE: hasTag('School')

INTEGRATION: Extension(KindlePromotionNode)
```

Código 5 *Afinidad* de extensión para la promoción de envío gratis

8.2 La ejecución de las consultas de Afinifidad

Para mejorar la flexibilidad, las consultas tienen la intención de ser ejecutadas en tiempo de ejecución, es decir cuando la aplicación se encuentra siendo ejecutada. Esto permite soportar extensiones irregulares, por ejemplo funcionalidades volátiles que aplican solo a algunos nodos específicos de una clase, como el ejemplo anterior lo muestra. De esta forma, la composición de funcionalidades volátiles en la capa navegacional no ocurre durante la codificación del modelo, sino durante la ejecución de la aplicación. La ejecución de las consultas y la composición de nodos en el contexto del método OOHDM son soportadas por un framework de aplicación llamado Cazon que será descripto en el Capítulo 11 .

8.3 Volatilidad en procesos de negocios

Un caso particular de volatilidad en modelos navegacional surge cuando una nueva actividad es inyectada en un proceso de negocio. Supongamos, por ejemplo, que el proceso de *Compra* en un

portal Web de comercio electrónico consiste en seis actividades, llamadas "Login", "Consolidar orden", "Confirmar dirección", "Seleccionar forma de pago", "Especificar opciones del envío" y "Confirmación". Supongamos que durante el periodo de Navidad queremos agregar la opción de especificar un "envoltorio personalizado" para mejorar la experiencia del usuario luego de seleccionar las "opciones del envío". En OOHDM, la contraparte navegacional de los objetos actividades son los *activity nodes* [17], en consecuencia, cada actividad en el proceso de *Compra* mencionado anteriormente es asociado a un nodo correspondiente en el modelo navegacional de la aplicación. Como consecuencia, la inyección de la actividad volátil que permite seleccionar el envoltorio consiste en:

1. Definir un nuevo nodo de actividad en el modelo de navegación volátil (además de su correspondiente clase en el modelo conceptual);

2. expresar la relación entre el nuevo nodo de actividad volátil y los otros nodos core de la aplicación;

3. especificar la integración como fue descripta en la Sección 8.1.

Los nodos de actividades OOHDM están enlazados entre ellos por medio de *activity links* donde su semántica es definida por las relaciones desde donde ellos fueron derivados en el modelo conceptual. Por lo tanto, mientras que las semánticas de el tipo de integración *Linkage* aun es válida (el nuevo nodo volátil "envoltorio personalizado" es navegado a partir de lo nodo "opciones de envío"), la semántica exacta del camino navegacional es definido en el modelo conceptual donde realmente las restricciones del workflow son definidas. Entonces, cuando un usuario navega por un *activity link*, el próximo nodo de actividad a ser abierto va a resultar desde la inserción volátil en el modelo conceptual que relaciona el objeto conceptual "envoltorio personalizado" con el objeto de la actividad previa ("Especificar las opciones de envío") y el siguiente "confirmación". El flujo completo de este tipo de personalización volátil puede ser también derivado desde una técnica presentada en [95].

Capítulo 9 Modelo de Interfaz de Usuario

Una vez finalizado el modelado Conceptual (paso 3) y posteriormente el modelado Navegacional (paso 4), se continua con el Modelado de Interfaz de Usuario (paso 5) que tiene como principal entrada los Modelos Navegacionales obtenido en la etapa anterior tal como se puede ver en la

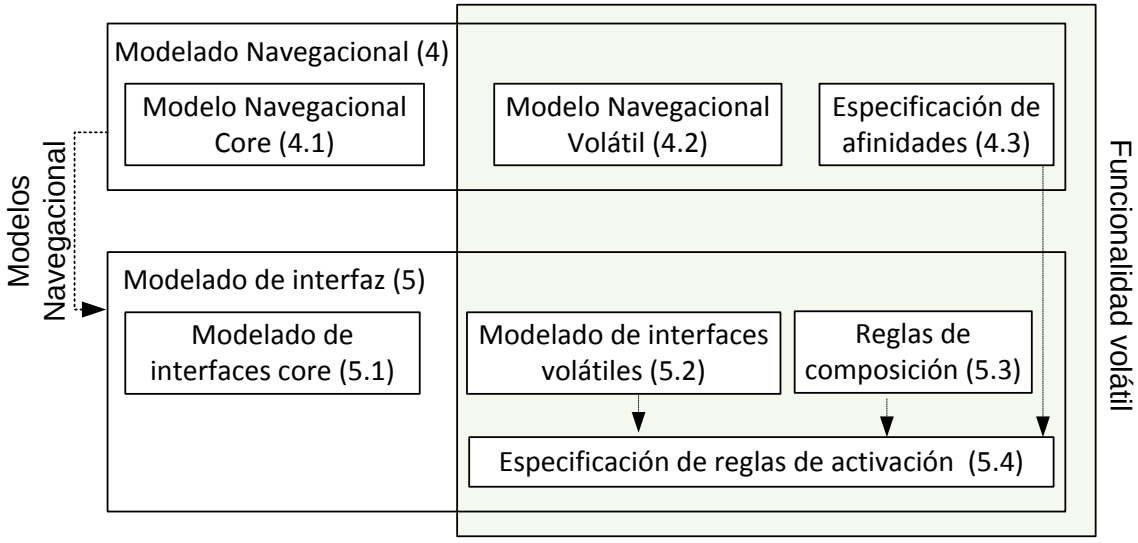


Ilustración 59 Tareas de la metodología especificas de la fase de Modelado de Interfaz de Usuario

Las interfaces de usuario también sufren el impacto de la incorporación y edición de funcionalidad volátil tanto en nivel de diseño como de implementación. Incluso si empujamos lenguajes como JSP[96] a sus límites en términos de modularidad, es prácticamente inevitable que el código que describe las interfaces de requerimientos core sea contaminado con tags (ya que es un lenguaje de marcas) perteneciente a funcionalidad volátil y en consecuencia ambos *concern* (core y volátil) se tornan *tangled*, o entrelazados. Por ejemplo, la página JSP que implementa la interfaz de un CD del ejemplo del comercio electrónico (para más detalle ver Ilustración 8 del Capítulo 1) va a tener conocimiento de dos *concerns* además del concern core *CD*: "San Valentín" (St. Valentine) y "Video promocional" tal como es mostrado en la Ilustración 60.

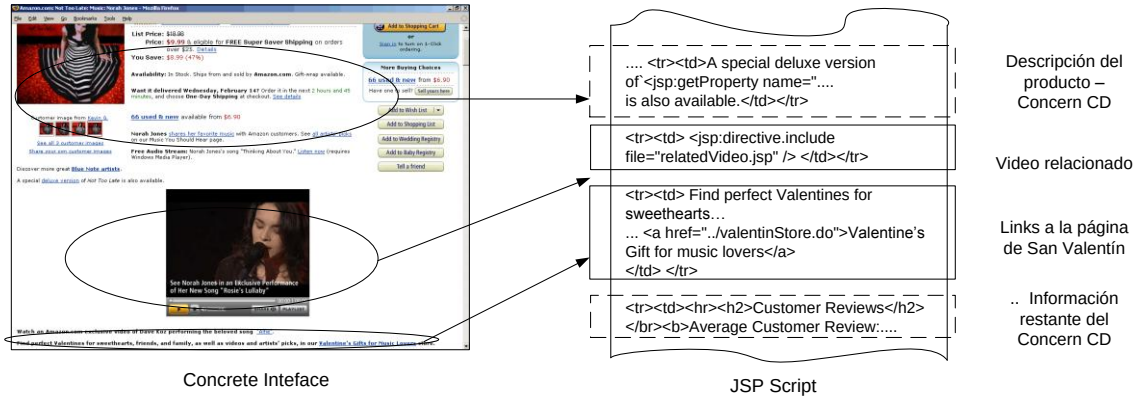


Ilustración 60 Código tangling en una interfaz de usuario

En la Ilustración 60 se puede apreciar dos bloques de código que refieren los *concern* "Video promocional" y "San Valentín", y que están claramente mezclados con el código del *concern* core (CD) con una pésima modularidad. Una situación similar surgiría a nivel de diseño de interfaz sin importar la notación utilizada para la tarea. Las clases correspondientes a la interfaz core tendrán referencias explicitas (tanto como atributos o relaciones) a las clases volátiles en el diseño de la interfaz.

Es importante dejar en claro que este problema es conceptual y no tecnológico. En tecnologías comunes (tal como JSP[96], JSF[97], etc.) o lenguajes de descripción de interfaces (tal como usiXML[58], UIML [98], etc.) se puede experimentar esta clase de mezcla de código debido a la falta de constructores primitivos para implementar una solución basada en polimorfismo (por ejemplo utilizando el patrón *Decorator*[26] para hacer transparente la inyección de lógica de interfaz) o en *pointcuts* tal como en la orientación a aspectos. La primitiva *include* [99], típica de los lenguajes de scripting, no garantiza transparencia porque produce una invocación explicita.

No sorprende que la separación de *concern* sea tan elusiva a nivel de interface; la mayoría de las técnicas de ingeniería Web han tratado estos aspectos como problemas de bajo nivel (por ejemplo posponiéndolos hasta la etapa de implementación). Evitar tangling, o mezcla, a nivel de interfaz permite una mejor composición de interfaces y diseño de las mismas. Sin embargo, las consecuencias de estos cambios son invasivos y parecen ser más perjudiciales a nivel de código (es decir en una página JSP). En esta tesis este problema será atacado en dos etapas diseño e implementación. Se proveerán herramientas conceptuales que permitan el adecuado diseño de las interfaces volátiles para luego mapear las construcciones abstractas en los correspondientes componentes de software (clases y widget visuales).

9.1 Diseño de Iinterfaces de Usuario Core

En OOHDM, la interfaz de usuario se define utilizando Abstract Data Views [100] el cual provee un modelo orientado a objetos para describir todos los objetos de la interfaz. Un ADV es definió

por cada nodo para indicar la información propuesta (atributos y subnodos en el caso de composición) por éste será presentada a los usuarios. Un ADV puede ser visto como la implementación de un patrón *Observer* [26] encargado de detectar cambios de estado en los nodos que serán propagados a la vista. Por otro lado la utilización de un diagrama de configuración [101] permite expresar cómo propiedades del ADV se relacionan con los atributos y operaciones del nodo navegacional.

Los ADV también pueden ser utilizados para indicar cómo la interacción va a proceder y cuales efectos de interfaz toman lugar como resultado de los eventos generados por el usuario al interactuar con ésta. Estos aspectos de comportamiento son expresados con los ADV-Charts [101], un tipo de diagrama de estados representando estados y transiciones para un ADV dado. Los ADV-Charts son útiles cuando nosotros necesitamos modelar comportamientos ricos de interfaz tal como los presentes en Aplicaciones Ricas de Internet (Rich Internet Applications, RIA).

ADV-Charts son diagramas de máquinas de estado[13] que permiten expresar las transformaciones de la interfaz que ocurren como resultado de la interacción del usuario sobre un ADV. El ADV-Chart describe el comportamiento de la interfaz a través de reglas Evento-Condición-Acción (ECA). Una descripción detallada de los ADV-Charts y su uso para especificar comportamientos de interfaz puede ser encontrada en [101].

En la Ilustración 61 se puede apreciar un ADV Product y su Interfaz de Usuario Final donde cada elemento del ADV (del lado izquierdo) es relacionado con la implementación final del mismo a la derecha. Es importante destacar que las Interfaz de Usuario está disponible una vez que la aplicación ha sido modelada completamente habiendo realizado los modelos conceptual, navegacional y de interfaz, por lo tanto, la Ilustración 61 solo se explicativo ya que la interfaz de usuario final aun no está disponible.

[13] O diagramas de estados si nos referimos a UML 1.x en lugar de UML 2.0

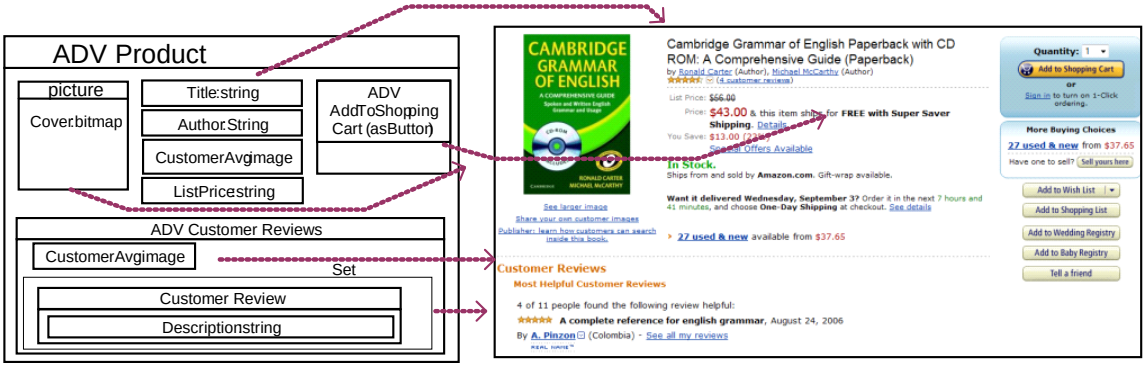


Ilustración 61 Relación de elementros del ADV Product y los elementos de su Interfaz de Usuario Final

9.2 Composición estructural de funcionalidad volátil en la Interfaz de Usuario

Como consecuencia de insertar funcionalidades volátiles dentro del modelo conceptual o modelo navegacional, nuevos elementos de la interfaz tal como nuevos campos de datos u objeto de control (botones y links) de usuario deben ser agregados dentro del modelo de la interfaz. El proceso para llevar a cabo esta tarea fue descripto en [65], a continuación se revisará brevemente éste procedimiento:

Cada *concern* (core y volátil) va a utilizar ADVs para describir cómo será presentada la información de sus correspondientes nodos. Durante la etapa de diseño de interfaz y cuando un nodo debe exhibir alguna funcionalidad volátil, se indica la apariencia (look and feel) de la página resultante de la incorporación por medio de la especificación de cómo la interfaz volátil va a ser insertada dentro del ADV core. Mas precisamente, se indica la posición relativa a un objeto core de la interfaz que se utilizará para guiar la inserción del ADV volátil. En Ilustración 62 se puede apreciar un esquema de cómo se realiza la inclusión de elementos volátiles (en la figura nombrado como ADV V1) en una interfaz core (en la figura Ilustración ADV C2). Del lado izquierdo se puede apreciar los ADVs core y volátil con una proyección de líneas punteadas denominada *Especificación de Integración.* Mientras que del lado derecho se puede apreciar el resultado de componer ambos ADVs teniendo dando como resultado una inclusión del ADV volátil V1 en el ADV core C2.

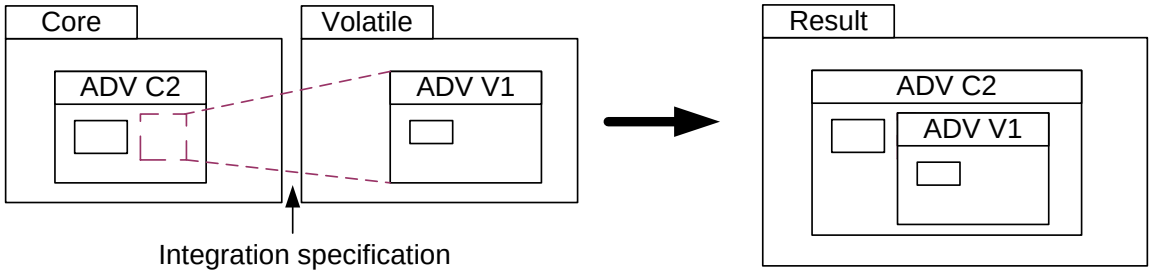


Ilustración 62 Esquema de composición de interfaces

Para expresar la integración, se definió un lenguaje de especificación simple que permite indicar *pointcuts* y el tipo de modificación a nivel de interfaz abstracta; esto es donde el ADV volátil tiene que ser insertado en el ADV core.

Es importante destacar que la *Especificación de Integración* está sujeta a la satisfacción de la *Afinidad* a nivel navegacional ya que la composición no puede darse sin que el nodo navegacional subyacente no brinde soporte a los elementos visuales que presentan información. El control enunciado aquí se realiza al momento de evaluar las el ciclo de vida de una funcionalidad volátil que será descripto en el Capítulo 10 .

La especificación generaliza la idea de *pointcuts* de la orientación a aspectos al espacio bidimensional de una interfaz Web. Un *pointcut* y la modificación deseada en la interfaz destino (core) son definidas utilizando la plantilla del Código 6.

```
Integration: IntegrationName

Target: ADVTargetName

Add: ADVSourceName | AnonymousDefinition

Relative to: ADV name

Position: [above | below | left | right | replace]
```

Código 6 Plantilla de Especificación de Integración

El campo *Integration* es un identificador para ésta especificación. Puede referir o no a una afinidad navegacional ya que la misma *Especificación de Integración* de interfaz de usuario puede ser utilizado con varias afinidades navegacionales. El campo *Target* indica el nombre del ADVs que recibirán el código de la interfaz volátil. Los ADV internos (también conocidos como anidados o *nested*) pueden ser especificado utilizando una notación de puntos (“.”) similar a la utilizada por OGNL [102] para navegar por los objetos compuestos. Por ejemplo, *Product.Reviews* indica que la inserción va a tomar lugar en el ADV *Reviews* que es parte del ADV *Product*. El campo *Add* indica cual elemento debe ser insertado en el destino, tanto un ADV o una especificación anónima, referenciado como AnonymousDefinition, (directamente se referencia el widget UI) que es utiliza cuando el campo a ser insertado es suficientemente simple

para evitar la especificación de otro ADV (auxiliar). Finalmente, la posición de inserción es señalada utilizando los campos *Relative* para disponer un punto de referencia y *Position* para indicar, a partir de la referencia, dónde se ubicará el ADV indicado por el campo *Add:* arriba (above), abajo (below), izquierda (left), y derecha (right). Existe un tipo de posición especial llamado *replace* que sobreescribe el elemento indicado en el campo *Relative to.*

Es importante destacar que la *Especificación de Integración* es abstracta, permitiendo flexibilidad para introducir ajustes personalizados al momento de su implementación.

Para profundizar el tema, consideremos de nuevo el ejemplo de funcionalidad volátil de *Volver a la Escuela* de la Sección 1. La especificación de abajo (integración de volver a la escuela) indica que el ADV de la página de inicio (ver Ilustración 1) de Amazon será enriquecida con un ADV que posee un link (como componente UI para la página resultante) a la página índice de los productos de *Volver a la Escuela* (ver Ilustración 2 para la página resultante).

