FACULTAD DE INFORMATICA
UNLP

TESINA DE LICENCIATURA

TITULO: Definición de Rich Internet Applications a través de Modelos de Dominio Específico

AUTORES: Buzzo, Marcos Hernán; Rivero, José Matías

DIRECTOR: Dr. Gustavo Rossi

CODIRECTOR: Dra. Claudia Pons

CARRERA: Licenciatura en Informática

Resumen

Las Rich Internet Applications o RIAs son un tipo de aplicaciones web las cuales exhiben características de interfaz de usuario y comportamiento usualmente presentes en aplicaciones de escritorio convencionales. Las mejoras que proveen en comparación con las aplicaciones web tradicionales las hace un área de interés en la actualidad, razón por la cual han surgido un conjunto heterogéneo de tecnologías que facilitan su implementación. Sin embargo, al seguir estando basadas fuertemente en la escritura de código fuente, éstas no se consideran suficientes para alcanzar mejoras sustanciales de productividad en el desarrollo de aplicaciones de gran magnitud o complejidad. Para resolver esta problemática, en el contexto de las aplicaciones web tradicionales han surgido un conjunto de metodologías basadas en modelos (MDD), las cuales proponen obtener el software final directamente desde modelos de alto nivel. Actualmente, se han propuesto extensiones a las mismas a través de las cuales pueden especificarse características RIA. En sintonía con esto, en esta Tesina se lleva a cabo el estudio detallado de distintos aspectos de las Rich Internet Applications con el fin de confeccionar un lenguaje de modelado (meta-modelo) que permita especificarlas y obtener las implementaciones finales completas a través de un proceso de transformación.

Líneas de Investigación

- Rich Internet Applications
- Modelado Específico de Dominio (DSM)
- Ingeniería web dirigida por modelos (MDWE)
- Tecnologías y patrones de diseño RIA
- Entornos y lenguajes de meta-modelado

Conclusiones

En este trabajo se mostró la viabilidad, ventajas y limitaciones de la metodología de Modelado Específico de Dominio aplicada en el área de las Rich Internet Applications. A través del lenguaje confeccionado y del entorno de modelado y derivación implementado, se probó que la generación de RIAs desde modelos de alto nivel es factible y, además, puede conllevar a la obtención de mejoras significativas de productividad en su desarrollo.

Trabajos Realizados

En este trabajo:
- Se analizaron las principales características de las Rich Internet Applications, distintas tecnologías que facilitan su desarrollo y patrones de diseño orientados a su especificación.
- Se comentó la metodología de Modelado Específico de Dominio, su aplicabilidad, ventajas y limitaciones.
- Fueron abordadas distintas metodologías de modelado web y RIA existentes.
- A partir de la investigación realizada, se definió un lenguaje formal que permite especificar RIAs en un alto nivel de abstracción y un conjunto de transformaciones que traducen instancias del mismo en implementaciones funcionales.

Trabajos Futuros

Los trabajos futuros que se proponen para esta Tesina son:
- El enriquecimiento del lenguaje confeccionado a fin de que permita especificar características RIA que no han sido contempladas.
- La implementación de una sintaxis concreta para el lenguaje propuesto.
- Mejorar el proceso de derivación de modelos a implementaciones funcionales, introduciendo una o más transformaciones modelo-a-modelo previas a la transformación modelo-a-texto final.

Fecha de la presentación: Septiembre de 2009

ÍNDICE

1 INTRODUCCIÓN

1.1 Motivación

Las Rich Internet Applications (Aplicaciones de Internet Ricas o RIAs), aplicaciones web que poseen características y funcionalidades inherentes a aplicaciones de escritorio, representan un gran área de interés en el desarrollo de software en la actualidad gracias al incremento de interactividad y experiencia de usuario que brindan en comparación con las aplicaciones web tradicionales. Por este motivo, han surgido una variedad heterogénea de tecnologías y frameworks orientados a su desarrollo los cuales, si bien permiten alcanzar mayor productividad en su construcción, no son capaces de proveer instrumentos necesarios para lograr la abstracción necesaria a la hora de especificar aplicaciones de gran complejidad y tamaño[33]. Esta carencia es la principal causa de que el incremento de productividad alcanzado en su desarrollo no pueda ser sustancial.

La problemática planteada es, en realidad, una instancia de un problema más general: los lenguajes de Tercera Generación, los cuales condujeron con su surgimiento a un aumento relevante en términos de abstracción (y su consecuente ganancia en productividad) en la construcción de software, a pesar de su evidente evolución, no han hecho grandes aportes en este campo en los últimos tiempos. Dado que los avances en lo que a productividad concierne en la historia del desarrollo de software han estado impulsados mayormente por incrementos en la abstracción de las representaciones que lo especifican, comienza a apreciarse la necesidad de proveer métodos y herramientas para hacer este incremento factible. La metodología de Modelado Específico de Dominio (Domain Specific Modeling o DSM) propone alcanzarlo acercando las especificaciones de software al dominio del problema que éste intenta solventar.[1][28]

1.2 Objetivos

Planteado este contexto, en esta Tesina se propone, luego de discutir las características teóricas, prácticas y tecnológicas de las Rich Internet Applications, aplicar la metodología de DSM en su desarrollo. Esta tarea comprende la correcta captura de un dominio particular en la construcción de RIAs, la definición formal de un lenguaje que permita describirlas en sus aspectos característicos y la derivación desde instancias de este último a implementaciones finales con calidad de producción.