```
Integration: Back to School integration – Home page

Target: ADVHomePage

Add: Anchor(ADV BackToSchool)

Relative to: ADVHomePage.CheckThisOut.NewAndUsedTextBooks

Position: below
```

Código 7 Especificación de Integración que introduce un link al ADV de BackToSchool

La Especificación de Integración del Código 8 indica que el ADV de la página principal (*ADVHomePage*) va a ser modificada con la introducción de un nuevo link definido en tiempo de ejecución por el ADV *BackToSchool*. El *joinpoint* es definido por la propiedad *Relative to* que describe el camino al componente que será utilizado como referencia relativa de la inserción. Finalmente el atributo *Position* señala cual lugar, en base al componente pívot, será utilizado para la incorporación del nuevo link anónimo. La especificación intenta mantener el documento ADV tan simple como sea posible, evitando la incorporación de información dependiente a la tecnología utilizada para su implementación. Como se mostró en la Ilustración 1 del capítulo 1, el link que fue agregado a la página es señalado con un óvalo punteado sin la necesidad de definir código intrusivo al ADV de la página principal.

Para poder agregar un link a cualquier producto etiquetado (o clasificado internamente) como "escolar" que permita navegar a la página de *Volver a la Escuela*, podemos utilizar la *Especificación de Integración* definida en Código 8.

```
Integration: Back to School integration – Generic link

Target: ADV Product

Add: Anchor(ADV BackToSchool)

Relative to: ADV AddToShoppingCart

Position: below
```

Código 8 Especificación de Integración que incorpora un link al ADV Product

Como puede ser fácilmente apreciado, la *Especificación de Integración* incrementa el *ADV Product* con un link al *ADV BackToSchool* para que se posicione debajo del ADV *AddToShoppingCart* (el cual básicamente corresponde a un componente UI del tipo botón). Para llevar a cabo esta modificación a la interfaz, se requiere que el modelo navegacional posea definido el link. En la Sección 8.1 se presenta la *afinidad* que implementa la incorporación del *link* en el modelo navegacional necesario para la integración de la interfaz indicada previamente.

9.3 Funcionalidad volátil de comportamiento en interfaces de usuario

Una interesante y ciertamente desafiante situación surge cuando la funcionalidad volátil modifica funcionalidades core. Es decir, cuando su incorporación modifica comportamiento existente en el modelo de Interfaz de Usuario core en lugar de cuestiones estructurales como se discutió en la Sección 9.2. Los comportamientos *crosscutting*, o transversales, pueden manifestarse de diferentes formas y, de acuerdo a la caracterización presentada (Tabla 1 del Capítulo 4), éste puede afectar los modelos conceptual, navegacional y de interfaz. Los comportamientos *crosscutting* en el modelo conceptual pueden ser resueltos de forma sencilla utilizando técnicas bien conocidas de ingeniería de software tal como la orientación a aspectos o con el uso de ciertos patrones de diseño; ambas alternativas de diseño fueron discutidas en el Capítulo 7 . Mientras tanto, comportamientos *crosscutting* en el modelo navegacional dependen fuertemente del estilo de modelado, y, como muestran los análisis de técnicas de modelado navegacional existentes, comportamientos complejos no son usuales en éste modelo ya que el modelo conceptual es quien contiene la mayor parte de los comportamientos *core*. Por lo tanto, y por una cuestión de consistencia, esta sección hará foco en el modelo de interfaz ya que, con la creciente popularidad de las aplicaciones Web RIA, más y más comportamientos sofisticados son presentados en las interfaces de usuario y pueden ser afectados por inserciones de funcionalidades volátiles.

Continuando con el ejemplo de *Volver a la Escuela*, supongamos que se desea implementar un requerimiento de nueva promoción que brinda un servicio de envío a domicilio sin cargo en la compra de tres o más productos (de aquí en adelante *Promoción*). El requerimiento especifica que cuando cualquier producto clasificado como "escolar" sea agregado al carrito de compras tras presionar el botón "*addToShoppingCart*", se presente una sugerencia (por ejemplo, mediante un pop-up). En el lado izquierdo de la Ilustración 63 se muestra un ejemplo donde se sugiere la compra de un lector de libros digitales Kindle con envío gratuito a domicilio y, en el lado derecho de la misma figura, se muestra el ADV de la interfaz donde líneas punteadas fueron utilizadas

para relacionar los elementos de las interfaces concretas y abstractas. Adicionalmente, se requiere disparar la adición del producto Kindle en el carrito de compras así como también aplicar la promoción de envío sin cargo si el usuario acepta la oferta.

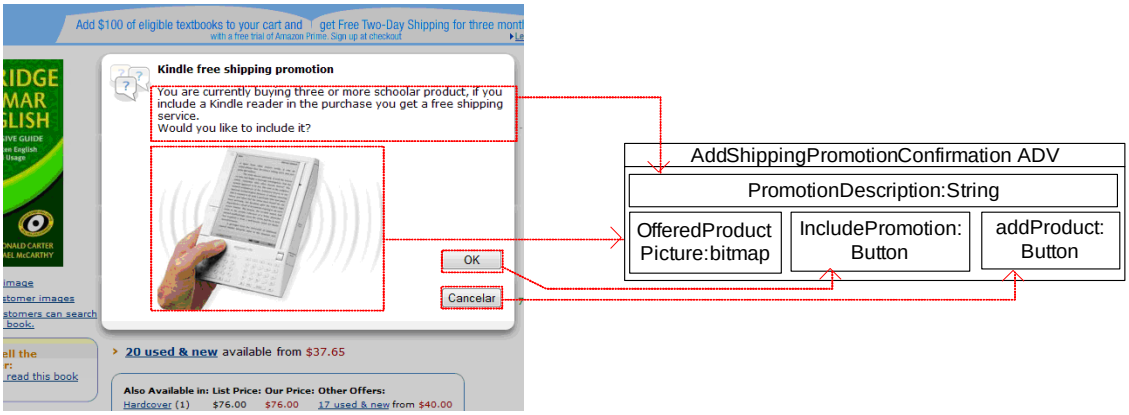


Ilustración 63 Pop-up mostrado tras presionar el botón AddToShoppingCart con su correspondiente ADV

Para resolver este problema, a continuación se desarrolla una extensión de la notación ADV-Chart para permitir la especificación de composiciones de comportamiento de forma no intrusiva la interfaz core.

En la Ilustración 64 se muestra cómo se especifica el comportamiento entre ambos ADVs por medio de ADV-Charts. Es decir, la composición del comportamiento del ADV de la promoción Kindle con el de un ADV Producto, en este caso un libro; posteriormente a la composición estructural de ADV.

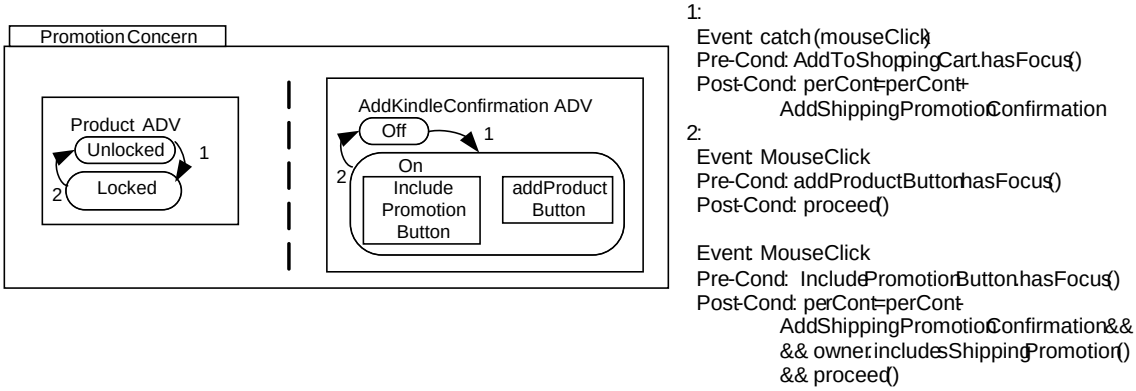


Ilustración 64 ADV-Chart correspondiente a los estados del ADV de Confirmación

Básicamente, el ADV-Chart describe el siguiente comportamiento para la interfaz resultante:

- Interceptar el evento *mouseClick* del botón *addToShoppingCart* (definido en el ADV de la Ilustración 61) utilizando la palabra clave *catch*;
- Habilitar la interfaz pop-up para estar en el estado *On* y bloquear la navegación a la página siguiente;
- Al presionar el botón *addProduct*, retornar el control al ADV-Chart original (utilizando la palabra clave *procced*) donde solo el producto seleccionado va a ser agregado al carro;
- Lanzar el evento que causa que, además de agregar el producto seleccionado, se agregue un lector Kindle. Luego, se permite al proceso de negocio continuar como fue originalmente ideado utilizando la palabra clave *proceed*.

Para que el procesamiento sea completo, dos métodos tienen que ser agregados al nodo volátil correspondiente, llamados *includesShippingPromotion* y *addShippingPromotion.* El primero verifica si el carrito de compras registra una promoción de envío a domicilio y el segundo método es utilizado para registrar la promoción.

La extensión de ADV-Charts presentada toma ventajas de las técnicas de orientación a aspectos bien conocidas para mantener la interfaz core (*Product ADV*) independiente de los requerimientos de la Promoción de envío a domicilio. En pocas palabras, la solución es alcanzada por medio del procesamiento de la página original del *Producto* e introduciendo el nuevo comportamiento por medio de una composición. El proceso de composición toma la especificación de la Promoción y vincula el punto de enganche (*joinpoint*), capturar el evento de click de mouse, con el componente de la página core destino (botón *AddToShoppingCart* definido el ADV de la Ilustración 61).

Cambiemos al ejemplo *ChangeablePicture* (introducido en la Sección 1.1.1 y diseñado en la Sección 2.1.1.4.2) para mostrar cómo diseñar otros comportamientos ricos. Supongamos que se requiere poder destacar ciertos aspectos de una imagen relacionada a un producto con diferentes comentarios realizados por los usuarios. Dado que esta funcionalidad es una iniciativa para facilitar la captura de conocimiento colectivo, se desea incorporarla en el sistema para ver cómo los usuarios la adoptan. En el caso de no causar el impacto deseado, la misma deberá ser retirada. De esta forma, esta funcionalidad encaja claramente en la definición de una funcionalidad volátil. En la Ilustración 12 (de la Sección 2.1.1.4.2) se muestra el diseño del ADV de detalle de un producto. Mientras que en la Ilustración 65 se muestra cómo el ADV *ChangeablePicture* ha sido mejorado incorporando la posibilidad de comentar áreas específicas de la imagen.

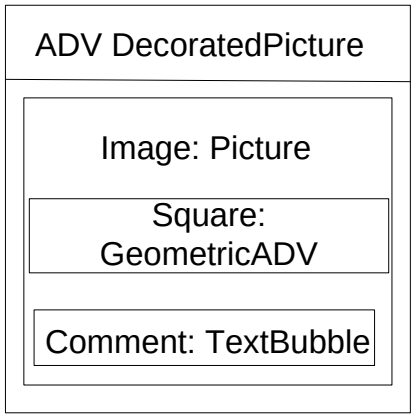


Ilustración 65 ADV del decorador de Imagen que permite comentarios

Dado que la nueva funcionalidad está en estado de prueba beta (esperando el retorno de los usuarios), el componente *Image* de interfaz original deberá permanecer intacto. Este ejemplo será acotado solo a la funcionalidad que permite visualizar los comentarios ya que la funcionalidad de edición de la imagen para agregar un comentario es similar.

Este comportamiento RIA puede ser incorporado de forma transparente utilizando un *Decorator* para decorar el comportamiento original. En primer lugar, un decorador debe ser instanciado para la incorporar comentarios sobre la Imagen original del ADV de la Ilustración 12. Este decorador lo enriquece con los mensajes *getSquarePosition()* y *getComment().* Luego el ADV *ChangeablePicture* es perfeccionado remplazando sus campos de imagen estáticos con una nueva versión decorada de los mismos; de acuerdo con el ADV especificado en Ilustración 12 y ADV-chart especificado en la Ilustración 66; ambos definidos en el paquete volátil de la funcionalidad.

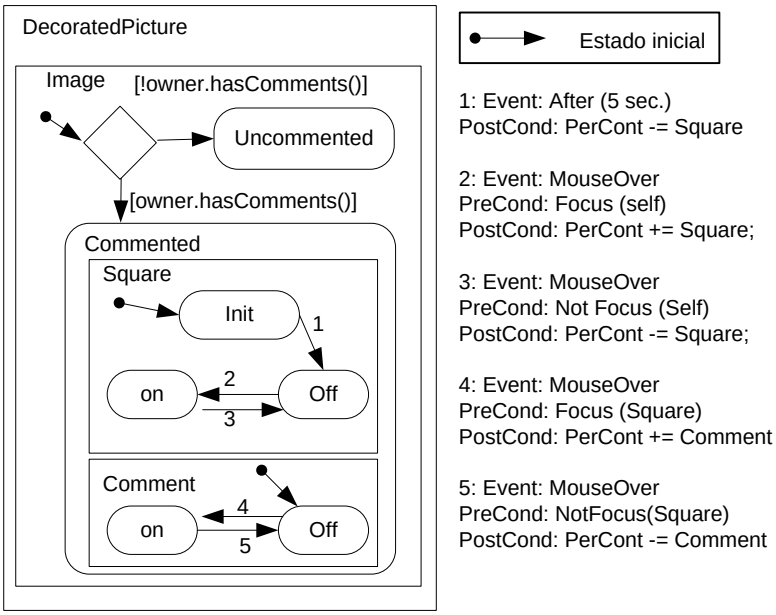


Ilustración 66 ADV-Chart correspondiente a la funcionalidad de comentarios sobre imagenes

El ADV-Chart del componente *DecoratedPicture* está compuesto por dos estados principales: estado *Commented* si su dueño (una figura decorada) tiene comentarios, y estado *UnCommented* cuando, caso contrario, imágenes estáticas son percibidas.

Cuando el ADV-Chart está en el estado *Commented*, los ADV *Square* (cuadrado utilizado para destacar una zona de la imagen) y *Comment* (utilizado para mostrar los comentarios de una zona destacable) deben ser visibles. La transición 1 del ADV-Chart especifica que el ADV *Square* tiene que desaparecer 5 segundos después que el ADV fue abierto. La segunda transición indica que cuándo el puntero del mouse está sobre la instancia del componente *DecoratedPicture*, el ADV S*quare* tiene que ser perceptible nuevamente hasta que el puntero no esté sobre el componente tal como lo especifica la transición 3. Las transiciones 4 y 5 respectivamente especifican la dinámica de la burbuja de comentarios en función del foco del mouse sobre el ADV S*quare*.

Finalmente, se necesita incluir la nueva estructura que permite comentar las imágenes dentro del componente original. Para expresar esta integración, se puede aplicar la Especificación de Integración del Código 9 sobre el ADV *ChangeablePicture* para reemplazar dinámicamente su campo de imagen estática con la nueva versión potenciada con la posibilidad de ser anotada.

```
Target ADV ChangeablePicture

Add Image DecoratedPicture

RelativeTo ADV ChangeablePicture.Image

Position replace
```

Código 9 Especificación de integración para incorporar comentarios a la una imagen

El campo *target* indica el nombre del ADV que va a sufrir la transformación. El campo *Add* indica cuáles elementos deben ser insertados en el ADV destino, tanto un ADV como una especificación anónima de elementos, que es utilizado cuando el campo insertado es lo suficientemente simple para evitar la especificación de otro ADV. Finalmente, se indica la posición de inserción utilizando el campo *Relative*, el cual en este caso, es un ADV anidado llamado *Image.* Como se pudo ver se utilizó la notación "." para navegar en los elementos de un ADV.

9.3.1 Implementación ADV y ADV-Charts

9.3.1.1 Interfaces

Los ADVs pueden ser mapeados de forma automática en páginas Web implementados con tecnologías vigentes tal como JSP, JSF, XSL, etc. y los aspectos de comportamiento rico asociado a aplicaciones RIA pueden ser implementados utilizando AJAX [103] u OpenLaszlo[104] donde ésta última es una tecnología basada en XML.

9.3.1.2 Composición de características estructurales y de comportamiento volátiles

El problema de implementar requerimientos volátiles a nivel de interfaz de usuario puede ser expresado en términos de transformaciones de documentos XML: dados dos documentos A y B que expresan el contenido de un nodo, se necesita describir como obtener un documento que integra B dentro de A (como un elemento adicional), sin una referencia explícita de dicha incorporación en A.

Más aun, en el caso de funcionalidad volátil irregular, se necesita que esta integración sea hecha en todos los documentos que cumplen algunas condiciones; esto finalmente puede involucrar instancias específicas de tipos diferentes de documentos.

Utilizando Transformaciones XSL, el comportamiento rico puede ser incorporado, en el caso de interfaces basadas en HTML y JavaScript, en la interfaz insertando bloques de funciones, o construcciones de más alto nivel, implementados con JavaScript. En algunos casos, cuando el comportamiento existente de las interfaces son sobrescritos, se puede beneficiar de las facilidades

JavaScript que permiten redefinir funciones en tiempo de ejecución para enriquecer una función decorándola con otra [105], [106]

La base de la solución propuesta en esta tesis para documentos XML es la utilización de Transformaciones XSL[107] para componer interfaces core y volátiles, y XPath[108] para indicar las partes del documento del código fuente en donde se inyectan los extractos de XML correspondiente a las interfaces volátiles. La transformación actúa como un aspecto en la orientación a aspecto: el contenido de una transformación XSL actúa como un *advice*, mientras que la expresión XPath indica el punto donde el *advice* debe ser insertado. Un Motor de XSLT (por ejemplo Xalan[109] o Saxon [110]) realiza la composición o tejido de *concern*.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://…." xmlns:jsp="http://….” >
   <!--Imports a transformation which copies all the elements-->
   <xsl:import href="defaultTemplate.xsl"/>
   <xsl:template match="//tr[contains(.,'Customer Reviews')]">

      <tr><td> <jsp:directive.include file="fishbowl.jsp" /></td></tr>
      <tr><xsl:apply-templates select="*"/></tr>

   </xsl:template>
</xsl:stylesheet>
```

Pointcut

Advice

Ilustración 67 Transformación que inserta lógica de “Video promocional” a la interfaz del ADV Product

Por ejemplo, para agregar la lógica de “Video promocional”, encapsulada en un componente, en una interface del *Producto* (ver el resultado y código fuente en Ilustración 60), se puede aplicar la transformación XSL (ver Ilustración 67) sobre la interfaz de usuario. La Expresión XPath "//tr[contains(.,'Customer Reviews')]" juega el rol de *pointcut* especificando como punto la fila de una tabla que contiene el texto “Customer Reviews” (ver Ilustración 60). La plantilla (*advice*) deja los elementos existentes de la fila con el texto sin cambiar e inserta debajo una nueva fila con el elemento *include* (que tiene la función de incluir el documento “fishbowl.jsp”).