El objetivo que se persigue consiste, por un lado, en capturar aspectos relativos a las Rich Internet Applications y plasmarlos en un lenguaje formal que permita caracterizarlas. Se espera que éste último ayude a comprender de manera más precisa las características fundamentales de las RIAs, en conjunción con los aportes de lenguajes ya existentes orientados a su modelado que se comentan en carácter de *trabajos relacionados*. Algunos de estos últimos inspiraron, ya sea directa o indirectamente, parte del lenguaje aquí propuesto.

Asimismo, a través de la presente Tesina también se busca mostrar la viabilidad de la metodología de Modelado Específico de Dominio focalizada en un área particular y contemporánea en el desarrollo de software como lo es la implementación de Rich Internet Applications. Esto incluye la descripción de todo el proceso involucrado en su aplicación: la

investigación acerca del dominio particular, el análisis de factores de importancia a considerar, la descripción informal y luego formal del lenguaje de dominio específico y la construcción de derivadores de código y del entorno de ejecución del software generado.

1.3 Estructura de la Tesina

En primer lugar, en este documento se describirá el concepto de Rich Internet Application, sus características fundacionales, aquellos aspectos que motivan el actual interés en su desarrollo y una breve mención a la evolución histórica del software que conlleva a su definición. Para cerrar esta temática particular, se comentan los resultados de un breve análisis efectuado sobre algunas tecnologías actuales orientadas a la definición de RIAs.

Acto seguido, se tratará la metodología de Modelado Específico de Dominio. Se introducirán sus motivaciones, ideas y conceptos principales, para luego comentar sus ventajas, restricciones de aplicabilidad, los preceptos a respetar que promueve y sus diferencias con otras metodologías de desarrollo basadas en modelos.

En las secciones posteriores, se relacionan estos dos tópicos que hasta el momento aparentan ser disjuntos, buscando aplicar la metodología DSM en la definición de Rich Internet Applications. En primer lugar, se mencionan aquellos aspectos de las RIAs que motivan su especificación a partir de modelos de alto nivel de abstracción. Luego, se lleva a cabo un análisis de patrones de diseño orientados a Rich Internet Applications y de metodologías de modelado web ya existentes. Estos, en conjunción con las características de las tecnologías ya tratadas, brindarán una base teórica general para comenzar con la definición informal de un lenguaje específico de dominio de alto nivel de abstracción abocado a la definición de RIAs.

Habiendo sido definido informalmente este lenguaje, se procede a elaborar su definición formal. Para ello, se comentan brevemente algunos entornos y lenguajes de modelado que fueron analizados con este objetivo. Finalmente, se escoge uno particular, y se confecciona con el mismo la mencionada definición.

Para terminar, se comentan en forma sintética aspectos concernientes a la implementación del lenguaje de alto nivel definido, de los derivadores de código que permitirán obtener la aplicación final y del entorno de ejecución del software generado.

2 RICH INTERNET APPLICATIONS

2.1 Introducción

En esta sección se introducirá el concepto de Rich Internet Application y sus principales características. Luego, se mencionará los motivos por los cuales su implementación es de gran interés en la actualidad. Finalmente, se describirá la evolución histórica de las aplicaciones distribuidas hasta llegar a las RIAs.

2.2 Definición

Las Rich Internet Applications (Aplicaciones de Internet Ricas o RIAs) son aplicaciones web que poseen características y funcionalidades presentes en aplicaciones tradicionales de escritorio como, por ejemplo, capacidad de interacción frente a diversos eventos de UI, procesamiento de datos y ejecución de aspectos de la lógica de negocios en el cliente, entre otros. Las RIAs representan un intermedio entre los dos extremos conformados por las aplicaciones web tradicionales o *thin-client*, totalmente dependientes del server-side, y las aplicaciones de escritorio o *fat-client*, cuya funcionalidad se centra en su totalidad en el client-side. Esta naturaleza híbrida permite reunir en las Rich Internet Applications los aspectos más favorables de las aplicaciones web tradicionales y de las aplicaciones desktop[21].

A diferencia de las aplicaciones web tradicionales, las Aplicaciones Ricas se ejecutan en el cliente, sobre sobre un *runtime*, plataforma o máquina virtual particular como, por ejemplo, un browser que implemente DHTML/Ajax[1]. Además de ser el mecanismo básico de ejecución de las aplicaciones, el susodicho *runtime* o plataforma también trabaja a modo de *sandbox*, limitando el acceso que las mismas tienen al entorno en el cual se ejecutan a fin de no comprometer la seguridad en el client-side.

Al igual que las aplicaciones de escritorio convencionales, las RIAs pueden almacenar información y alocar memoria en el entorno sobre el cual se ejecutan, lo que permite que el estado de ejecución de las aplicaciones se traslade parcial o totalmente a la faceta cliente. Esto se contrapone con las aplicaciones web tradicionales, cuyo estado de ejecución es contenido en el server-side prácticamente en su totalidad.

Desde un punto de vista arquitectural, las Rich Internet Applications conforman una variante particular de la ya conocida arquitectura cliente/servidor. Las características que hacen que las RIAs sean una variante de esta arquitectura y no directamente una instancia particular de la misma es la portabilidad y seguridad provista por el *runtime* en el que se ejecutan, la utilización de Internet para su descarga y como medio de comunicación con el server-side y la facilidad que proveen en términos de su distribución, instalación y actualización.Más allá de la complejidad de las tareas que las Aplicaciones Ricas permitan llevar a cabo en el client-side, el hecho de que su arquitectura subyacente se corresponda al

1 En este documento, se utilizará el término *DHTML/Ajax* para hacer referencia al *runtime* de RIAs que implementan de forma nativa la mayoría de los *browsers* modernos. El mismo consiste en la definición de la UI a través de un documento HTML, el cual puede ser modificado dinámicamente a través del lenguaje JavaScript, permitiendo además llevar a cabo peticiones asincrónicas al servidor a través de la tecnología denominada *Ajax*.

esquema cliente-servidor implica que deberán indefectiblemente interactuar con el server-side, más allá de que no dependan del mismo para efectuar gran parte de sus actividades. Esto último establece que las RIAs deberán implementar y considerar como un aspecto importante en su construcción, mecanismos de interacción y transferencia de datos con la faceta servidor, por ejemplo, facilitando la invocación sincrónica o asincrónica de servicios provistos en el mismo.

Siguiendo los preceptos de las aplicaciones web tradicionales, las RIAs deben implementar una manera sencilla y robusta de instalación y distribución de las aplicaciones bajo distintas plataformas de ejecución como, por ejemplo, computadoras de escritorio, PDAs, teléfonos celulares, etc. Este aspecto ha sido uno de los principales impulsores del notable crecimiento de las aplicaciones web y por lo tanto es también de vital necesidad en las Rich Internet Applications.

A fin de proveer capacidades de interactividad avanzadas de las cuales las aplicaciones web tradicionales carecen, las RIAs encuentran la necesidad de brindar un modelo de eventos y aplicaciones altamente extensible, el cual sea capaz de implementar y reunir aspectos de interfaz de usuario, comunicaciones y servicios a nivel de sistema. El mismo deberá permitir, entre otras cosas, modificar en tiempo real la interfaz de usuario, lo cual abre paso a implementar interactividad sin necesidad de navegación y evitando comunicaciones innecesarias con el servidor. Esto implica que en las Aplicaciones Ricas, la interacción y la navegación son considerados como aspectos separados, siendo el primero dependiente del segundo en las aplicaciones web tradicionales. Adicionalmente, es deseable que se habiliten en las mismas mecanismos que faciliten y agilicen el desarrollo a través de componentes reusables, los cuales puedan combinarse y relacionarse para componer nuevas aplicaciones, como se observará más adelante al analizar distintas tecnologías orientadas al desarrollo de RIAs.

Un aspecto novedoso de las RIAs es la capacidad para que las aplicaciones puedan ejecutarse sin conexión con el servidor. Esta capacidad implica, además de la necesidad de almacenamiento en el client-side, el establecer una política de sincronización a llevar a cabo cuando la conexión con el server-side se reestablezca. Finalmente, otro aspecto de interés que actualmente es objeto de investigación en las Aplicaciones Ricas, es la posibilidad de iniciar transferencias de datos (al menos, virtualmente) desde el server-side.

2.3 Motivación e interés en su desarrollo

La motivación principal que impulsa el desarrollo de las RIAs es la reducida capacidad de interacción que las aplicaciones web tradicionales proveen. Mientras que estas últimas han tenido gran éxito gracias a su facilidad de distribución y *deployment*, su modelo de programación e interacción inherente no es el deseable para desarrolladores ni usuarios. [20][21]. A continuación, se mencionan las limitaciones inherentes a las aplicaciones web tradicionales, describiendo luego cómo se busca solucionarlas a través de las RIAs. Finalmente, se destacan ventajas que las RIAs promueven, más allá de representar una solución a la problemática inherente de las aplicaciones web tradicionales.

2.3.1 Limitaciones en las aplicaciones web tradicionales

La primer limitación evidente que enfrentan las aplicaciones web tradicionales es la

falta de interactividad inherente a utilizar documentos HTML como única interfaz de usuario[21]. Estos documentos son descargadas en una sola unidad y la única interactividad que proveen es el efectuar una acción de navegación, la cual tiene como consecuencia la descarga completa de otro documento y, por ende, una modificación total de la UI[2]. El envío de datos desde el cliente al server-side se efectúa a través de *formularios* los cuales son conformados por componentes orientados al ingreso de información, incluidos dentro de los documentos HTML. Las acciones de navegación a partir de documentos que incluyan estos componentes pueden acarrear datos asociados los cuales son comunicados a la faceta servidor y pueden ser utilizados para efectuar tareas relativas a la lógica de negocios de la aplicación. En consecuencia, en las aplicaciones web tradicionales, el client-side no posee prácticamente información alguna acerca del estado de ejecución, siendo este almacenado en el server-side.

En el contexto planteado en el párrafo anterior, siendo la navegación la única interacción posible, la representación de los flujos de trabajo de relativa complejidad puede llevarse a cabo de dos maneras: o bien incluyéndolo por completo en una misma UI (documento), o bien dividiéndolo en distintos documentos interconectados entre sí a través de acciones de navegación. Mientras que la primer opción resulta en una UI de gran complejidad, difícil de entender para el usuario y de desarrollo poco escalable, la segunda implica el efectuar dos o más navegaciones a fin de completar un único proceso de trabajo. En este último caso, cada una de estas navegaciones implica un retardo de comunicación y transferencia de datos con el server-side, el cual deberá transferir por completo una nueva UI por más que las modificaciones necesarias en la existente en la faceta cliente a fin de expresar el cambio de estado sean mínimos. La cantidad de veces que este retardo debe enfrentarse es directamente proporcional a la complejidad del proceso.