En Ilustración 68, se muestra el esquema se toma el código fuente de las interfaces core y volátiles para producir una interfaz de usuario enriquecida.

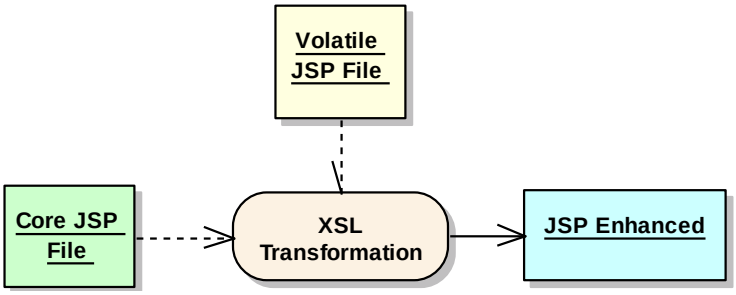


Ilustración 68 Composición de interfaces basadas en documentos XML

Un problema con esta solución es que cuando la funcionalidad volátil solo afecta a algunas interfaces de una clase (por ejemplo todos los productos que van a ser recomendados para ser presentados en las promociones de "San Valentín"), tipos de condiciones estructurales deben ser incluidas en los tags para discriminar que instancias requieren el cambio, ensuciando el código fuente resultante.

9.4 Modelando comportamiento volátil en interfaces de usuario con aspectos

Existen situaciones en donde los ADV-Charts no son suficientes para describir el comportamiento rico de las interfaces de usuario. Para estos casos, se propone utilizar la metodología aplicada para el diseño del Modelo Conceptual utilizando diagramas de secuencia UML en combinación con MATA (ver Sección 7.4.1). Para ilustrar esta alternativa de modelado, continuaremos el ejemplo de la aplicación Web de comercio electrónico que permite calcular la hoja de ruta que los transportistas deben seguir para el servicio de envío a domicilio.

La funcionalidad de búsqueda de camino ya fue diseñada a nivel Conceptual en Sección 7.4.1. En esta Sección se presenta el diagrama de secuencia Ilustración 47 para mostrar cómo se resuelve un requerimiento de búsqueda de camino desde que el usuario realiza el pedido a objetos que corresponden a la Interfaz de Usuario hasta que alanzan objetos de negocio correspondientes al Modelo Conceptual. El diseño de la resolución de caminos ya ha sido discutido en la Sección 7.4.1. En el diagrama de la Ilustración 47, se presentan los objetos *RoutePlannerController* y *PlannerView*. El primero corresponde al Nodo navegacional mientras que el segundo representa al ADV que presenta la información al usuario. Por simplicidad, este diagrama solo muestra las clases y comportamiento relevante.

El ejemplo de funcionalidad volátil introducido en la Sección 1.1.2, requiere poder contemplar de forma indeterminada la posibilidad de soportar calles bloqueadas al momento de generar las hojas de ruta para los encargados del transporte de productos. En la sección 7.4, se discutió el diseño del Modelo Conceptual del comportamiento introducido por la funcionalidad volátil dejando pendiente cómo presentar adecuadamente al usuario el estado de las calles. La no "notificación"

de las calles cortadas al usuario podría generar desconfianza en el sistema ya que a simple vista, en el mapa de la ruta, el camino visualizado no es "óptimo".

9.4.1 Diseño de Interfaz de Usuario utilizando MATA

Para proveer una adecuada experiencia del usuario, se diseñará el requerimiento R13 (de la Sección 7.1) que demanda una presentación adecuada de las *calles bloqueadas* en la interfaz de usuario. En este caso, utilizaremos un diagrama de secuencia UML con MATA para describir cómo introducir la adecuada presentación de las *calles bloqueadas*. En el diagrama de la Ilustración 69 se muestra un diseño de aspecto para el controlador *RouteController* que decora el método *renderPath* con la lógica necesaria para dibujar las calles cortadas en el mapa. En el diagrama, el método *renderBlockedStreet* es invocada una vez que se procesa el método *renderPath*, dibujando una línea azul en el mapa para indicar que la calle está cortada y aagregando un pop-up describiendo la situación tal como puede ser apreciado en Ilustración 6 del Capítulo 1.

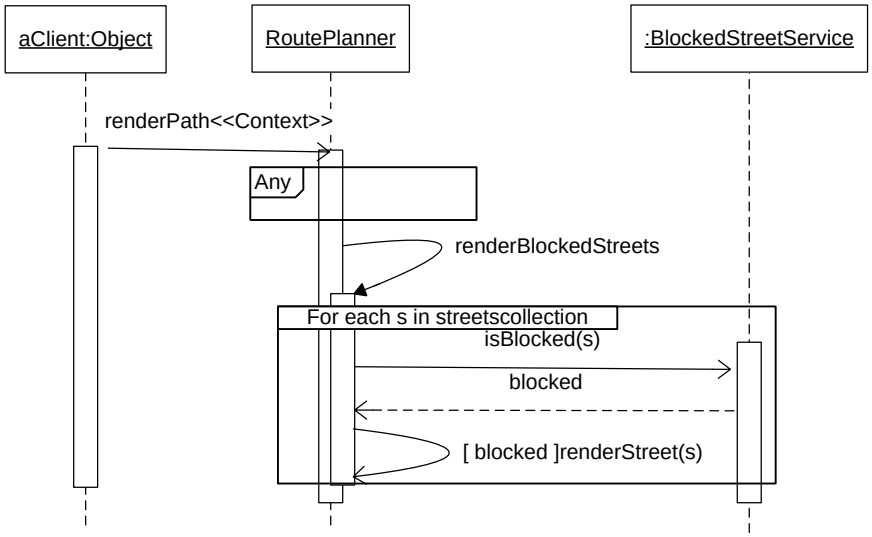


Ilustración 69 Aspecto para la adecuada presentación de la *calles bloqueadas*

9.4.2 Resultado de la composición de aspectos

Esta sección tiene como objetivo ilustrar como los modelos core son aumentados siendo enriquecidos con Crosscutting concerns.

En la Ilustración 70 presenta el diagrama de secuencia final en el cual el concern de *Calles bloqueadas* ha sido compuesto con el concern core *Route Planning* que permite calcular el recorrido que debe realizarse para llevar un producto al domicilio del cliente. El resultado de la composición es ilustrativo ya que los aspectos deben ser compuestos de forma automática por la plataforma subyacente.

El diagrama describe como una red de calles es reducida por medio del filtrado de *calles bloqueadas*, y como estas calles son "dibujadas" (de forma descriptiva) antes de que el mapa sea presentado al usuario. La composición del aspecto de que permite reducir el conjunto de datos (Ilustración 48) resulta en el enriquecimiento del método *getAdjacents* (en la clase *Node*); la mejora agrega la lógica requerida para el filtrado de las calles bloqueadas por medio de una llamada al mensaje *filterBlockedStreets*. El aspecto que provee la apropiada presentación de las *calles cortadas* contribuye enriqueciendo el método *renderPath* (en la clase *PlannerView*) que dibuja cada calle cortada de forma diferente además de las calles utilizadas por las ruta de forma tradicional tal como puede ser visto en Ilustración 69.

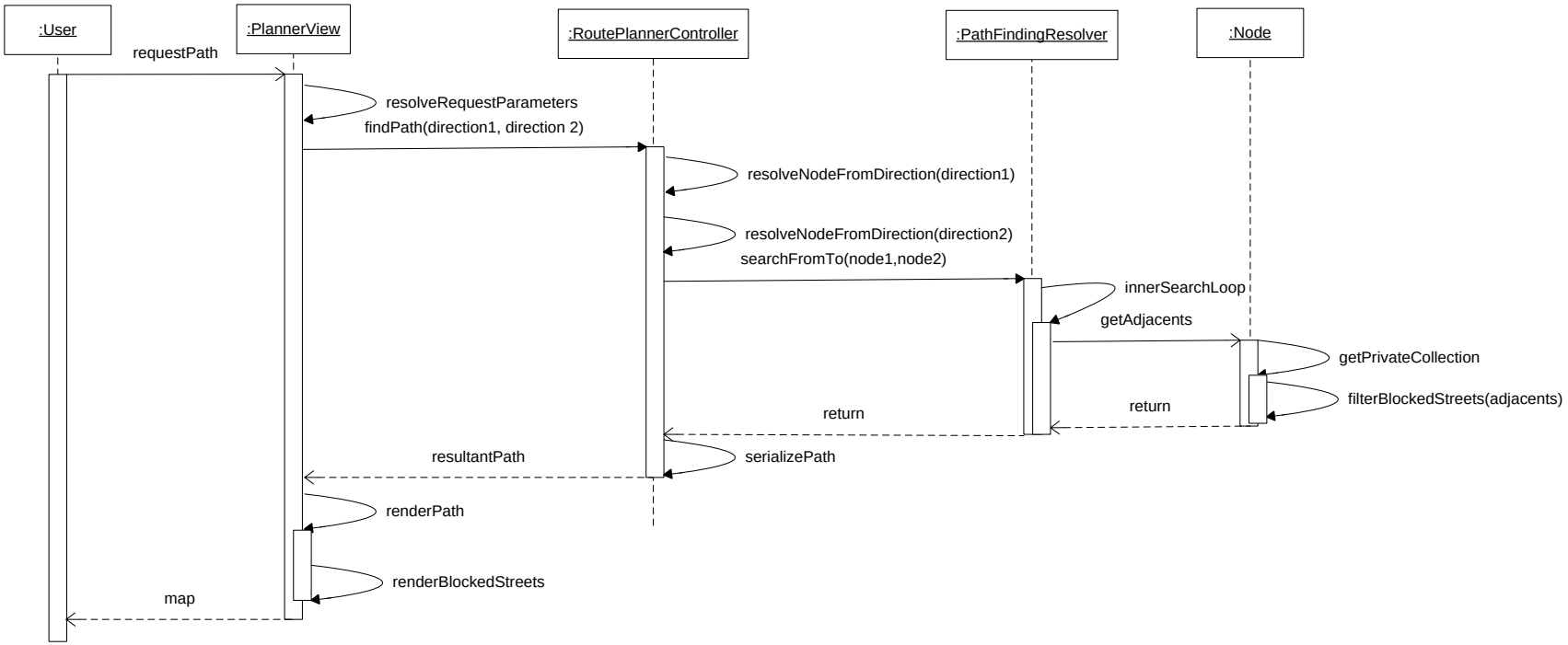


Ilustración 70 Resultado de componer aspectos basados MATA con sobre Modelos Conceptual y de Interfaz de Usuario core

9.4.3 Detalles de implementación del ejemplo de *Calles bloqueadas*

Las calles bloqueadas deben ser claramente presentadas a usuario permitiéndole a él entender el resultado a primera vista de una consulta; es decir, las calles bloqueadas deben visualizarse de alguna forma en especial utilizando colores y marcas visuales. En este ejemplo, la aplicación utiliza el framework de código abierto OpenLayers [111] que provee facilidades para dibujar mapas con diferentes capas. En este caso, al aplicación superpone un mapa básico (en una capa) con la ruta resuelta (en otro mapa). Con el objetivo de implementar el diseño presentado en la Sección 9.4, se debe introducir la lógica para dibujar las *calles bloqueadas*.

La solución que dibuja las calles bloqueadas, mostrada en Ilustración 6 (del Capítulo 1) fue aplicada como una nueva capa de mapa (de ahora en adelante *capa de calles bloqueadas*) que son superpuestas sobre la capa del mapa básico y la capa de la ruta calculada.

Además, tomando en cuenta que las capas de presentación son usualmente definidas utilizando documentos XML tal como HTML, la solución propone utilizar transformaciones de interfaz para aplicar los cambios.

Por lo tanto, utilizando la técnica basada en transformaciones de la Sección 9.2, nuevos Widgets, etiquetas, manejadores de eventos, etc. son adjuntados al documento que define el mapa sin ser consiente por los cambios producidos por el *concern* calle cortada.

En términos técnicos, el controlador de mapa, en el lado del cliente, fue enriquecido con la lógica necesitaría para manejar los eventos (o requisitos HTTP) producidos desde la *capa calles bloqueadas*, del lado del cliente. La mejora es aplicada utilizando un aspecto interceptor que captura cada requisito HTTP proveniente desde el navegador Web; éste realiza procesamiento personalizado a cada pieza[14], o baldosa, de mapa resultante de una solicitud HTTP proveniente de la *capa calles bloqueadas*.

[14] La estrategia habitual para dibujar mapas es fragmentar el mapa en baldosas. Por ende, un mapa podría estar compuesto por varias baldosas tal como si fuese un puzle.

Capítulo 10 Gestión del ciclo de vida de las funcionalidades volátiles

En este Capítulo se describirá brevemente como automatizar el proceso de activación/desactivación de funcionalidades volátiles que fueron modelados siguiendo la metodología de diseño de funcionalidades volátiles descripta en los Capítulos anteriores. Primero se presentará un modelo simple para el entendimiento del ciclo de estas funcionalidades, luego se describirá un lenguaje basado en reglas para simplificar la especificación de las condiciones de activación y desactivación.

10.1 Modelo de ciclo de vida

En el contexto de ésta metodología basada en OOHDM, se puede razonar en el ciclo de vida de una funcionalidad volátil para entender mejor cuándo ésta tiene que ser conectada en la aplicación, cuándo desconectada o reconectada nuevamente, cuáles modelos de diseño ésta afecta, etc..

Inicialmente, la funcionalidad volátil es introducida en el sistema en un estado pasivo. Cuando un evento específico surge, como la llegada de una fecha específica o un evento de aplicación es disparado, la funcionalidad "despierta" y pasa a un estado "activo" donde las estructuras y comportamientos comprendidos son ejecutables mediante los cambios introducidos por la maquinaria de composición, o *weaving*. Más tarde, otros eventos pueden causar que la funcionalidad sea desactivada; este par de transiciones (activo-pasivo) puede suceder varias veces dependiendo de los requerimientos de la funcionalidad. En la Ilustración 71 se presenta esquemáticamente los posibles estados y transiciones de estados de una funcionalidad volátil en su ciclo de vida.

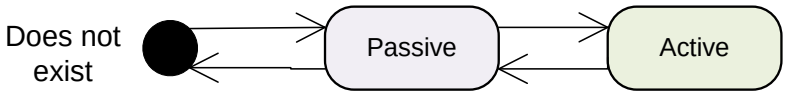


Ilustración 71 Esquema de estados del ciclo de vida de una funcionalidad volátil

Una vez que una funcionalidad volátil finaliza su actividad entrando en estado pasivo, ingenieros pueden decidir mantenerla inactiva en la aplicación durante un tiempo indeterminado para ser reactivada en un futuro, descartarla inmediatamente o preservarla para resolver información registrada cuando sea necesario; todos estos escenarios pueden cumplirse sin afectar los componentes core de la aplicación Web.

En la Ilustración 72 se representa el impacto de incorporar las funcionalidades volátiles del "San Valentín", "Videos promocionales", y "Conciertos promocionales". Los dos primeros ya fueron desarrollados a lo largo de esta tesis mientras que el último corresponde a una funcionalidad volátil que lanza la publicidad y venta de entradas a conciertos del autor del CD; es decir, al acceder al CD será posible conocer información del próximo concierto promocional del autor así como también adquirir las entradas para el mismo. El esquema destaca para cada nueva funcionalidad los objetos afectados (tanto nuevas relaciones como nuevos objetos), los modelos afectados (las modificaciones en cada modelo) y el patrón de volatilidad. Considerando sus ciclos de vida, las funcionalidades pueden cubrir diferentes periodos de tiempo donde en algunos momentos pueden incluso convivir; por ejemplo, la funcionalidad volátil de "San Valentín convive tanto con la funcionalidad volátil "Videos promocionales" así como también "Conciertos Promocionales". La ventana de tiempo puede ser fija para aquellos casos donde vigencia es conocida; por ejemplo, la promoción del "San Valentín" tiene un día de inicio anterior al 14 de febrero y finaliza el 18 del mismo mes. Pero hay casos donde el final de vigencia no es fijo y depende de predicados definidos sobre el contexto del momento. En el caso de los "Videos promocionales" y "Conciertos promocionales", el primero es desactivado cuando el CD deja de ser una novedad en base a su consumo mientras que el segundo se desactiva una vez que el cantante publica un nuevo CD; seguramente el cantante promocionará el CD con un nuevo concierto.