Las complicaciones que se observan en la ejecución de flujos de trabajo, se evidencian de igual manera al intentar proveer exploración de datos complejos en una aplicación web tradicional. La misma debe ser dividida en páginas, lo cual conlleva a las problemáticas ya mencionadas. Al explorar grandes conjuntos de datos, el problema es similar. Por ende, es necesario proveer mejoras en la interactividad tales que, en primer lugar, el navegar a través datos complejos o de gran magnitud requiera menos transferencia de información. Un ejemplo de este tipo de mejoras es el evitar transferir por completo la UI y comunicar sólo aquella información que se requiera al efectuar una acción de exploración. En segundo lugar, la demora que conlleva la latencia asociada a la comunicación inicial con el servidor puede enmascararse, pudiendo efectuarse ésta última asincrónicamente desde la misma UI, inclusive antes de que el usuario la solicite.

Finalmente, otra gran limitación de las aplicaciones web tradicionales es la incapacidad para implementar comportamiento complejo de interfaz de usuario sin intervención del servidor. Ejemplos de este tipo de situaciones se dan cuando es deseable generar bajo demanda del usuario componentes a través de los cuales se ingresarán datos o cuando el estado de un componente determina de manera interactiva el comportamiento de otros. Si bien este tipo de casos sólo conciernen a la UI presente en el cliente, en una aplicación web tradicional es necesario interactuar continuamente con el server-side para

2 En realidad, a través del lenguaje de scripting JavaScript y del modelo DOM, implementados en la mayoría de los navegadores, los documentos HTML pueden ser modificados desde el client-side. Sin embargo, en la bibliografía y en este documento, este tipo de interacción es considerada como *rica* y, por ende, no es incluida dentro de las capacidades atribuidas a las aplicaciones web tradicionales.

hacer posible su implementación.

2.3.2 Las RIAs como solución a las problemáticas presentes por las aplicaciones web tradicionales

La riqueza de interacción que las RIAs promueven, según lo mencionado en la caracterización de las mismas, implica la capacidad de modificar la UI en tiempo real sin intervención del server-side, la reacción frente a eventos de usuario diversos y la capacidad de comunicación asincrónica con el servidor. Estas funcionalidades proveen la base para plantear soluciones a las limitaciones relativas a las aplicaciones web tradicionales mencionadas en la sub-sección anterior.

En primer lugar, la capacidad de modificar la UI en el client-side permite reemplazar aquellas acciones de navegación que son necesarias a fin de dividir flujos de trabajo, los cuales pueden ser representados ahora en una misma UI la cual mute y cambie de estado a medida que se ejecuta el proceso. Al requerirse la exploración de datos complejos o de gran volumen, la solución es similar: en lugar de cargarse una nueva UI con los nuevos datos, los mismos son solicitados al servidor a través de una petición asincrónica y, una vez recibidos, la interfaz de usuario es modificada a fin de adecuarla a la nueva información a mostrar. Asimismo, aquellas aplicaciones web tradicionales que requieran de una invocación al server-side para llevar a cabo comportamiento de interfaz de usuario avanzado como, por ejemplo, creación dinámica de componentes, pueden ser re-implementadas por una RIA la cual realice estas acciones en el client-side, sin intervención de la faceta servidor.

Finalmente, la capacidad de captar eventos de usuario variados en conjunción con la posibilidad de hacer mutar la UI, permiten implementar de manera natural interfaces de usuario que respondan en tiempo real a eventos de UI de distinta índole.

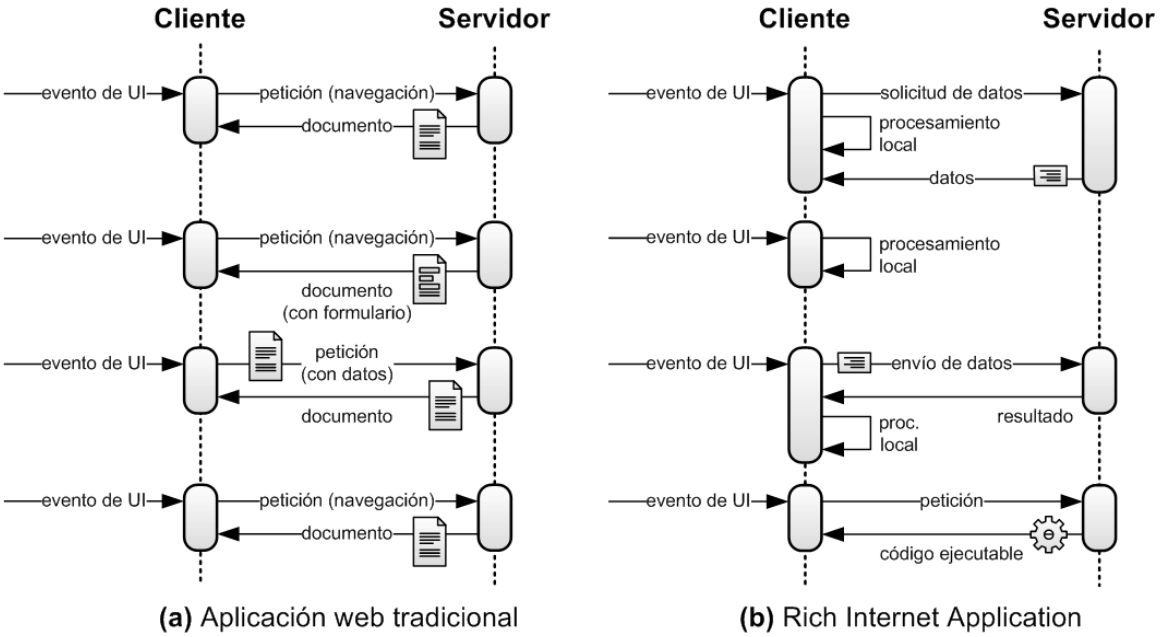


Figura 1. Ejemplo de interacción diferenciando el comportamiento una aplicación web tradicional (a) y una RIA (b).