Podemos apreciar en la Ilustración 72 los diferentes objetos comprometidos por la funcionalidad volátil. Destacado con magenta, el requerimiento volátil "San Valentín" introduce el objeto *ValentinStore* en el Modelo Conceptual y una relación entre este ultimo y la clase CD; el nodo *ValentinStoreNode* en el Modelo Navegacional con un *link* desde el nodo CD a este nuevo nodo ; y en *anchor* a el nodo *ValentinStoreNode* en el Modelo de Interfaz. Por otro lado, el concern "Video promocional" resaltado con azul, modifica los tres modelos de forma similar al concern anterior: incorporar la clase *Video* en el Modelo Conceptual y el *Video ADV* correspondiente en la interfaz de Usuario. El concern de Conciertos promocionales, incorpora un nuevo nodo ConcernNode para presentar la información del concierto y por cuestiones de simplicidad no se mostró el ADV de este nodo. Se puede apreciar en la ilustración que al finalizar los periodos de vigencia de los concern "San Valentín" y "Videos promocionales" los cambios introducidos dejan de ser validos y son retirados del los modelos correspondientes.

Las funcionalidades volátiles activas en el mismo momento pueden afectar el mismo objeto obteniendo mejoras complejas de la aplicación. Por ejemplo, la funcionalidad volátil correspondiente al *concern* "San Valentín" introduce objetos en las tres capas de la aplicación mientras los "Video promocional", nombrado como Video, mejora los mismos objetos conceptuales, navegacionales y de interfaz que el *concern* anterior.

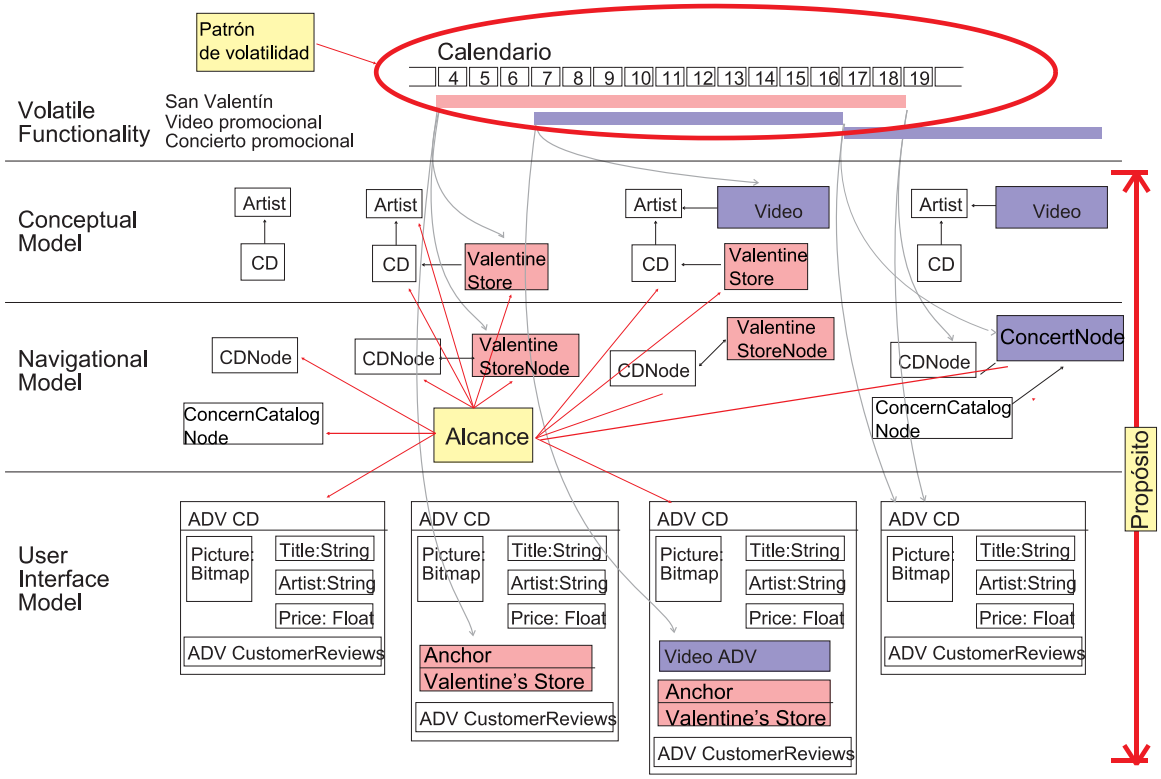


Ilustración 72 Esquema de impacto de las funcionalidades volátiles Dia de la madre y Videos promocionales en una aplicación Web

Dado que la metodología presentada en este trabajo para el diseño de funcionalidad volátil se basada en modelos sin acople con las funcionalidades core, el proceso para conectarlas y desconectarlas debe ser también contemplado y para ello se utiliza un Lenguaje Específico de Dominio ya que las características de la volatilidad pueden seguir algunos patrones predecibles que pueden ayudar automatizar su activación o desactivación. Por lo tanto, en éste trabajo se extiende el lenguaje de *Especificación de Integración* descripto en la Sección 9.2 incorporando *Reglas de Producción* [112] para expresar reglas complejas de activación de funcionalidad volátil.

Las condiciones basadas en eventos que disparan la activación/desactivación de una funcionalidad son expresadas utilizando patrones de eventos [113], la cual permite expresar condiciones de eventos ricas, correlación y posiblemente ventana de tiempo de vigencia. Condiciones complejas de eventos pueden ser expresados combinando diferentes patrones de eventos con lógica de primer orden.

También, para poder facilitar el entendimiento de las *Reglas de Producción* por clientes sin conocimientos técnicos, las condiciones y acciones son expresadas utilizando un DSL conocido.

Las reglas escritas bajo el DSL son luego interpretados por un motor de reglas con soporte de Procesamiento Complejo de Eventos (CEP, *Complex Event Processing*) [113].

La regla de activación respeta el formato del Código 10.

```
WHEN
    (Event_Pattern_Expression)
THEN
    (CONNECT | DISCONNECT)
    Concern concern_Name
    NAV_Affinity Affinity_Name
    UI_Integration Integration_Name
```

Código 10 Plantilla de *Regla de Producción*

En el ejemplo del *Día de la Madre*, la funcionalidad volátil puede ser activada/desactivada en fechas específicas utilizando eventos de tiempo. En la expresión del Código 11, la funcionalidad volátil está siendo activada un mes antes del *Día de la Madre*.

```
WHEN
    Time is *-Apr-14 00:00
THEN CONNECT Concern MothersDay
    NAV_Affinity MothersDay
    UI_Integration MothersDayHomePage
```

Código 11 Regla de Producción de activación del concern del Día de la Madre

Para desconectar la funcionalidad volátil, se utiliza otra condición basada en fechas tal como se puede apreciar en el Código 12 en la cual la los elementos de las diferentes capas (conceptual, navegacional, e Interfaz de Usuario) introducidos por la funcionalidad volátil son removidos cuando el Día de la Madre finaliza.

```
WHEN
    Time is *-May-14 23:59
THEN
    DISCONNECT Concern MothersDay
    NAV_Affinity MothersDay
    UI_Integration MothersDayHomePage
```

Código 12 Regla de Producción de desactivación del concern del Día de la Madre

Combinando expresiones basadas en eventos y en tiempo, se pueden describir ciclos de vida complejos para requerimientos volátiles.

Capítulo 11 Framework de soporte de la metodología: Cazon

Para poder soportar los conceptos desarrollados a lo largo de las diferentes investigaciones que dieron origen a esta tesis, se ha desarrollado en framework Web llamado Cazon para lidiar con funcionalidades volátiles en el contexto de OOHDM. Este está montado sobre Struts[114]. El framework soporta la traducción de modelos OOHDM de forma semi-automática en una aplicación y brinda el soporte necesario para gestionar las funcionalidades volátiles. Cazon permite mapear las afinidades e integración de especificación descripta en el Capítulo 8 y Capítulo 9 respectivamente en documentos XML y soporta tanto la ejecución de consultas durante la construcción de las páginas como la composición de interfaces. El framework Cazon fue iniciado en [20] brindando soporte inicial a OOHDM y afinidades. A partir de los diferentes trabajos científicos, principalmente [28] y [66], se incorporó soporte a las *Especificaciones de Integración* y *Reglas de Producción.*

En la Ilustración 73 se muestran la etapa de implementación (paso 6) donde se puede apreciar como datos requeridos los modelos definidos en las etapas anteriores de Modelado Conceptual (paso 3), Modelado Navegacional (paso 4) y Modelado de Interfaz (paso 5).

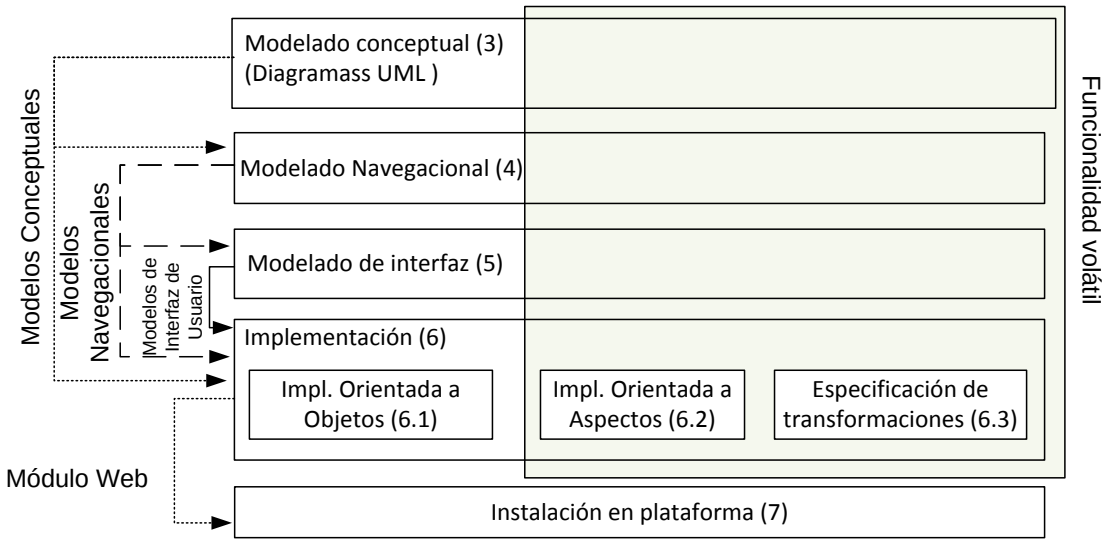


Ilustración 73 Tareas de la metodología especificas de la fase de Codificación e Instalación

Cazon básicamente decora los requerimientos HTTP procesados por Struts y respuestas generadas en el ciclo incorporando soporte a la volatilidad. Una descripción detallada del framework puede ser encontrada en [20] y [65].

Una aplicación basada en Cazon requiere la definición de un módulo OOHDM y el módulo VService. El primero lidia con la definición de constructores navegacionales (básicamente nodos y links) y puede ser utilizado con o sin el último. El último, mientras tanto, decora el módulo OOHDM y está a cargo de aumentar los nodos de la aplicación con funcionalidades volátiles de acuerdo a las afinidades navegacionales asociadas a cada uno. La Ilustración 74 describe los componentes principales mencionados.

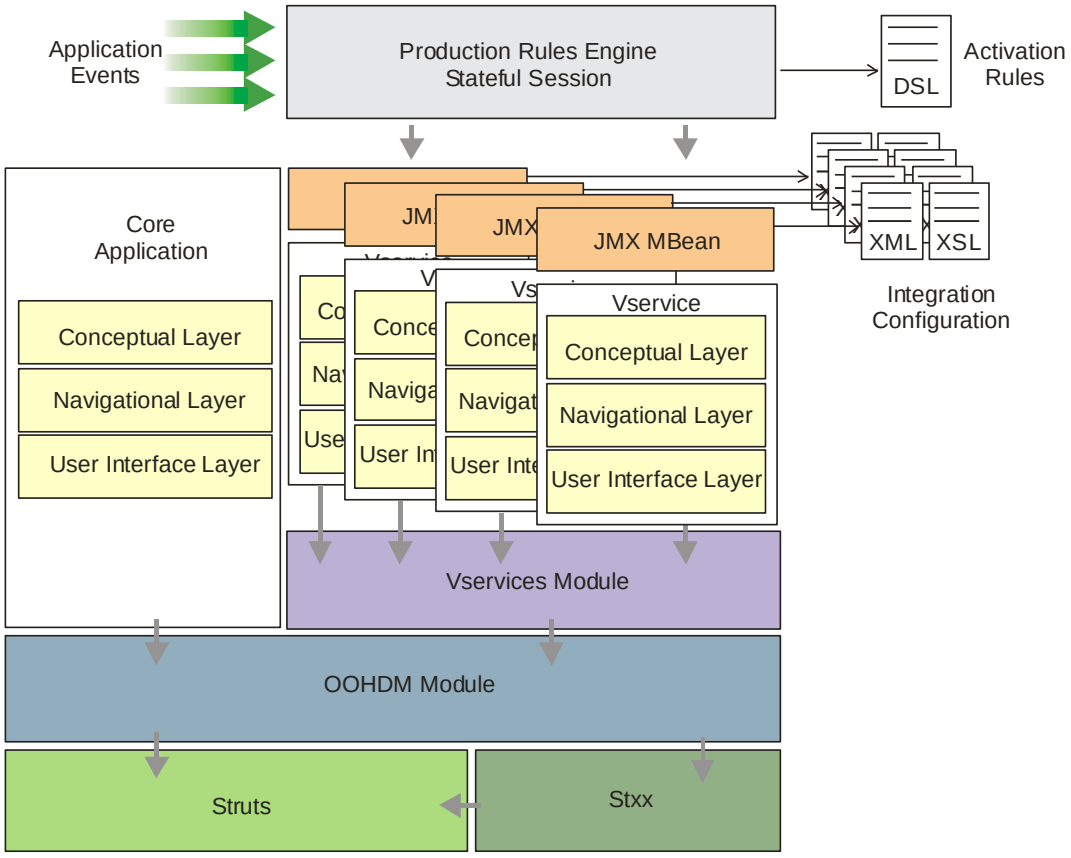


Ilustración 74 Arquitectura de Cazon, una extensión del framework Struts para funcionalidad volátil.

Cada VService es la implementación de una funcionalidad volátil y es también un modelo OOHDM. Este contiene todas las clases conceptuales, la definición de nodos y las interfaces UI que implementan el ADV volátil. Los objetos VService pueden tener referencias a algunos componentes core de la aplicación, pero el VService es transparente a la navegación y a la integración de especificación que son mantenidos en archivos de configuración diferentes.

La especificación de *afinidad* de navegación son mapeadas casi directamente en archivos XML[20]. La *Integración de Especificación* es realizada mediante transformaciones XSL. Los *pointcuts* 2D donde los *concern* volátiles son incorporados en la interfaz core son expresados utilizando expresiones XPath[108]. Una descripción completa de la implementación básica puede ser consultada en [65].

Para poder introducir una activación/desactivación en tiempo de ejecución y configuración de funcionalidades volátiles, cada VService es manejado por un JMX managed bean (Mbean)[115]. Un MBean es un objeto java que representa un recurso gestionable, tal como un comportamiento volátil. A través de la interface de MBean, todas las propiedades de una funcionalidad pueden ser indicadas en tiempo de ejecución; tal como el ciclo de vida de activación/desactivación, afinidad navegacional e integración UI. Cada VService es empaquetado separadamente dentro de un ARchivo de Servicio[116] (SAR, *Service ARchive*) para ser instalado en un servidor de aplicaciones JBoss[117].

La especificación del ciclo de vida de un VService es realizada utilizando un archivo de *Reglas de Producción* que indican las condiciones de activación/desactivación. Las reglas son interpretadas por Drools[112], un motor de reglas de código abierto con soporte de Procesamiento de Eventos Complejos (CEP).

Para facilitar la escritura de las reglas se definió un DSL que permite la especificación de éstas en casi en lenguaje natural. Las condiciones de Activación escritas en el DSL son mapeadas como patrones de eventos. Los eventos son modelados como clases simple que contienen un estado que lo identifique y represente.

Una vez que las *Reglas de Producción* son instaladas, una sesión con estado (tipo *stateful)* del motor de regla va a escuchar por el flujo de eventos de la aplicación que puede capturarse de diferentes canales. Cuando la condición de una regla es satisfecha, el motor dispara las consecuencias de la misma. En este caso, los mensajes *connect()* y *disconnect()* son invocados con el nombre de la funcionalidad, una Afinidad navegacional y la Especificación de Integración como parámetros. Finalmente, el VServiceManager ejecuta una operación sobre la interfaz VService JMX. Esta operación puede activar/desactivar una funcionalidad volátil o cambiar su patrón de integración (especificando nuevos archivos de configuración).

Las *Reglas de Producción* pueden ser cambiadas en tiempo de ejecución a través de una interfaz Web que expone la aplicación, permitiendo un cambio rápido del comportamiento de la misma según el negocio lo requiera; las reglas también son volátiles.

Cazon utiliza Stxx[118], una extensión de Struts que permite a las clases acción retornar datos en formato XML.

Para permitir la composición dinámica en tiempo de ejecución, se creó una subclase de Stxx AbstractXSLTransformer de tal forma que el método *transform*() colabora con el artefacto

VServiceManager de Cazon para obtener la lista de los servicios volátiles que tienen afinidades con el nodo actual. Por cada uno de estos servicios volátiles, su transformación de integración asociada se resuelve y aplica sobre la interfaz base. Finalmente, el resultado de la transformación es una transformación (XSL) que es aplicado a su documento XML (que ha sido aumentado previamente) dando como resultado la interfaz final.

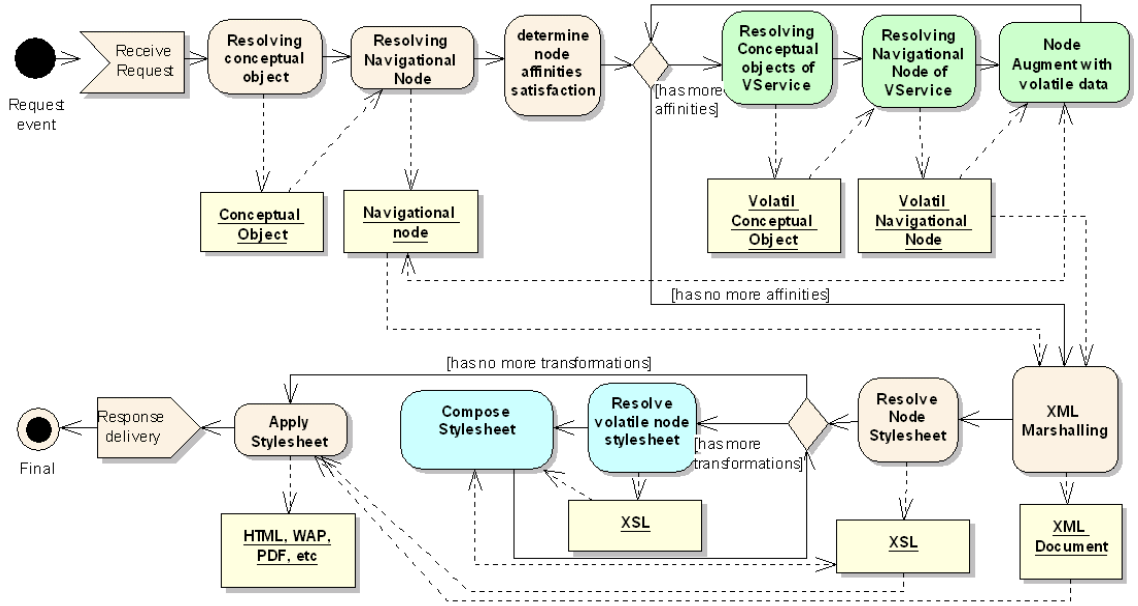


Ilustración 75 Flujo de procesamiento de un request procesado por Cazon

En Ilustración 75 se puede apreciar un diagrama de actividad UML que resumen los principales pasos del framework Cazon. En una respuesta a un requisito HTTP, el nodo resultante es identificado, mejorado y finalmente dibujado. Para llevar a cabo esta tarea, el objeto conceptual correspondiente al requisito y su nodo navegacional son resueltos. Si el nodo tiene alguna afinidad con un servicio volátil, de forma similar al caso core, se resuelven sus objetos conceptuales y nodos navegacionales para que los nodos core sean aumentado (introduciendo links) con los nodos de los respectivos servicios. Luego, la capa de presentación obtiene una representación del contenido de nodo que es procesado con una plantilla (style sheet) XSL que provee el formato visual final para los elementos XML. Si un nodo tiene afinidades con los servicios volátiles, la plantilla core asociado al nodo es compuesto con los documentos que describen las interfaces de usuario correspondiente a los nodos volátiles. Esta etapa es realizada construyendo una línea de trabajo con las transformaciones XSL obtenidas de cada servicio volátil. Cuando la plantilla core es introducida el flujo del proceso, cada transformación es aplicada sobre esta. Por lo tanto, la plantilla es sucesivamente mejorada con los elementos de las interfaces de cada *concern* volátil hasta que la versión definitiva es obtenida. Finalmente, la plantilla mejorada es aplicada sobre el XML que representa los nodos navegacionales aumentado

el contenido con el de los nodos para producir la respuesta que será enviada de vuelta al cliente que la solicitó.

En el Apéndice D podrá encontrar un ejemplo de cómo instanciar una aplicación en el framework Web Cazon y una funcionalidad volátil.

Capítulo 12 Evaluación técnica de pruebas

Esta tesis promueve la realización de funcionalidad volátil en aplicaciones Web como requerimientos claramente diferenciados de los requerimientos core y propone una estrategia para su adecuado diseño, implementación e instalación de las correspondientes funcionalidades. El enfoque también permite controlar la activación y desactivación de estas funcionalidades de acuerdo a las necesidades de negocio de la aplicación. Dado que las aplicaciones Web comprenden una diversidad de componentes y artefactos que son diseñados e implementados utilizando diferentes tipos de herramientas, el impacto de introducir las funcionalidades volátiles plantea variados problemas y desafíos dependiendo de los componentes afectados de la aplicación.

Para validar este enfoque y evaluar sus beneficios, se realizaron mediciones de distintos características introducir dos nuevos *concern*, imprevistos en la etapa de diseño, en una aplicación de comercio electrónico que es utilizada como ejemplo[15]. Los dos nuevos *concerns* son *geolocalización*, el cual brinda la posibilidad de geolocalizar los objetos mediante atributos de longitud y latitud, y un *concern comentable*, que permite a los usuarios agregar comentarios a objetos de negocios específicos. En ambos casos, se incluye una presentación adecuada de los objetos manipulados; por ejemplo utilizando un mapa.

Luego de aplicar el enfoque utilizando el framework Cazon en diferentes sistemas, podemos resumir algunas lecciones aprendidas. A continuación, se presenta un análisis de dos partes: un análisis de impacto de alto nivel donde se provee una explicación detallada de los cambios de código, y un análisis de código fuente utilizando métricas de la programación orientada a objetos para medir consecuencias sobre el código fuente de la introducción de código volátil (implementación de código fuente).

12.1 Análisis de impacto

A continuación, por cada modelo de una aplicación Web, se esbozarán los potenciales cambios que pueden ser introducidos cuando se implementan requerimientos volátiles utilizando una metodología orientada a objetos. Estos cambios serán contrastados con aquellos introducidos por el enfoque propuesto en esta tesis.

[15] El código fuente está disponible en: http://www.lifia.info.unlp.edu.ar/~murbieta/.

12.1.1 Modelo Conceptual

- *Variables de instancia y métodos volátiles*: cuando se utiliza una metodología OO para implementar funcionalidades volátiles, nuevos elementos tal como variables, relaciones, y métodos deben ser codificado dentro de las clases existentes. Sin embargo, utilizando el presentado, nuevos elementos son encapsulados aplicando el patrón *Decorator* sin modificar las clases core.

- *Clases volátiles*: En una metodología convencional OO, la introducción de clases volátiles no producen un impacto significativo en el código existente. Sin embargo, cuando es necesario remover las nuevas clases (una vez cumplido el ciclo del requerimiento volátil), la aplicación debe ser versionada nuevamente; es decir, tareas habituales tal como gestión de test de aceptación, gestión de versiones y tareas de deploy en general. En enfoque, nuevas clases son empaquetadas en componentes volátiles evitando remover código volátil luego que este se convierta en obsoleto y posteriormente innecesario.

- Comportamiento *crosscutting* volátil: Para introducir comportamiento volátil utilizando una metodología OO, el comportamiento de objetos existentes debe ser actualizando cambiando código que ya fue probado y es actualmente estable. Con el enfoque propuesto, por el contrario, los requerimientos volátiles son encapsulados utilizando conceptos de la programación orientada a aspectos (*advice* y *pointcuts*) o utilizando patrones de diseño (tal como el patrón *Decorator)*. La funcionalidad volátil enriquece el comportamiento core con diferentes estrategias agregando comportamiento antes o después de la ejecución de métodos core.

12.1.2 Modelo navegacional

- Nodos navegacionales volátiles: En metodologías convencionales, de forma similar al caso de agregar clases volátiles en la capa de aplicación, agregar nuevos nodos en la capa de navegación no produce ni código esparcido ni entrelazado; teniendo, en consecuencia, un impacto mínimo en la aplicación core. Sin embargo, desactivar una funcionalidad volátil requiere remover los nuevos nodos y modificar de nuevo la aplicación. De forma opuesta, la metodología propuesta promueve el empaquetado de nuevos nodos de tal forma que ellos sean fácilmente acoplables/desacoplables.

- Operaciones de nodos navegacionales: En frameworks Web MVC tal como Struts [114] o Spring MVC [119] donde la capa de control está basada en el patrón *Command* [26], nuevas operaciones pueden ser encapsuladas como una nueva clase, de este modo evitando la necesidad de modificar cualquier página de la aplicación. Esta solución produce el mismo impacto que la introducción de nuevas clases en el modelo conceptual (ya discutido en la capa anterior).

12.1.3 Modelo de interfaz

- *Widgets de la interfaz de usuario*: Cuando se lidia con funcionalidades volátiles con metodologías convencionales OO, las nuevas características de la interfaz son incorporadas en componentes produciendo código más complejo donde se destaca el entrelazado con funcionalidades core. En contraste, en el enfoque propuesto, nuevas características estructurales, tal como widgets o distribución visual de elementos (también conocido como layout de interfaz), son especificados utilizando transformaciones XSL y luego compuestos por un motor de transformación. Por lo tanto, la interfaz de usuario base se mantiene intacta.

- *Comportamiento RIA*: En metodologías convencionales, la declaración de comportamiento está esparcido en diferentes objetos, por ejemplo, utilizando funciones JavaScript, código definido anónimamente en Widgets, o métodos de objetos. La complejidad del código se vuelve alta debido a que la definición de la interfaz de usuario se ve afectada en varios puntos por el nuevo comportamiento rico (RIA). Alternativamente, el enfoque propuesto encapsula el comportamiento en documentos HTML y artefactos JavaScript complementados con una *Especificación de Integración* que contiene la especificación del destino de tal comportamiento y cómo éste debe ser insertado.

En resumen, uno de las mayores desventajas de utilizar metodologías convencionales (por ej. metodologías OO) para manejar requerimientos volátiles es el hecho que su ciclo de vida (activación/desactivación) tiene que ser soportado de forma ad-hoc; es decir, poniendo "a mano" la lógica asociada en el código. Esto reprime la reusabilidad y, cuando el requerimiento volátil expira, exige que la aplicación sea modificada y redesplegada nuevamente sin el requerimiento volátil. En el enfoque propuesto, la gestión del ciclo de vida de requerimientos volátiles es soportada de forma nativa por medio de reglas de negocio combinado con CEP lo que permite especificar eventos, de tiempo y negocio, responsables de habilitar y deshabilitar funcionalidades volátiles.

Cuando se utilizan metodologías tradicionales OO, el desarrollo de funcionalidad volátil no sólo comprende la tarea de codificación, sino que éste requiere el esfuerzo adicional de testear ya que las funcionalidades core afectadas deben ser verificadas dos veces: cuando la funcionalidad volátil es introducida y cuando es removida. En una metodología tradicional OO, los requerimientos volátiles introducen cambios en la funcionalidad de aplicación la aplicación Web (por ej., variables de instancia, comportamiento *crosscutting* en métodos, o comportamiento rico) requiriendo tests de regresión con el objetivo de asegurar que las funcionalidades base aún funcionan. Cuando la funcionalidad volátil pasa a ser innecesaria, su código debe ser erradicado por medio de tareas manuales propensas a errores y, por ende, una tarea posterior de testing es necesario para validar el funcionamiento. Test de integración son juegan un rol vital para verificar que los *concern* compuestos se comportan como es esperado. En cambio, siguiendo el enfoque

propuesto, sólo la activación de requerimientos volátil requieren testeo porque su desactivación lleva a la aplicación a un estado anterior que ya fue testeado.

12.2 Análisis de código fuente

Para evaluar cómo las ideas presentadas durante el transcurso de esta tesis impactan la implementación de las aplicaciones, a continuación se analizará código fuente de diferentes aplicaciones midiendo una variedad de aspectos utilizando métricas bien conocidas de la POO [120]. En este análisis de código fuente se ha enfocado en:

(i) métrica de cantidad de líneas de código (SLOC, *Source Line Of Code*), que mide el tamaño del código fuente en termino de líneas sin líneas de comentarios;

(ii) métrica de falta de cohesión (LCOM4, *Lack of COhesión Metric*), el nivel de coherencia en los elementos de un artefacto que no está relacionado a su modularización;

*(iii)*complejidad de clase, definida como el tamaño de una clase en términos de línea de sentencias;

(iv) duplicación de código, detectando sentencias de código duplicadas.

Para la analizar el código Java utilizado en los ejemplos de forma automatizada, se utilizó la herramienta de control de calidad de código Sonar[121]. Esta herramienta permite estudiar la calidad del código fuente analizando complejidad de código, diseño, reglas de codificación, duplicaciones, y potenciales bugs, entre otros. Utilizando esta herramienta, se contrastará cómo varia el código cuando se introducen funcionalidades volátiles utilizando metodologías OO con respecto al impacto de al impacto producido por el enfoque propuesto en esta tesis.

Antes de presentar los resultados del análisis, se debe remarcar que la falta de modularización de los *crosscutting concern* incrementa la complejidad de la aplicación cuando ellas evolucionan y crecen, afectando diferentes aspectos del código fuente de la aplicación. Primero se analizará cómo la aplicación reacciona a las nuevas funcionalidades volátiles cuando se utiliza una metodología OO, y luego se proveerá una breve descripción de cómo la metodología descripta a lo largo de ésta tesis mantiene los módulos simples.

La métrica SLOC puede ser utilizada como un indicador para predecir la densidad de defecto [122]. El análisis del código fuente realizado utilizando esta métrica destacó que cuando se introduce una funcionalidad volátil, tanto en metodologías convencionales OO como con la propuesta en esta tesis, la métrica SLOC incrementa. Esto no es sorprendente ya que en ambos casos las funcionalidades volátiles son introducidas por medio de nuevos objetos, y nuevos estados y comportamiento de objetos extendiendo definiciones core, que en conclusión, corresponden a líneas de código. A pesar de que le metodología presentada promueva la modularización, las extensiones propuestas deben ser implementadas y entonces su SLOC suma a

la cuenta general. A diferencia de que una metodología convencional OO, el enfoque discutido a lo largo de la tesis promueve la separación de *concern* teniendo como resultado un incremento de artefactos (clases pequeñas) donde cada aspecto es encapsulado en clases en lugar de tener una clase compleja (monolítica) tanto con funcionalidad core como volátil. Este tipo de modularización ha demostrado prevenir defectos[123].

La posibilidad de testear una aplicación se ve comprometida, como mínimo, por dos factores: SLOC se incrementa en artefactos existentes, tal como una clase, y LCOM también. Cuando una nueva sentencia de código es introducida en una clase, su complejidad incrementa demandando nuevos conjuntos de test de unidad para validar cada funcionalidad. Por otro lado, la falta de cohesión (LCOM) produce clases que encapsulan diferentes funcionalidades disjuntas, es decir que conceptualmente sería conveniente refactorizar la clase para disponer de dos artefactos o más para encapsular las diferentes funcionalidades. Este problema es conocido como la *tiranía de la descomposición dominante*[16] [31], donde no se puede modularizar *concern* que no son enmarcados dentro del criterio principal de descomposición. En algunas clases, por la imposibilidad de modularizar adecuadamente, éste problema fue detectado como un incremento de la métrica de código duplicado.

La eliminación de funcionalidades volátil requiere una tarea propensa a errores debido a su naturaleza *crosscutting* intrínseca. A pesar de que no se haya evaluado el esfuerzo de remover manualmente una funcionalidad volátil, varios trabajos han demostrado que a medida que un componente sufre cambios, más alto es el riesgo de la existencia de un bug[124]. En su lugar, utilizando el enfoque propuesto, no habría chances de introducir un error en los componentes core porque su código no es modificado en ningún momento por la incorporación de la funcionalidad volátil y en consecuencia no es necesaria la tarea de retirar código volátil.

[16] En inglés *tyranny of dominant decomposition*.

Capítulo 13 Conclusiones

A lo largo de esta tesis, se discutió con ejemplos simples pero ilustrativos la problemática generada al surgir requerimientos de forma inesperada sin un claro ciclo de vida (*patrón de volatilidad* dinámico). Tras haber motivado el problema, se propuso un enfoque de análisis de las funcionalidades volátiles para ayudar a detectar el propósito, y alcance de la misma.

Para dar una solución a este problema de funcionalidad volátil en aplicaciones Web se ha propuesto un enfoque modular dirigido por modelos que puede ser aplicado a casi cualquier metodología de ingeniería Web; en este caso se utilizó OOHDM. El eje de la tesis es el diseño simétrico de cualquier funcionalidad volátil por más simple que sea. Este concepto se propaga a las diferentes modelos de una aplicación Web (conceptual, navegacional e interfaz) y para ello se definieron herramientas conceptuales que permiten el adecuado modelado de los requerimientos volátiles de forma simétrica, transparente y sin ataduras.

Siguiendo el enfoque, las funcionalidades volátiles son detectadas en las etapas tempranas del desarrollo de Software para identificar conflictos con otros requerimientos core y proponer soluciones de forma automática o semi-automática. Es un novedoso aporte que mediante la formalización de requerimientos de interacción utilizando WebSpec se pueden detectar conflictos (de estructura y de navegación) a nivel sintáctico y semántico. Para detectar conflictos e inconsistencias los requerimientos de en aplicaciones, cada nuevo requerimiento volátil es validado contra los requerimientos consolidados (que ya demostraron mantener la consistencia). Esta herramienta conceptual permite ahorrar tiempo y esfuerzo en detectar y resolver errores en las etapas tardías del desarrollo de Software. Se desarrollo un prototipo de herramienta CASE que automatiza el análisis y corrección de inconsistencias.

Posteriormente, se procede al diseño del modelo conceptual donde los objetos de negocio son modelados describiendo su estructura y comportamiento utilizando diagramas de UML según establece OOHDM. Los requerimientos volátiles son diseñados con la misma prioridad que los requerimientos core. Utilizando MATA y Pattern Specification se modelan la introducción de estado y cambio de comportamiento de objetos.

Como siguiente paso, se diseña cómo un *concern* volátil altera la forma en que la aplicación es navegada. Definiendo las *afinidades* como herramienta conceptual, se logra especificar la relación de los modelos navegacionales core y volátiles.