En la Figura 1 se esquematiza una secuencia de interacciones entre cliente y servidor, remarcando las diferencias en las actividades llevadas a cabo en una aplicación web tradicional (a) y una Rich Internet Application (b). En primer lugar, se hace notar la capacidad de procesamiento local de las RIAs, la cual no se hace presente en las aplicaciones web tradicionales. Esta capacidad permite evitar peticiones y transferencias de datos innecesarias con el servidor. En segundo lugar, se puede observar que en las RIAs no se requiere la transferencia de un nuevo documento que conforme la UI por cada petición al server-side, como sí ocurre en las aplicaciones web tradicionales.

Como también se denota en el esquema, algunas tecnologías RIA permiten la transferencia de código ejecutable desde el servidor a fin de permitir la carga progresiva y bajo demanda de partes de la aplicación y así mejorar su tiempo de respuesta y descarga inicial. Finalmente, se hace notar la capacidad de las RIAs de efectuar peticiones asincrónicas de datos al server-side, las cuales permiten que la faceta cliente reciba eventos de UI y lleve a cabo procesamiento local mientras las mismas se llevan a cabo.

2.3.3 Ventajas adicionales que las RIAs promueven

En las Aplicaciones Ricas, la mejorada equidad de trabajo entre cliente y servidor favorece a ambas partes. La faceta cliente se ve favorecida gracias a la mayor respuesta en cuanto a interacción, dado que las tareas que no requieran interactuar o trasferir datos con el servidor son llevadas a cabo localmente. Adicionalmente, estos trabajos que son ahora llevados a cabo por la faceta client-side de la aplicación implican una menor cantidad de carga e interacción con el servidor, lo cual se traduce en menos necesidad de procesamiento y de comunicación en el mismo. En términos económicos, el resultado es un menor gasto en hardware y software en servidores y clientes o usuarios más satisfechos en cuanto a la eficiencia e interactividad de las aplicaciones. Algunas tecnologías de RIAs ofrecen la posibilidad de carga dinámica de partes de las aplicaciones bajo demanda, de manera de ahorrar ancho de banda y mejorar la descarga inicial, lo cual significa una mejora adicional con respecto a lo dicho anteriormente.

El hecho de que las Aplicaciones Ricas se ejecuten bajo un *runtime* particular en el host cliente dotan a las mismas de una gran portabilidad, dado que usualmente se provee una versión del mismo para cada una de las plataformas[3] más comunes en el mercado. Los únicos requerimientos fundamentales que una RIA requiere para ejecutarse son una conexión a Internet, la cual en algunos casos puede necesitarse sólo por períodos, y la existencia del *runtime* particular sobre la plataforma de ejecución elegida.

La posibilidad de omnipresencia a nivel de plataformas de ejecución, en conjunción con las ventajas de fácil instalación y de interacción, preservando aquellos aspectos que promueven al gran éxito de las aplicaciones web tradicionales, han hecho que las Aplicaciones Ricas sean un área de actual interés en el desarrollo de software. Debido a esto, han surgido una variedad de tecnologías centradas en su confección, las cuales comparten varios aspectos en común. Algunos de estos aspectos serán analizados a posteriori y serán la base para temas tratados más avanzado este documento.

La seguridad es otra característica favorable en las RIAs en comparación a la

3 En este contexto, la palabra *plataforma* no se utiliza como sinónimo del *runtime* sobre el cual corre una RIA, sino a la combinación de un Sistema Operativo y hardware subyacente particulares.

arquitectura cliente/servidor clásica. Los *runtimes* sobre los cuales las Aplicaciones Ricas corren ofrecen protección frente a accesos a su ambiente externo, usualmente limitando el mismo al entorno de datos de la propia aplicación y a los servidores remotos desde donde esta última fue descargada. Esta característica es diferente en las aplicaciones de escritorio o cliente/servidor usuales, cuyo client-side se ejecuta comúnmente con privilegios similares al de cualquier otra aplicación y, por lo tanto, tiene acceso a la mayor parte de su entorno de ejecución, sin mayores restricciones más que las impuestas por el Sistema Operativo. Adicionalmente, las aplicaciones descargadas pueden ser digitalmente firmadas y/o accedidas a través conexiones seguras, a fin de que los usuarios finales sepan que las mismas fueron diseñadas por organizaciones o empresas de confianza.

En términos arquitectónicos, las RIAs promueven una separación verdadera entre la lógica de presentación e interfaz de usuario de la lógica de la aplicación en el server-side. La lógica de negocios, implementada en su totalidad en el server-side en las aplicaciones web convencionales, es ahora distribuida entre cliente y servidor, existiendo inclusive la posibilidad de implementarse en la faceta cliente en su totalidad. Como consecuencia, las capas de la aplicación que deben desarrollarse en el servidor se ven reducidas en cantidad y/o calidad. Por ejemplo, la riqueza en términos de comportamiento en el client-side permiten implementar la capa de servicios en el servidor de una manera más simple, siguiendo una arquitectura RESTful[13], como se verá más adelante.