Como último paso de la etapa de diseño, los cambios de estructura de las interfaces de usuario y su comportamiento son diseñados con ADV y ADV-charts respectivamente. Los cambios se describen de forma transparentes y sin introducir cambios a los modelos core. Para describir

cómo se introducen cambios estructurales en modelos de interfaz de usuario, se definió el concepto de *Especificación de Integración* que establece la interfaz core donde incorporar, cómo será la incorporación y qué elementos de interfaz se introducirán. En relación a los cambios de comportamiento de interfaz se definió una extensión a los ADV-Charts para describir cambios de comportamiento en interfaces core introducidos por requerimientos volátiles. La integración de interfaces fue solucionada utilizando transformaciones de documentos XML. Utilizando una sintaxis simple, se permite indicar la forma en la que el diseño de las interfaces son compuesta y utilizando transformaciones XSL se es capaz de componer los documentos XML. De esta forma, no es necesario invadir y contaminar el modelo y código de implementación con referencias a componentes volátiles que pueden ser eliminados sin previo aviso (requiriendo una nueva edición de diseño y código).

Los cambios producidos por la funcionalidad volátil en las tres capas (conceptual, navegacional, y de interfaz) son expresados de tal forma que los modelos core no son invadidos de ninguna forma y por ende no son modificados.

La metodología propone una solución para cubrir el ciclo completo de una funcionalidad volátil desde el diseño a la implementación, hasta la gestión del tiempo de ejecución (activación/desactivación, de acuerdo al patrón de volatilidad). La metodología hace posible incorporar funcionalidad volátil en aplicaciones Web "en el aire"(on the fly), y habilita a personas no técnicas controlar su activación mediante reglas en tiempo de ejecución mejorando la adaptabilidad de la aplicación.

Como parte de las investigaciones realizadas a lo largo de esta tesis se continuó con el desarrollo del framework Cazon basado en OOHDM; iniciado en [20] para incorporar además, del soporte original a las *afinidades*, el soporte a *Especificación de Integración* y *Reglas de Producción.*

Con estas extensiones se pueden desarrollar aplicaciones con soporte a funcionalidades volátiles donde cada una de ella es instalada en tiempo de ejecución sin requerir la modificación de la aplicación (conjunto de *concern*) base que será modificado por los requerimientos volátiles instalados; permitiendo además una evolución independiente de cada uno. Cazon automatiza la composición dinámica de la funcionalidad volátil dentro de los módulos core en la tres capas (conceptual, navegacional y de interfaz).

Por último, en esta tesis se discutió las mejoras detectadas al realizar separación de *concern* estudiando el código fuente de diferentes aplicaciones en base a métricas OOP.

Capítulo 14 Trabajos futuros

14.1 En validación de requerimientos

Actualmente estoy colaborando con un grupo de investigación en España para realizar experimentos utilizando este enfoque pero utilizando como meta-modelo de requerimientos NDT en lugar de utilizar el de WebSpec.

Por otro lado se está trabajando para:

- completar el enfoque de detección de inconsistencias con un conjunto de algoritmos de *ontology matching* para mejorar la detección de conflictos semánticos;
- extender el conjunto de heurísticas disponibles para resolver conflictos detectados;
- llevar a cabo experimentos de utilización para poder analizar y reportar evidencias y métricas de tiempo y esfuerzo efectivamente ahorro;
- y por último, analizar conflictos de navegación complejos. Muchas veces no es suficiente pensar en los aspectos estructurales del modelo sino que debemos contemplar el conjunto de datos que contextualizan la navegación ya que bajo diferentes conjuntos de datos ambas navegaciones serían válidas.

14.2 Metodología

Una línea de investigación pendiente está relacionada con el estudio de la correcta composición de *concerns* tal como en [125] para la composición de *Features* para garantizar la correctitud de la aplicación resultante.

Un lenguaje para describir dependencias (restricciones) de un *concern* así como los puntos de extensión esperados en los diferentes modelos (conceptual, navegacional y modelos de interfaz). La validación de modelos y control de errores debe ser automatizado para facilitar las tareas del diseñador.

Para validar la flexibilidad del enfoque sería conveniente su integración con otros métodos de ingeniería Web dirigidas por modelo y particularmente analizar la integración a nivel de meta modelos. Mediante el análisis de las ideas existentes para adaptar o unificar métodos[17] se puede encontrar una forma de expresar volatilidad en un nivel de abstracción más alto.

Finalmente, se debe realizar evaluaciones de experiencias en el desarrollo de aplicaciones Web para obtener retorno (feedback) del enfoque desarrollado en esta tesis.

14.3 Interfaces de Usuario convencionales

Actualmente, estoy participando en varios tópicos de investigación. En primer lugar, se está diseñando una herramienta para automatizar la traducción de archivos XSL en ADVs, y especificación de *pointcuts* en transformaciones XSL para mejorar el soporte dirigido a modelos en Cazon. Por otro lado, como línea alternativa de investigación, se abre la oportunidad de estudiar y mejorar el lenguaje de especificación para soportar *crosscutting concern* más complejo. En segundo lugar, el orden en el que se produce de composición y el resultado obtenido es una línea de trabajo obligada y el impacto en la apariencia de la aplicación (*look and feel*) es vital.

14.4 Herramienta WebSpec

En la Sección 7.2 se presentó una matriz de *crosscutting* utilizada para asistir a los analistas en la detección de crosscuting *concern* y llamando la atención del analista ante un posible caso que requiera el diseño de aspectos. Esta tabla de *crosscutting concern*, fue inspirada en [87] en donde la granularidad del *crosscutting* es más fina ya que se cruzan *concern* con Navigational Units en lugar de *concern* contra diagramas WebSpec; como se realiza en esta metodología. Esto se propuso de esta forma ya que utilizar la propuesta de [87] requería demasiado esfuerzo en desarrollar la matriz a tal punto que resulta impráctico. Sin embargo, esta funcionalidad podría fácilmente incluirse en la herramienta CASE de WebSpec. Se puede destacar que la utilización de la matriz permitirá detectar roles[126] y temas[37] de forma temprana tal como fue discutido en [87].

Capítulo 15 Referencias

[1] «Internet Access - Households and Individuals, 2011». [Online]. Available: http://www.ons.gov.uk/ons/rel/rdit2/internet-access---households-and-individuals/2011/index.html. [Accessed: 29-feb-2012].

[2] «World Internet Usage Statistics News and World Population Stats». [Online]. Available: http://www.internetworldstats.com/stats.htm. [Accessed: 29-feb-2012].

[3] «CoreHouseholdIndicators.xls». [Online]. Available: http://www.itu.int/ITU-D/ict/statistics/material/excel/2010/CoreHouseholdIndicators.xls. [Accessed: 29-feb-2012].

[4] G. Schneider, *Electronic Commerce*. Cengage Learning, 2008.

[5] A. Moreira, J. Araújo, y J. Whittle, «Modeling Volatile Concerns as Aspects», in *Advanced Information Systems Engineering*, vol. 4001, E. Dubois y K. Pohl, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 544–558.

[6] «Amazon.com: Online Shopping for Electronics, Apparel, Computers, Books, DVDs & more». [Online]. Available: http://www.amazon.com/. [Accessed: 04-mar-2012].

[7] S. Aronoff, *Geographic Information Systems: A Management Perspective*. Wdl Pubns, 1991.

[8] P. A. Longley, M. Goodchild, D. J. Maguire, y D. W. Rhind, *Geographic Information Systems and Science*, 3.ª ed. Wiley, 2010.

[9] L. Jordan, *Blogger: Beyond the Basics: Customize and promote your blog with original templates, analytics, advertising, and SEO*. Packt Publishing, 2008.

[10] «lanacion.com - Las noticias que importan y los temas que interesan». [Online]. Available: http://www.lanacion.com.ar/. [Accessed: 29-feb-2012].

[11] J. Lowe, *Google speaks: secrets of the world's greatest billionaire entrepreneurs, Sergey Brin and Larry Page*. John Wiley and Sons, 2009.

[12] «What Is Web 2.0 - O'Reilly Media - Perpetual Beta». [Online]. Available: http://oreilly.com/web2/archive/what-is-web-20.html?page=4. [Accessed: 15-mar-2012].

[13] D. Moore, R. Budd, y E. Benson, *Professional Rich Internet Applications: AJAX and Beyond*. John Wiley & Sons, 2007.

[14] J. Schiller y A. Voisard, Eds., *Location-Based Services*, 1.ª ed. Morgan Kaufmann, 2004.

[15] M. T. Goodrich y R. Tamassia, *Data Structures and Algorithms in Java*, 5.ª ed. Wiley, 2010.

[16] M. Fowler y K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[17] G. Rossi, O. Pastor, D. Schwabe, y L. Olsina, *Web Engineering: Modelling and Implementing Web Applications (Human-Computer Interaction Series)*, 1.ª ed. .

[18] D. Schwabe y G. Rossi, «An object oriented approach to Web-based applications design», *Theor. Pract. Object Syst.*, vol. 4, nº. 4, pp. 207–225, oct. 1998.

[19] D. A. Nunes y D. Schwabe, «Rapid prototyping of web applications combining domain specific languages and model driven design», in *Proceedings of the 6th international conference on Web engineering*, New York, NY, USA, 2006, pp. 153–160.

[20] G. Rossi, A. Nieto, L. Mengoni, N. Lofeudo, L. N. Silva, y D. Distante, «Model-Based Design of Volatile Functionality in Web Applications», in *Proceedings of the Fourth Latin American Web Congress*, Washington, DC, USA, 2006, pp. 179–188.

[21] D. Schwabe, G. Szundy, S. S. de Moura, y F. Lima, «Design and Implementation of Semantic Web Applications», in *WWW Workshop on Application Design, Development and Implementation Issues in the Semantic Web*, 2004.

[22] «RDF Vocabulary Description Language 1.0: RDF Schema». [Online]. Available: http://www.w3.org/TR/rdf-schema/. [Accessed: 06-mar-2012].

[23] «The Internet Movie Database (IMDb)». [Online]. Available: http://www.imdb.com/. [Accessed: 19-mar-2012].

[24] P. Vilain, D. Schwabe, y C. S. de Souza, «A diagrammatic tool for representing user interaction in UML», in *Proceedings of the 3rd international conference on The unified modeling language: advancing the standard*, Berlin, Heidelberg, 2000, pp. 133–147.

[25] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3.ª ed. Addison-Wesley Professional, 2003.

[26] E. Gamma, R. Helm, R. Johnson, y J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1.ª ed. Addison-Wesley Professional, 1994.

[27] «Google Web Toolkit - Google Code». [Online]. Available: http://code.google.com/webtoolkit/. [Accessed: 19-mar-2012].

[28] J. Ginzburg, G. Rossi, M. Urbieta, y D. Distante, «Transparent Interface Composition in Web Applications», presented at the ICWE, 2007, pp. 152–166.

[29] «Object Management Group - UML». [Online]. Available: http://www.uml.org/. [Accessed: 07-mar-2012].

[30] «Ruby on Rails». [Online]. Available: http://rubyonrails.org/. [Accessed: 19-mar-2012].

[31] E. R. Luna, I. Garrigós, J. Grigera, y M. Winckler, «Capture and evolution of web requirements using webspec», in *Proceedings of the 10th international conference on Web engineering*, Vienna, Austria, 2010, pp. 173–188.

[32] M. Hitz y B. Montazeri, «Measuring coupling and cohesion in object-oriented systems», in *Proc. Intl. Sym. on Applied Corporate Computing*, 1995.

[33] E. Freeman, E. Freeman, K. Sierra, y B. Bates, *Head First design patterns*. O'Reilly Media, Inc., 2004.

[34] «Inversion of Control Containers and the Dependency Injection pattern». [Online]. Available: http://martinfowler.com/articles/injection.html. [Accessed: 20-mar-2012].

[35] R. Laddad, *Aspectj in Action: Practical Aspect-Oriented Programming*. Manning Publications, 2003.

[36] R. E. Filman, T. Elrad, S. Clarke, y M. Aksit, *Aspect-Oriented Software Development*, 1.ª ed. Addison-Wesley Professional, 2004.

[37] S. Clarke y E. Baniassad, *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley Professional, 2005.

[38] «Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design». [Online]. Available: http://early-aspects.net/. [Accessed: 08-mar-2012].

[39] I. Jacobson y P.-W. Ng, *Aspect-Oriented Software Development with Use Cases*, 1.ª ed. Addison-Wesley Professional, 2005.

[40] R. B. France, Dae-Kyoo Kim, S. Ghosh, y Eunjee Song, «A UML-based pattern specification technique», *IEEE Transactions on Software Engineering*, vol. 30, nº. 3, pp. 193–206, mar. 2004.

[41] G. P. Kulk y C. Verhoef, «Quantifying requirements volatility effects», *Sci. Comput. Program.*, vol. 72, nº. 3, pp. 136–175, ago. 2008.

[42] M. Gaedke y G. Gräf, «Development and Evolution of Web-Applications Using the WebComposition Process Model», in *Web Engineering*, vol. 2016, S. Murugesan y Y. Deshpande, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 58–76.

[43] M. Bebjak, V. Vranic, y P. Dolog, «Evolution of Web Applications with Aspect-Oriented Design Patterns», in *AEWSE*, 2007.

[44] M. Niederhausen, K. Sluijs, J. Hidders, E. Leonardi, G.-J. Houben, y K. Meißner, «Harnessing the Power of Semantics-Based, Aspect-Oriented Adaptation for amacont», in *Proceedings of the 9th International Conference on Web Engineering*, Berlin, Heidelberg, 2009, pp. 106–120.

[45] H. Baumeister, N. Koch, y G. Zhang, «Modelling Adaptivity with Aspects», *INTERNATIONAL CONFERENCE ON WEB ENGINEERING (ICWE 2005*, pp. 406–416, 2005.

[46] S. Casteleyn, W. Van Woensel, y G.-J. Houben, «A semantics-based aspect-oriented approach to adaptation in web engineering», in *Proceedings of the eighteenth conference on Hypertext and hypermedia*, New York, NY, USA, 2007, pp. 189–198.

[47] I. Garrigós, S. Meliá, y S. Casteleyn, «Personalizing the Interface in Rich Internet Applications», in *Proceedings of the 10th International Conference on Web Information Systems Engineering*, Berlin, Heidelberg, 2009, pp. 365–378.

[48] A. Kraus y N. Koch, «Generation of Web Applications from UML Design Models using an XML Publishing Framework», presented at the Integrated Design and Process Technology Conference (IDPT'2002), 2002.

[49] «Cocoon Main Site - Welcome». [Online]. Available: http://cocoon.apache.org/. [Accessed: 22-feb-2012].

[50] «Feature-Oriented Software Development Research». [Online]. Available: http://fosd.de/. [Accessed: 08-mar-2012].

[51] S. Apel, C. Lengauer, B. Möller, y C. Kästner, «An algebraic foundation for automatic feature-based program synthesis», *Sci. Comput. Program.*, vol. 75, nº. 11, pp. 1022–1047, nov. 2010.

[52] S. Trujillo, D. Batory, y O. Diaz, «Feature Oriented Model Driven Development: A Case Study for Portlets», in *29th International Conference on Software Engineering, 2007. ICSE 2007*, 2007, pp. 44–53.

[53] W. C. Richardson, D. Avondolio, J. Vitale, P. Len, K. T. Smith, y K. B. Smith, *Professional Portal Development with Open Source Tools: JavaPortlet API, Lucene, James, Slide*. John Wiley & Sons, 2004.

[54] J. Dingel, Z. Diskin, y A. Zito, «Understanding and improving UML package merge», *Software & Systems Modeling*, vol. 7, nº. 4, pp. 443–467, dic. 2007.

[55] T. Nash, *Accelerated C# 2010*. Apress, 2009.

[56] «Fuji: A Compiler for Feature-Oriented Programming in Java». [Online]. Available: http://fosd.de/fuji. [Accessed: 22-feb-2012].

[57] S. Lepreux, J. V, y erdonckt, «Towards A Support Of User Interface Design By Composition Rules», presented at the CADUI, 2006, pp. 231–244.

[58] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, y V. López-Jaquero, «USIXML: A Language Supporting Multi-path Development of User Interfaces», in *Engineering Human Computer Interaction and Interactive Systems*, vol. 3425, R. Bastide, P. Palanque, y J. Roth, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 200–220.

[59] «[Part 3] Assets, Atom Feeds, and AspectXML - The Triple Threat of Web Development? - O'Reilly XML Blog». [Online]. Available: http://www.oreillynet.com/xml/blog/2005/09/part_3_assets_atom_feeds_and_a.html. [Accessed: 23-feb-2012].

[60] T. Cohen y J. (Yossi) Gil, «AspectJ2EE = AOP J2EE: Towards An Aspect Based, Programmable and Extensible Middleware Framework», *IN PROC. ECOOP '04, VOLUME 3086 OF LNCS*, pp. 14–18, 2004.

[61] I. Garrigós, C. Cruz, y J. Gómez, «A Prototype Tool for the Automatic Generation of Adaptive Websites», *AEWSE*, vol. 267, 2007.

[62] J. Gomez, C. Cachero, y O. Pastor, «Conceptual modeling of device-independent Web applications», *IEEE Multimedia*, vol. 8, nº. 2, pp. 26–39, jun. 2001.

[63] F. Daniel, M. Matera, R. Mor, y M. Mortari, «Active Rules for Runtime Adaptivity Management».

[64] M. Gaedke y G. Gräf, «Development and Evolution of Web-Applications Using the WebComposition Process Model», in *Web Engineering, Software Engineering and Web Application Development*, 2001, pp. 58–76.

[65] J. Ginzburg, D. Distante, G. Rossi, y M. Urbieta, «Oblivious integration of volatile functionality in web application interfaces», *J. Web Eng.*, vol. 8, nº. 1, pp. 25–47, mar. 2009.

[66] M. Urbieta, G. Rossi, D. Distante, y J. Ginzburg, «Modeling, Deploying, and Controlling Volatile Functionalities in Web Applications», *IJSEKE*, vol. 22, nº. 1, pp. 129–155, 2012.