2.4 Breve historia

Desde el punto de vista histórico, las RIAs representan un retorno hacia la arquitectura de software cliente-servidor. En los albores del software y de los sistemas de información, la arquitectura por defecto de los sistemas fue dominada por los *mainframes* con terminales *livianas*, migrándose luego al paradigma cliente/servidor, en donde los clientes cobraron mayor importancia y dejaron de ser *livianos*. Más tarde, el surgimiento de la WWW introduce la posibilidad de brindar aplicaciones on-line que no requieren de instalación en el client-side, sino que se ejecutan directamente en el servidor, transfiriéndose al cliente documentos de texto enriquecidos a través de un lenguaje de marcas (HTML) que conforman la única interfaz de usuario. En este contexto, la interacción desde el cliente hacia el servidor está dada a través de la navegación, la cual puede acarrear datos enviados desde el client-side como ya fue mencionado. Siendo en las aplicaciones web tradicionales el ingreso de datos y la navegación la únicas tareas factibles de ser ejecutadas en el client-side, el resto de la carga de procesamiento queda delegada completamente al servidor. Adicionalmente, cualquier interacción necesaria como, por ejemplo, una aviso de error en el formato de los datos ingresados, debe ser incluida en un nuevo documento, el cual tiene que ser enviado por completo desde el servidor al cliente como resultado de una acción de navegación.

Más allá de las limitaciones ya mencionadas, el gran éxito de la WWW impulsa nuevamente el retorno a los clientes *livianos* a través de las aplicaciones web tradicionales. La gran aceptación de este tipo de aplicaciones proviene de no necesitarse instalación de software alguna en los hosts clientes para hacer que las mismas estén disponible para los usuarios, siendo un efecto colateral de esta capacidad la posibilidad de actualizar la aplicación sin necesidad de llevar a cabo tarea alguna en el client-side. En este contexto, el surgimiento de las RIAs significa una nueva migración hacia la arquitectura cliente/servidor,

esta vez en una variante *global*, que utiliza Internet como medio de bajo costo para distribución y desplegado de las aplicaciones, así como también de comunicación entre las facetas client- y server-side. [2][6]

El término Rich Internet Application es introducido en 2002 por Macromedia a fin al dar a conocer su producto Macromedia Flash MX[20], buscando expresar las restricciones que las aplicaciones web poseen y proponiendo cómo solventarlas. Sin embargo, las RIAs tienen su origen en el año 1995 bajo la plataforma Java a través de la implementación de los *applets*, aplicaciones que se descargaban directamente desde Internet y eran ejecutadas en una máquina virtual en el host cliente. El notorio tiempo y costo de carga asociados a la máquina virtual necesaria para correr dichas aplicaciones fue la principal falencia de este esquema, que no tuvo mayor éxito, siendo la tecnología Java hoy en día extremadamente popular en el server-side. Tiempo después, los browsers más conocidos en el mercado comienzan a concebir métodos para extender su funcionalidad y brindarles a los usuarios aplicaciones de mayor riqueza a través de la inclusión de plug-ins o extensiones instaladas en los mismos navegadores. La necesidad de instalación de dicho software para obtener acceso a los beneficios de las nuevas y ricas aplicaciones es en esta ocasión el impulsor del fracaso de este nuevo intento por proveer una experiencia de usuario más rica, permaneciendo Macromedia Flash y Java como plug-ins de mayor difusión para ejecución de RIAs[6].

En la actualidad, la necesidad de proveer mayor riqueza en las aplicaciones web impulsa no sólo el desarrollo de aplicaciones bajo las 2 tecnologías mencionadas en el párrafo anterior, sino el surgimiento de nuevos productos de software orientados a su ejecución y desarrollo como, por ejemplo, Microsoft Silverlight[4], JavaFX[5], y Adobe Flex[6]. Estas tecnologías apuntan a proveer desarrollo ágil de aplicaciones a través del uso de lenguajes específicos y a lograr eficiencia en su ejecución, facilitando además la portabilidad de las mismas a una variedad heterogénea de plataformas como, por ejemplo, dispositivos móviles. Asimismo, la definición de estándares y el cumplimiento de los mismos por parte de los browsers modernos hace cada vez más posible el implementar Aplicaciones Ricas portables capaces de correr directamente sobre estos. Las RIAs implementadas para ejecutarse de forma nativa en los navegadores corren con la ventaja adicional de no requerir instalación ni actualización de ningún plug-in o *runtime* en el host cliente. Por este motivo, los browsers representan una de las plataformas más atractivas en la actualidad para implementar y ejecutar Rich Internet Applications, lo cual puede apreciarse a través del sinfín de librerías y tecnologías que han surgido para facilitar la construcción de Aplicaciones Ricas capaces de correr de manera nativa en los mismos.

4 http://www.microsoft.com/silverlight/
5 http://javafx.com/
6 http://www.adobe.com/products/flex/

3 ANÁLISIS DE TECNOLOGÍAS ACTUALES PARA LA CONSTRUCCIÓN DE RIAS

3.1 Introducción

En esta sección se analizarán distintas tecnologías modernas concebidas para facilitar la tarea de construir RIAs en la actualidad. Para cada una, se incluirá una breve introducción a su arquitectura y fundamento, así como también una descripción de las facilidades que brinda en la definición de la interfaz de usuario rica en relación a aspectos de presentación, interacción client-side y cliente-servidor y vinculación con datos de la aplicación. Se finalizará con un breve resumen de las diferencias y similitudes entre las tecnologías exploradas.

3.2 Frameworks y Herramientas

Para obtener una visión amplia de las diferentes tecnologías modernas para el desarrollo de Aplicaciones Ricas, se ha investigado y probado un conjunto heterogéneo de las mismas. El objeto de este análisis es capturar los aspectos de las Aplicaciones Ricas que estas herramientas consideran a fin de facilitar su definición y los métodos a través de los cuales la misma puede elaborarse, siendo esto útil para construir los meta-modelos que a futuro describirán formalmente las RIAs.