[67] C. Pons, R. Giandini, y G. Pérez, *Desarrollo de Software dirigido por modelos*. McGraw-Hill Education Y Edulp, 2010.

[68] P. Vora, *Web application design patterns*. Morgan Kaufmann, 2009.

[69] M. Fowler, *Domain-Specific Languages (Addison-Wesley Signature Series*, 1.ª ed. Addison-Wesley Professional, 2010.

[70] O. Pastor, J. Fons, V. Pelechano, y S. Abrahão, «Conceptual Modelling of Web Applications: The OOWS Approach», in *Web Engineering*, E. Mendes y N. Mosley, Eds. Berlin/Heidelberg: Springer-Verlag, pp. 277–302.

[71] D. Distante, G. Rossi, G. Canfora, y S. Tilley, «A comprehensive design model for integrating business processes in web applications», *International Journal of Web Engineering and Technology*, vol. 3, nº. 1, p. 43, 2007.

[72] M. Cohn, *Succeeding with Agile: Software Development Using Scrum*, 1.ª ed. Addison-Wesley Professional, 2009.

[73] G. Kotonya y I. Sommerville, «Requirements engineering with viewpoints», *Software Engineering Journal*, vol. 11, nº. 1, pp. 5–18, ene. 1996.

[74] D. Yang, Q. Wang, M. Li, Y. Yang, K. Ye, y J. Du, «A survey on software cost estimation in the chinese software industry», in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, Kaiserslautern, Germany, 2008, pp. 253–262.

[75] D. Leffingwell, «Calculating the Return on Investment from More Effective Requirements Management», *American Programmer*, vol. 10, nº. 4, pp. 13–16, 1997.

[76] M. J. Escalona y N. Koch, «Metamodeling the Requirements of Web Systems», in *Web Information Systems and Technologies*, vol. 1, J. Filipe, J. Cordeiro, y V. Pedrosa, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 267–280.

[77] M. G. de Paula, B. S. da Silva, y S. D. J. Barbosa, «Using an interaction model as a resource for communication in design», in *CHI '05 extended abstracts on Human factors in computing systems*, Portland, OR, USA, 2005, pp. 1713–1716.

[78] «IEEE Recommended Practice for Software Requirements Specifications». 1998.

[79] J. Wood y D. Silver, *Joint Application Development*, 2.ª ed. Wiley, 1995.

[80] P. Kroll y P. Kruchten, *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*, 1.ª ed. Addison-Wesley Professional, 2003.

[81] M. Fowler, *Domain-Specific Languages (Addison-Wesley Signature Series*, 1.ª ed. Addison-Wesley Professional, 2010.

[82] «webspec-language - A DSL for Web requirements specification - Google Project Hosting». [Online]. Available: http://code.google.com/p/webspec-language/. [Accessed: 06-jul-2011].

[83] «OCL». [Online]. Available: http://www.omg.org/spec/OCL/. [Accessed: 06-jul-2011].

[84] K. Altmanninger, «Models in conflict – towards a semantically enhanced version control system for models», *PROC. OF THE MODELS 2007 DOCTORAL SYMPOSIUM. (2007*.

[85] C. Li y T. W. Ling, «OWL-Based Semantic Conflicts Detection and Resolution for Data Interoperability», in *Conceptual Modeling for Advanced Application Domains*, vol. 3289, S. Wang, K. Tanaka, S. Zhou, T.-W. Ling, J. Guan, D. Yang, F. Grandi, E. E. Mangina, I.-Y. Song, y H. C. Mayr, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 266–277.

[86] B. Scott y T. Neil, *Designing Web interfaces*. O'Reilly Media, Inc., 2009.

[87] Silvia Gordillo, Gustavo Rossi, Ana Moreira, Joao Araujo, Carla Vairetti, y Matias Urbieta, «Modeling and Composing Navigational Concerns in Web Applications. Requirements and Design Issues.», in *Web Congress, 2006. LA-Web '06. Fourth Latin American*, 2006, vol. 0, pp. 25–31.

[88] J. M. Conejero, J. Hernández, A. Moreira, y J. Araújo, «Adapting Software by Identifying Volatile and Aspectual Requirements», in *Jornadas de Ingeniería del Software y Bases de Datos*, 2009, pp. 103–114.

[89] «Pyroute - OpenStreetMap Wiki». [Online]. Available: http://wiki.openstreetmap.org/wiki/Pyroute. [Accessed: 19-feb-2012].

[90] «OpenStreetMap», 19-feb-2012. [Online]. Available: http://www.openstreetmap.org/. [Accessed: 19-feb-2012].

[91] H. A. Schmid y G. Rossi, «Modeling and Designing Processes in E-Commerce Applications», *IEEE Internet Computing*, vol. 8, nº. 1, pp. 19–27, ene. 2004.

[92] «Defining N-ary Relations on the Semantic Web: Use With Individuals». [Online]. Available: http://www.w3.org/TR/2004/WD-swbp-n-aryRelations-20040721/#pattern2. [Accessed: 11-mar-2012].

[93] «Object-built ins - GNU Smalltalk Library Reference». [Online]. Available: http://www.gnu.org/software/smalltalk/manual-base/html_node/Object_002dbuilt-ins.html. [Accessed: 11-mar-2012].

[94] M. Nanard, J. Nanard, y P. King, «IUHM: a hypermedia-based model for integrating open services, data and metadata», in *Proceedings of the fourteenth ACM conference on Hypertext and hypermedia*, New York, NY, USA, 2003, pp. 128–137.

[95] G. Rossi, H. A. Schmid, y F. Lyardet, «Customizing Business Processes in Web Applications», in *E-Commerce and Web Technologies*, vol. 2738, K. Bauknecht, A. M. Tjoa, y G. Quirchmayr, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 359–368.

[96] «JavaServer Pages Technology». [Online]. Available: http://www.oracle.com/technetwork/java/javaee/jsp/index.html. [Accessed: 26-feb-2012].

[97] «JavaServer Faces Technology». [Online]. Available: http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html. [Accessed: 26-feb-2012].

[98] C. Phanouriou, «UIML: A Device-Independent User Interface Markup Language», Ph. D. Thesis, Virginia University, 2000.

[99] «JavaServer Pages(TM) v1.2 Syntax Reference - Include sentence». [Online]. Available: http://java.sun.com/products/jsp/tags/12/syntaxref1214.html. [Accessed: 26-feb-2012].

[100] D. D. Cowan y C. J. P. Lucena, «Abstract data views: an interface specification concept to enhance design for reuse», *IEEE Transactions on Software Engineering*, vol. 21, nº. 3, pp. 229–243, mar. 1995.

[101] M. Urbieta, G. Rossi, J. Ginzburg, y D. Schwabe, «Designing the Interface of Rich Internet Applications», in *Web Conference, 2007. LA-WEB 2007. Latin American*, 2007, pp. 144–153.

[102] «OGNL - Wikipedia, the free encyclopedia». .

[103] «Ajax: A New Approach to Web Applications - Adaptive Path». [Online]. Available: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications. [Accessed: 27-feb-2012].

[104] «OpenLaszlo | the premier platform for rich internet applications». [Online]. Available: http://www.openlaszlo.org/. [Accessed: 27-feb-2012].

[105] «Aspect Oriented Programming and javascript». [Online]. Available: http://www.dotvoid.com/2005/06/aspect-oriented-programming-and-javascript/. [Accessed: 27-feb-2012].

[106] «jquery-aop - Aspect Oriented Extensions for jQuery - Google Project Hosting». [Online]. Available: http://code.google.com/p/jquery-aop/. [Accessed: 27-feb-2012].

[107] «XSL Transformations (XSLT)». [Online]. Available: http://www.w3.org/TR/xslt. [Accessed: 27-feb-2012].

[108] «XML Path Language (XPath)». [Online]. Available: http://www.w3.org/TR/xpath/. [Accessed: 27-feb-2012].

[109] «Xalan-Java Version 2.7.1». [Online]. Available: http://xml.apache.org/xalan-j/. [Accessed: 27-feb-2012].

[110] «The SAXON XSLT and XQuery Processor». [Online]. Available: http://saxon.sourceforge.net/. [Accessed: 27-feb-2012].

[111] «OpenLayers: Home». [Online]. Available: http://openlayers.org/. [Accessed: 29-feb-2012].

[112] «Drools - JBoss Community». [Online]. Available: http://www.jboss.org/drools. [Accessed: 28-feb-2012].

[113] D. Zimmer y R. Unland, «On the semantics of complex events in active database management systems», in *, 15th International Conference on Data Engineering, 1999. Proceedings*, 1999, pp. 392–399.

[114] «Apache Struts project». [Online]. Available: http://struts.apache.org/. [Accessed: 28-feb-2012].

[115] «Java Management Extensions (JMX)». [Online]. Available: http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html. [Accessed: 28-feb-2012].

[116] «JBossService | JBoss Community». [Online]. Available: https://community.jboss.org/wiki/JBossService. [Accessed: 27-mar-2012].

[117] «JBoss Application Server 7 - JBoss Community». [Online]. Available: http://www.jboss.org/jbossas. [Accessed: 28-feb-2012].

[118] «Struts for Transforming XML with XSL (stxx)». [Online]. Available: http://stxx.sourceforge.net/. [Accessed: 29-feb-2012].

[119] «15. Web MVC framework». [Online]. Available: http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/mvc.html. [Accessed: 02-mar-2012].

[120] S. R. Chidamber y C. F. Kemerer, «A Metrics Suite for Object Oriented Design», *IEEE Trans. Softw. Eng.*, vol. 20, nº. 6, pp. 476–493, jun. 1994.

[121] «Sonar». [Online]. Available: http://www.sonarsource.org/. [Accessed: 02-mar-2012].

[122] C. Rahmani y D. Khazanchi, «A Study on Defect Density of Open Source Software», in *2010 IEEE/ACIS 9th International Conference on Computer and Information Science (ICIS)*, 2010, pp. 679–683.

[123] K. El Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, y S. N. Rai, «The optimal class size for object-oriented software», *IEEE Transactions on Software Engineering*, vol. 28, nº. 5, pp. 494–509, may 2002.

[124] A. E. Hassan, «Predicting faults using the complexity of code changes», in *Proceedings of the 31st International Conference on Software Engineering*, Washington, DC, USA, 2009, pp. 78–88.

[125] M. Kuhlemann, D. Batory, y C. Kästner, «Safe composition of non-monotonic features», in *Proceedings of the eighth international conference on Generative programming and component engineering*, New York, NY, USA, 2009, pp. 177–186.

[126] F. Steimann, «On the representation of roles in object-oriented and conceptual modelling», *Data Knowl. Eng.*, vol. 35, nº. 1, pp. 83–106, oct. 2000.

[127] G. Rossi, M. Urbieta, J. Ginzburg, D. Distante, y A. Garrido, «Refactoring to Rich Internet Applications. A Model-Driven Approach», in *International Conference on Web Engineering*, 2008, pp. 1–12.

[128] S. Murugesan, Ed., *Handbook of Research on Web 2.0, 3.0, and X.0*. IGI Global, 2009.

[129] A. Oliveira, M. Urbieta, J. Araújo, A. Rodrigues, A. Moreira, S. E. Gordillo, y G. Rossi, «Modelling Location-aware Behaviour in Web-GIS using Aspects», in *ICEIS (3)*, 2009, pp. 416–419.

[130] A. Oliveira, M. Urbieta, J. Araújo, A. Rodrigues, A. Moreira, S. E. Gordillo, y G. Rossi, «Improving the Quality of Web-GIS Modularity Using Aspects», in *QUATIC*, 2010, pp. 132–141.

[131] M. Urbieta, M. J. Escalona, E. Robles Luna, y G. Rossi, «Detecting Conflicts and Inconsistencies in Web Application Requirements», in *Current Trends in Web Engineering*, vol. 7059, A. Harth y N. Koch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 278–288.

[132] S. Firmenich, G. Rossi, M. Urbieta, S. Gordillo, C. Challiol, J. Nanard, M. Nanard, y J. Araujo, «Engineering concern-sensitive navigation structures, concepts, tools and examples», *J. Web Eng.*, vol. 9, n°. 2, pp. 157–185, jun. 2010.

[133] «jQuery: The Write Less, Do More, JavaScript Library». [Online]. Available: http://jquery.com/. [Accessed: 29-mar-2012].

Apéndice A. Abreviaturas

A continuación se enumeran las abreviaturas utilizadas durante la tesis:

Acrónimo	Descripción
ADO	Objeto de Dato abstracto (*Abstract Data Object*)
ADV	Vista de Datos Abstracta (*Abstract Data View*)
A-OOH	Hipermedia Orientada a Objetos Adaptativa (*Adaptive Object Oriented Hypermedia*)
AOP	Programación Orientada a Aspectos (*Aspect-Oriented Programming*)
AOSD	Diseño de Software Orientado a Aspectos (*Aspect-Oriented Software Design*)
CD	Disco Compacto (*Compact Disc*)
CEP	Procesamiento de Eventos Complejos (*Complex Event Processing*)
DI	Inyección de Dependencias (*Dependency Injection*).
DSL	Lenguaje Especifico de Dominio (*Domain Specific Language*)
ECA	Evento Condición Acción (*Event-Condition-Action*)
EJB	Objeto Java de Negocio (*Enterprise Java Bean*)

FOMDD	Diseño Dirigido por Modelos Orientado a Funcionalidades (*Feature-Oriented Model Driven Development*)
FOP	Programación Orientada a Funcionalidades [17](*Feature Oriented Programming*)
GIS	Sistemas de Información Geográfica (*Geographic Information System*)
IoC	Inversión de Control (*Inversion of Control*)
J2EE	Edición Enpresarial Java 2 (*Java 2 Enterprise Edition*)
LCOM	Falta de Cohesión en Métodos (*Lack of COhesion in Methods*)
LHS	Lado Izquierdo (*Left-Hand Side*)
MATA	Modelado de Aspectos usando un Método de Transformaciones (*Modeling Aspects using a Transformation Approach*)
MDD	Diseño Dirigido por Modelos (*Model-Driven Design*)
OO	Orientado a Objetos (*Object-Oriented*)
OOH	Hipermedia Orientada a Objetos (*Object-Oriented Hypermedia*)
OOHDM	Metodología de Diseño de Hipermedia Orientado a Objetos (*Object-Oriented Hypermedia Design Method*)

[17] Al no haber encontrado una adecuada traducción a este concepto, lo he definido de esta forma para debido a que me la traducción de *feature* no es representativo en español a la idea de la metodología.

OOP	Programación Orientada a Objetos (*Object-Oriented Programming*)
OOWS	Solución Web Orientada a Objetos (*Object Oriented Web Solution*)
PS	Especificación de Patrón (*Pattern Specification*)
RHS	Lado Derecho (*Right-Hand Side*)
RIA	Aplicaciones Ricas de Internet (*Rich Internet Applications*)
SLOC	Líneas de Código Fuente (*Source Lines Of Code*)
SRS	Especificación de Requerimientos de Software (*Software Requirement Specification*)
UID	Diagrama de Interacción de Usuario (*User Interaction Diagram*)
UML	Lenguaje de Modelado Unificado (*Unified Modelling Language*)
UWE	Ingeniería Web basada en UML (*UML-based Web Engineering*)
WebML	Lenguaje de Modelado Web (*Web Modelling Language*)
XSL	Lenguaje de Hoja de Estilo Extensible (*EXtensible Stylesheet Language*)
XSLT	Transformación de Lenguaje de Hoja de Estilo Extensible (*EXtensible Stylesheet Language Transformation*)

Apéndice B. Producción científica

La tesis resume los resultados obtenidos en diferentes trabajos de investigación realizados sobre el tópico. Estos trabajos fueron presentados tanto en conferencias con foco en Ingeniería Web como en revistas (journals) científicas también especializadas en el foco. En ambos casos, siempre fueron ámbitos académicos con referato.

A continuación se presenta la lista de las publicaciones realizadas:

Funcionalidad Volátil

- Modeling and Composing Navigational *Concern* in Web Applications. Requirements and Design Issues [87]

Crosscuting concern en Aplicaciones Web

- Oblivious integration of volatile functionality in web application interfaces [65]
- Designing the Interface of Rich Internet Applications [101]
- Refactoring to Rich Internet Applications. A Model-Driven Approach [127]
- Transparent Interface Composition in Web Applications [28]
- Modular and Systematic Interface Design for Rich Internet Applications [128]
- Modeling, Deploying, And Controlling Volatile Functionalities In Web Applications [66]

Sistemas de Información Geográfica

- Modelling Location-aware Behaviour in Web-GIS using Aspects[129]
- Improving the Quality of Web-GIS Modularity Using Aspects [130]

Validación de requerimientos de aplicaciones Web

- Detecting Conflicts and Inconsistencies in Web Application Requirements[131]

Concern navegacionales

- Engineering Concern-Sensitive Navigation Structures, Concepts, Tools and Examples [132]

Apéndice C. Términos

ADV (Abstract Data View): un modelo que permite especificar la estructura de las interfaces y su relación con otros componentes de software.

ADV-Charts: Variante de StateChart, es un modelo que permite especificar los aspectos de comportamientos de las interfaces diseñadas con ADV.

Crosscutting Concern: un concern que afecta a otros *concern*. Este tipo de concern usualmente no puede ser claramente descompuesto desde el resto del sistema tanto en su diseño como implementación.

Funcionalidad volátil: es un tipo de funcionalidad que está presente en la aplicación durante un periodo de tiempo corto y que es solicitada de forma inesperada y retirada a partir de eventos específicos del negocio subyacente.

OOHDM (Object Oriented Hypermedia Design Method): OOHDM es una metodología para el diseño y desarrollo de aplicaciones Web que consiste en cinco etapas: análisis de requerimientos, diseño conceptual, diseño navegacional, diseño de interfaces abstracta, e implementación.

Patrón de interface de Usuario: es una solución general y reusable para un problema de diseño de interfaz de usuario recurrente.

Portlets: Los portlets son componentes modulares de las interfaces de usuario gestionadas y visualizadas en un portal web.