3.2.1 ZK y Echo2

ZK[7] es un framework disponible para la plataforma J2EE, el cual tiene como objetivo facilitar la construcción de Aplicaciones Ricas que utilizan como plataforma de ejecución HTML Dinámico y AJAX, siendo su principal meta compensar las carencias de la plataforma y automatizar la generación de gran parte de la lógica de interfaz de UI de la aplicación.

El framework ofrece un método de construcción de Aplicaciones Ricas en el cual la interfaz de usuario es definida a partir de componentes y manejada por eventos. La mayor cantidad de trabajo de la aplicación es llevado a cabo en el servidor, delegando en él la lógica de negocios y modelo de datos, y haciendo que el software del lado cliente actúe mayormente como una capa de presentación. El framework mantiene el estado de los componentes de UI tanto en el client-side (DOM) como en el server side (objetos Java) y, salvo algunos casos excepcionales que deben ser explicitados por el desarrollador, la mayoría de los eventos que ocurren en el client-side provocan una invocación al servidor, la cual determina el cambio en el estado de la interfaz de usuario y le indica luego al cliente las actualizaciones locales que debe llevar a cabo en la vista. El hecho de que los eventos sean atendidos de lado servidor posibilita al desarrollador a acceder directamente al backend de la aplicación sin necesidad de ningún otro mecanismo especial, aunque esto puede llegar a representar un costo razonable si se realiza para cada evento que ocurra en la interfaz de usuario.

7 http://www.zkoss.org/

La definición de la UI es llevada a cabo de una manera declarativa, a través de un lenguaje de marcas definido a tal fin, el cual permite describir la estructura y componentes, existiendo también la posibilidad de hacerlo programáticamente desde el server-side. El lenguaje de marcas ofrece atributos adicionales que permiten crear uno o más componentes ejecutando estructuras de control básicas como repetición, iteración o selección condicional.

Las acciones a ejecutar frente a la ocurrencia de eventos son establecidas asignando al tipo de evento particular un método server-side que lo atienda, aunque también pueden asociarse métodos client-side a los mismos. La funcionalidad de estos últimos es bastante limitada y está relacionadas mayormente a animación y mostrar u ocultar elementos de UI dinámicamente. La posibilidad de realizar ciertas validaciones del lado cliente y paginación automática de gran cantidad de datos, dos características comúnmente presentes en RIAs, son algunas posibilidades adicionales que ZK brinda.

Las fuentes de datos de los componentes de interfaz de usuario pueden especificarse declarativamente en la especificación de la UI, o bien programáticamente en el server-side. Esta última puede llegar a implicar la conversión y serialización de los datos representados por objetos en el server-side para que sean recibidos y mostrados correctamente del lado cliente, la cual es llevada a cabo automáticamente por el framework.

Finalmente, se analizó brevemente otro framework llamado Echo2[8], el cual tiene en común con ZK gran parte de sus características, con la salvedad de no proveer métodos declarativos para definir UIs.

3.2.2 GWT

GWT[9] es una tecnología para generar Rich Internet Applications que permite desarrollar el frontend de la aplicación o parte del mismo en lenguaje Java y transformar el código fuente Java en código JavaScript optimizado, facilitando la tarea de escribir RIAs que utilicen DHTML/Ajax para su ejecución. La idea detrás de esta traducción es que el desarrollador trabaje con los IDE existentes para Java y no necesite comprender en detalle JavaScript para crear este tipo de implementaciones. El principal componente de la arquitectura del toolkit es el compilador Java a JavaScript que hace posible la traducción a partir de la cual se obtiene la aplicación final. La comunicación entre el client-side (JavaScript) y server-side (Java) es efectuada a través de Remote Procedure Calls (RPC), un mecanismo que permite invocar desde el lado cliente un método en el lado servidor que, en este caso particular, incluye la serialización de objetos en ambas direcciones, traduciéndolos en objetos compatibles de un lenguaje a otro.

La separación explícita entre cliente y servidor separa claramente la lógica de negocios relativa al servidor de la lógica de interfaz de usuario correspondiente al cliente, brindando mayor interacción al evitar la comunicación innecesaria con el servidor cuando ocurren eventos relativos a la lógica de UI, aunque esto requiere más trabajo en el desarrollo de la interfaz cliente-servidor por parte del usuario. Para mejorar la interacción de la aplicación, GWT brinda la posibilidad de definir métodos que son invocados al completarse la llamada remota, lo que permite continuar con el uso normal de la aplicación mientras la petición se está llevando a cabo asincrónicamente.

8 http://echo.nextapp.com/site/
9 http://code.google.com/webtoolkit/

Las interfaces de usuario ricas son definidas programáticamente, creando instancias de los componentes provistos por el framework. Puede utilizarse GWT para definir la interfaz rica por completo, o bien para incluir componentes ricos en código HTML estático. Los métodos que atenderán eventos de interfaz de usuario son definidos en el client-side, debido a que todo el código Java relativo al mismo será traducido a JavaScript y, por lo tanto, se ejecutará en el browser. Los componentes de datos relativos a la interfaz de usuario deben ser completados con los mismos programáticamente, pudiéndose obtener datos desde los llamados a procedimientos remotos invocados.

3.2.3 OpenLaszlo

OpenLaszlo[10] es una plataforma de desarrollo de RIAs la cual permite especificar aplicaciones que pueden ser compiladas para correr en la plataforma Flash o en DHTML/Ajax. Su arquitectura consiste principalmente en un compilador que genera la aplicación web rica ejecutable, a partir de su descripción en un lenguaje propio denominado LZX. Dicho lenguaje persigue la portabilidad entre plataformas, abriendo paso a la posibilidad de incluir nuevas plataformas destino en el futuro. Para cada plataforma particular, OpenLaszlo define un framework que incrementa su funcionalidad básica y hace más sencilla la traducción.