Separación de *Concern*: Es la habilidad de identificar, encapsular y manipular aquellos artefactos de software que son relevantes para un concepto específico, tarea o propósito.

Apéndice D. Ejemplo de implementación de funcionalidades volátiles con Cazon

En este apéndice se describe un ejemplo completo y detallado de cómo utilizar Cazon par implementar una funcionalidad volátil. Se utilizará el framework Cazon para la implementación del ejemplo de funcionalidad volátil "Volver a la escuela". Se presentarán las *Afinidades*, *Especificación de Integración*, y *Reglas de producción* necesarias para el ejemplo, y extractos de los modelos OOHDM y archivos XML envueltos en el proceso de modelado e implementación.

Como se comento anteriormente (Sección 1.1) en la tesis, el concern de promoción de "Volver a la escuela" comprende varios nuevos requerimientos (volátiles). Estos requerimientos extienden cada modelo de la aplicación (conceptual, navegacional e interfaz) con nuevos tipos de contenidos, nodos, e interfaces de usuario, respectivamente, que deben ser retiradas o deshabilitadas cuando la promoción termina. Siguiendo la metodología presentada en esta tesis, se mostrará paso por paso como las mejoras de los requerimientos volátiles pueden ser introducidos y removidos de forma transparente y sin invadir la base de la aplicación.

El concern volátil comprende los siguientes requerimientos:

- La definición de un índice de productos escolares que presenta productos con algún tipo de conexión con la escuela, tal como: libros, lápices, mochilas, PC, notebooks, etc. y provee *links* hacia los nodos con los detalles de los productos.
- La incorporación de *links* desde cada producto etiquetado como "escolar"hacia el nodo de índice de productos.
- Disponibilidad de descuentos en el servicio de envío de productos escolares (los descuentos son aplicados en el proceso de compra).
- El periodo de vigencia es bien definido: inicio 15 días antes del comienzo de actividad escolar y 1 mes posterior al inicio del ciclo lectivo.

La Ilustración 76 presenta los modelos core y volátiles del concern "Volver a la escuela". Por una cuestión de simplicidad, se ha limitado los modelos core a aquellas clases conceptuales, navegacionales y de interfaz que pueden ser "afectadas" por alguna característica del concern "Volver a la escuela".

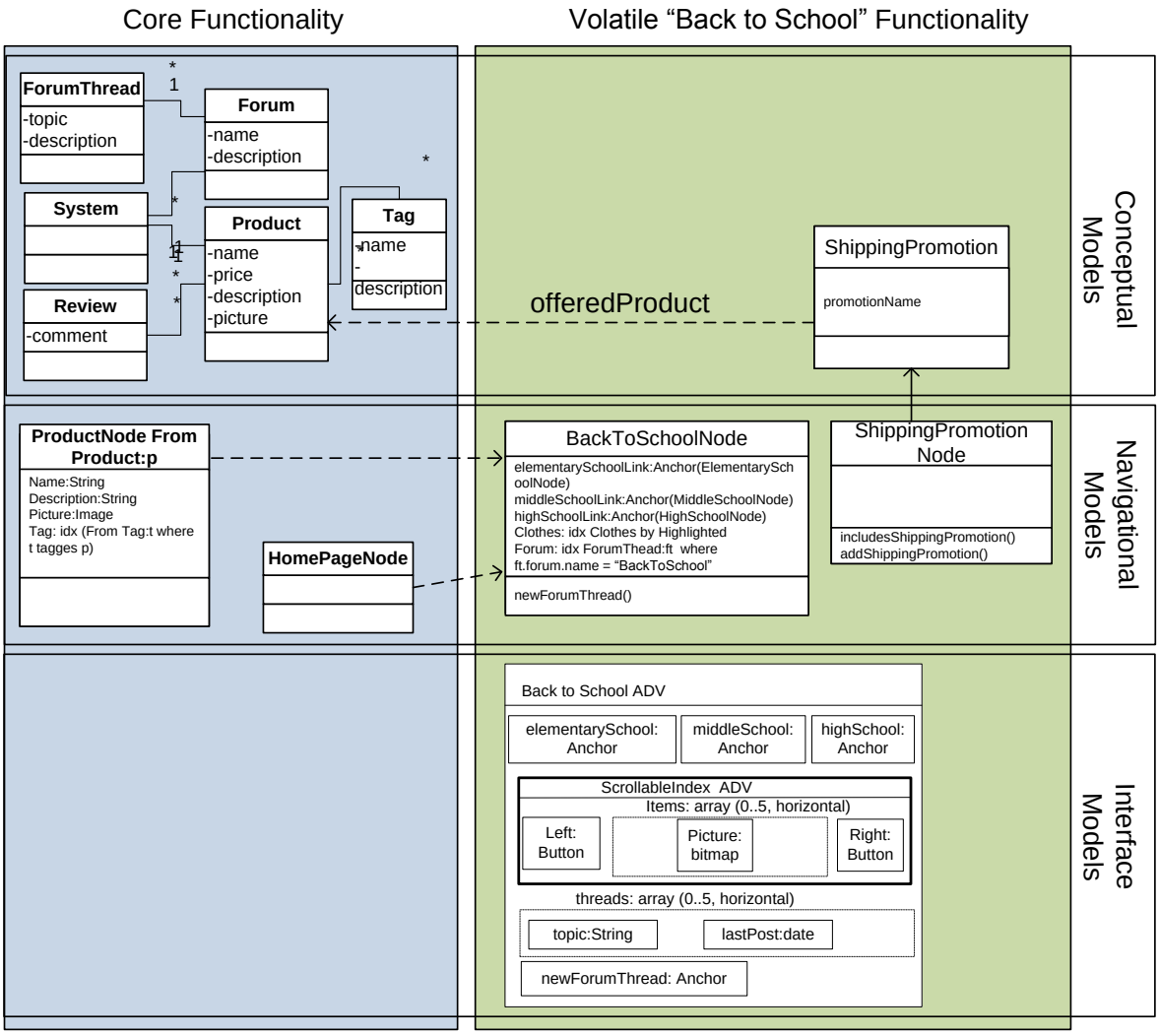


Ilustración 76 Modelos core del sitio de comercio electronico y modelo volátil del concern "Volver a la escuela"

Las afinidades utilizadas en este ejercicio se encuentran diseñadas en la Sección 8.1 mientras que las interfaces de usuario se encuentran diseñadas en la Sección 9.3.

A continuación se describirán los archivos requeridos para mapear el modelo explicado anteriormente en una aplicación ejecutable utilizando Cazon.

El Código 13 presenta un documento XML que define el nodo navegacional *ShippingPromotionNode* definido por la funcionalidad volátil, mostrando como estos atributos se relacionan con los pertenecientes al Modelo Conceptual. La palabra clave *subject* hace referencia al objeto conceptual de tipo “com.eshop.promotions.Shipping” y, utilizando una nomenclatura

similar a la de OGNL, se accede mediante "." a características anidadas del objeto tal como el nombre de la promoción "promotionName". Este documento respeta el esquema de especificación de Nodos de Cazon. Tanto Nodos core como volátiles son descriptos de la misma forma en Cazon.

```
<node-classes>

  <node-class name="ShippingPromotionNode">

    <conceptual-class>com.eshop.promotions.Shipping</conceptual-class>

    <attributes>

        <attribute name="name">subject.promotionName</attribute>

        <attribute name="productName">subject.product.name</attribute>

        <attribute name="picture">subject.product.picture</attribute>

    </attributes>

  </node-class>

</node-classes>
```

Código 13 Especificación del nodo ShippingPromotion en Cazon

El *ADV AddShippingPromotionConfirmation* es mapeado en una transformación XSLT que transforma la representación XML del nodo *ShippingPromotionNode* en un documento HTML. En el Código 14 podemos apreciar la transformación que será referenciada más adelante como *shippingPromotion.xsl*.

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

  <xsl:template name="shippingPromotion">

   <div id="shippingPromotion">

        <b><xsl:value-of select="//name"/> promotion</b>

       ... ... If you include a <xsl:value-of select="//product/productName"/> you get a
free shipping service. Would you like to include it?

     <FORM METHOD="POST" ACTION="AcceptPromotion">

        <INPUT id="OK" TYPE="submit"  value="OK" onclick="acceptPromotion">

        <INPUT id="Cancel" TYPE="submit" value="Cancel" onclick="rejectPromotion">

      </FORM>

   </div>

  </xsl:template>

</xsl:stylesheet>
```

Código 14 Documento XSL que implementa el ADV de la promoción de envío

Como se ha visto hasta el momento, los artefactos que implementan el modelo volátil no están consientes de la aplicación core. La implementación transparente y no intrusiva de los concern volátiles ocurre, primero, definiendo un archivo de configuración de servicio volátil para implementar la *afinidad* navegacional "Back to school affinity.Shipping Promotion" mencionada anteriormente en esta sección. El archivo de configuración de servicio (Código 15) especifica la afinidad y también contiene una representación en XML de los modelos conceptuales. Este archivo se llama */kindle/service-config.xml*.

```
<service-types>

       <service-type name="Back to school"

                    home="ShippingPromotionNode">

              <service-instance name="Kindle promotion"

                                integration-kind="extension">

                      <affinity>

                             FROM ProductNode

                                    WHERE subject.tags.contains('School')

                      </affinity>

                      <conceptualObject>

                             <com.eshop.promotions.Shipping>

                                    <name>Kindle Free Shipping</name>

                                    <product>

                                           <name>Kindle reader</name>

                                           ...

                                           ...

                                    </product>

                             </com.eshop.promotions.Shipping>

                      </conceptualObject>

              </service-instance>

       </service-type>

<service-types>
```

Código 15 Implementación de *Afinidad* en Cazon

En segundo lugar, se describen las Interfaces de Usuario Core y volátil, y los mecanismos de composición. La Interfaz de Usuario core no es originalmente definida como parte de una funcionalidad volátil pero será definida para proveer conocimiento de su estructura que serán utilizados para la definición de las *Especificaciones de Integración.*

En el Código 16 se muestra un fragmento del XML que contiene el botón *AddToShoppingCart*, perteneciente a la interfaz de usuario del concern *shopping* core. Este botón será utilizado más adelante para la definición del *pointcut* de las *Especificaciones de Integración.*

```
        <HTML>

        <HEAD>...</HEAD>

        <BODY>...

        <FORM METHOD="POST" ACTION="AddToShoppingCart">

        ...

                <!—form body -->

                <INPUT id="addProductButton" TYPE="submit"  value="submit"
        onclick="addProduct">

        </FORM>...

        ...

        </BODY>

</HTML>
```

Código 16 Interfaz de usuario del *Producto* donde se muestra el el widget utilizado como pointcut.

El próximo paso corresponde a la implementación del comportamiento de la interfaz de usuario volátil que intercepta el click del botón *AddToProductButton* y muestra el pop-up *addShippingPromotionConfirmation* tal como se diseña en la Sección 9.3. En Código 17, la función *showShippingPromotionConfirmation* juega el rol de *advice* que abre el pop-up confirmación de la promoción de envío cuando se presiona el botón *addProductButton*. Cuando un evento click sobre el botón es capturado, se invoca la función *showShippingPromotionConfirmation* responsable de mostrar el pop-up de que permite al cliente adherirse a la promoción de envió gratuito. Técnicamente el código HTML (ver Código 14) es presentado por la función en forma de un pop-up (dialogo modal) utilizando el componente de dialogo de JQuery[133]. El evento *onclick* es capturado para mostrar el pop-up utilizando un *pointcut* del tipo "around" sobre el widget *addProductButton* utilizando la función del *Aspect.addArround* motor AOP utilizado[105]. Una vez capturado el evento, este es "conjelado" hasta que el usuario decida si acepta o no la promoción ya que el evento ejecuta la lógica original de agregar el producto al carrito sin tener en cuenta la promoción. En el caso de que se acepte la promoción, el evento *onclick* original sobre el botón *addProduct* es descartado y se invocar la función *acceptPromotion* que eventualmente delega dispara la invocación del mensaje *addShippingPromotion* (introducido por la afinidad) del nodo *ProductNode*; caso contrario, se ejecuta *rejectPromotion* donde este descongela el evento *onclick* sobre el botón *addProductButton.* Este código será referenciado como el archivo "shippingPromotionInterceptor.xml".

```
<XML>

<!-- API inclusion -->

<script type="text/javascript"   language="javascript" src="aspect2.js" ></script>

<script>

       <!- - this code wraps onclick function - ->

       Aspects.addAround(showShippingPromotionConfirmation,document.getElementById('addPro
ductButton'), "onclick");

       function showShippingPromotionConfirmation(){

       $("# shippingPromotion").dialog({modal: true });

       }

       function acceptPromotion(){

         //Makes an invocation to the shipping promotion node's method.

       ...

       }

       function rejectPromotion (){

         //triggers the addProductButton.onclick behavior.

         addProduct()

       }

   </SCRIPT>
       <!- -shipping promotion stylesheet - ->

       <include shippingPromotion.xsl>

</XML>
```

Código 17 Comportamiento volátil del concern "Volver a la escuela"

La composición de las interfaces volátiles y core es implementada utilizando la transformación XSL del Código 18 que será referenciado más adelante como el archivo "shippingPromotionIntegration.xsl".

```
    <xsl:stylesheet … >
        <!-- Imports a transformation which copies all the elements -->
        <xsl:import href="defaultTemplate.xsl"/>
        <xsl:template match="//BODY”>
              <xsl:copy-of
           select="document('shippingPromotionInterceptor.xml')//XML/*"/>
              <xsl:copy-of select="."/>
        </xsl:template>
    </xsl:stylesheet>
```

Código 18 Implementación de *Especificación de Integración* en Cazon

Finalmente, estos archives son empaquetados dentro de un archive JBoss SAR con la configuración del servicio MBean del Código 19.

```
<?xml version="1.0" encoding="UTF-8"?>

<server>

  <mbean code="...BasicService"name="user:service=KindleShippingPromo">

    <attribute name="ServiceConfigurationFile">/kindle/service-config.xml</attribute>

    <attribute name="StrutsConfigFile">/kindle/kindle-config.xml</attribute>

    <attribute name="ServiceName">KindleShippingPromotion</attribute>

    <attribute name="ServiceXslt">/shippingPromotionIntegration.xsl</attribute>

    <attribute name="NodeTypesConfigFile">/kindle/shipping-nodeTypes.xml</attribute>

    <depends>user:name=Cazon,type=Barrier</depends>

  </mbean>

</server>
```

Código 19 Configuración MBean del servicio volátil

La clase *BasicService* provee el comportamiento común para todos los servicios volátiles. En particular, ésta registra el servicio en el componente *VServiceManager*.

Cuando el servicio *KindleShippingPromo* es iniciado y un nodo del tipo *ProductNode* es solicitado, el componente VServiceManager intercepta el pedido en flujo de trabajo de Cazon para incorporar el nodo *ShippingPromotionNode* en el nodo *ProductNode*, de acuerdo a lo que especifica la *Afinidad* de tipo *Extension* y su correspondiente implementación en el archivo de configuración de servicio. Inmediatamente, la capa de presentación de Cazon encargada de procesar los Modelos de Interfaz de Usuario, la *Especificación de Integración* expresada en XSL va a componer la interfaz del *Producto* en tiempo de ejecución con la interfaz de promoción de envío a domicilio. El resultado va a ser una segunda transformación que será aplicada al nodo aumentado (*ProductNode* + *ShippingPromotionNode*) para obtener el código HTML final que es enviado al cliente en la respuesta HTTP.

El último paso es la especificación de las *Reglas de Producción* que nos permitirán activar y desactivar la funcionalidad. Inicialmente, tal como el código Código 20 lo muestra, la promoción será activada cada año, un mes y medio antes de que el periodo escolar comience. Dado que las promociones son usualmente especificadas por el personal de ventas, las fechas de inicio y fin podrán ser modificadas también en tiempo de ejecución para adecuarse a las necesidades del negocio. Cada una de estas reglas de negocio es implementada en un archivo Drools con extensión .dslr, utilizando un DSL apropiado.

```
WHEN
   Time is *-Ago-15 00:00
THEN
CONNECT
   Concern BackToSchool
NAV_Affinity Back to school affinity  - Generic link,
   Back to school affinity  - Home Page
   Back to school affinity  - Kindle Promotion
UI_Integration Back to school integration  - Generic link,
   Back to school integration  - Home Page
   Back to school integration  - Kindle Promotion
END
```

Código 20 Regla de Producción de activación de la funcionalidad volátil

El DSL mapea la palabra clave "CONNECT" con el código especifico Java, requiriendo al componente VServiceManager que inicie el servicio volátil con los parámetros definidos.

Luego que el periodo de escolar ha comenzado, se querrá que un mes después algunas funcionalidades volátiles sean desactivadas tal como lo muestra el Código 21.

```
WHEN
    Time is *-Oct-2 00:00
THEN
    DISCONNECT
    CONCERN BACKTOSCHOOL
    NAV_AFFINITY BACK TO SCHOOL AFFINITY  - KINDLE PROMOTION
    UI_INTEGRATION BACK TO SCHOOL INTEGRATION - KINDLE PROMOTION
END
```

Código 21 Regla de Producción de desactivación de la funcionalidad volátil en base a eventos temprales

Hasta ahora sólo se ha definido reglas en base a eventos de tiempo, sin embargo puede ocurrir que el lector Kindle se quede sin stock por el éxito de la promoción. Para poder desactivar la promoción cuando esto sucede, en Código 22 se especifica una nueva regla que está pendiente de eventos de negocio que indiquen que el lector Kindle se ha quedado sin stock. Esto va a causar que el sistema desactive la promoción cuando el evento lanzado por la realización de la venta de la última unidad en stock.

```
WHEN
    Kindle is out of stock
THEN
    DISCONNECT
    Concern BackToSchool
    NAV_Affinity Back to school affinity  - Kindle Promotion
    UI_Integration Back to school integration  - Kindle Promotion
END
```

Código 22 Regla de Producción de desactivación de la funcionalidad volátil en base a eventos de negocio