La definición estructural de la interfaz de usuario es puramente declarativa. La interacción client-side es definida a través del lenguaje JavaScript, aunque en un contexto distinto de su uso asociado a DHTML. También es posible definir nuevos componentes a través de un mecanismo de clasificación, herencia y prototipado, basado en conceptos de la programación orientada a objetos y orientada a prototipos. Los nuevos componentes obtenidos a través de este mecanismo pueden ser instanciados y utilizados en las UI. Al igual que con los frameworks anteriores, OpenLaszlo define interfaces de usuario orientadas a eventos, permitiendo especificar métodos que son invocados frente a la ocurrencia de distintos sucesos.

La plataforma se caracteriza por permitir la especificación de varias características de la aplicación de una manera declarativa, es decir, sin escribir código imperativo, llevando a cabo la actualización de las propiedades y/o contenido de los componentes de manera automática, cuando fuera necesario. Esto permite ahorrar el trabajo manual concerniente a la especificación de dependencias entre elementos de interfaz de usuario. El establecimiento de las fuentes de datos que los componentes de UI utilizan sigue precisamente esta política: para definirlas se utilizan componentes denominados *datasets*, los cuales contienen información o referencian la fuente desde donde la misma es obtenida. Los componentes de datos pueden efectuar consultas sobre estos *datasets* y establecer, de esta manera, su propio contenido. Dado que la única manera de comunicación con el server-side es a través de peticiones efectuadas por este tipo de componentes, se tiene un esquema de división explícita entre cliente y servidor como el establecido en el toolkit GWT. Las acciones de comunicación entre ambas facetas deben ser impuestas obligatoriamente por los desarrolladores lo cual, aunque significa trabajo extra, permite explotar con mayor libertad las capacidades del cliente en cuanto a transferencia de información se refiere.

Finalmente, además del uso de *datasets* mencionado en el párrafo anterior,

10 http://www.openlaszlo.org/

OpenLaszlo brinda la posibilidad de comunicación e interacción con el servidor a través de RPC, pudiendo el desarrollador optar entre las diferentes implementaciones actuales de este paradigma de comunicación cliente/servidor como SOAP, XML-RPC, entre otros. Al igual que GWT, esta tecnología permite definir acciones que se llevarán a cabo al completarse la ejecución de la acción remota.

3.3 Análisis final y conclusión

En términos de interacción y comunicación con el server-side, se halla una dicotomía entre las tecnologías que hacen uso intensivo de invocaciones a la faceta servidor (ZK y Echo2) y aquellas que las efectúan sólo cuando es necesario y, en la mayoría de los casos, con el objetivo de transferir datos (GWT y OpenLaszlo). Las primeras contienen parte de la lógica de interfaz de usuario implementada en la faceta servidor, lo cual implica que aquellos eventos que provoquen cambios estructurales en la UI requieren efectuar una invocación al server-side que llevará a cabo el cómputo de los cambios de estructura, los cuales serán comunicados posteriormente al client-side. Este *overhead* es compensado con una mayor abstracción y facilidad en el desarrollo de RIAs. En primer lugar, dado que la mayoría de los eventos de UI son atendidos en la faceta servidor, los métodos o funciones que los tratan tienen acceso directo al modelo de datos, lo cual le ahorra a los desarrolladores la tarea de explicitar detalles de comunicación entre client- y server-side. En segundo lugar, se utiliza un mismo lenguaje y plataforma para el desarrollo[11]. Por su forma de trabajo, las tecnologías RIA como ZK y Echo2 se aproximan más en su arquitectura a un *cliente liviano* dado que, aunque ofrecen riqueza de interfaz de usuario, gran parte de su procesamiento es efectuado en el server-side. Las tecnologías como GWT y OpenLaszlo reducen al mínimo posible la interacción y dependencia con el servidor, por lo cual se acercan a una arquitectura de *cliente pesado*.

OpenLaszlo y ZK muestran que es posible y deseable la construcción completa de una UI a través de un lenguaje de dominio específico (en este caso, definido con XML). El primero en particular, permite ahorrar gran cantidad de trabajo de desarrollo al detectar automáticamente la dependencia entre componentes y actualizar la UI de modo acorde, detección la cual es inferida desde la interfaz de usuario definida de manera declarativa.

Se observa además que las tecnologías RIA de *cliente pesado* analizadas (GWT y OpenLaszlo), en su afán por permitir explícitamente el control de la comunicación con el server-side, proveen mecanismos para la ejecución asincrónica de peticiones, lo cual permite mejorar los tiempos de respuesta e interacción de la UI.

11 Una excepción a este caso es GWT, framework RIA el cual utiliza el lenguaje Java para implementar tanto el server-side como la faceta cliente (traduciendo para esta última el código Java al lenguaje JavaScript).

4 MODELADO ESPECÍFICO DE DOMINIO (DSM)

En esta sección se describirá la metodología de Modelado Específico de Dominio, sus características, diferencias con metodologías clásicas y otras metodologías similares que se pueden hallar en la actualidad, ventajas provistas y factores negativos.

En primer lugar, se introducirá la metodología y sus aspectos característicos. Acto seguido, se comentarán aquellos aspectos y problemáticas asociados al desarrollo de software que motivan en la actualidad su adopción. Luego, serán descriptas sus características con mayor grado de detalle, así como también las ventajas que la metodología provee en todo lo relativo a la construcción y diseño de software. Finalmente, se comentarán los preceptos que su adopción promueve e implica.

4.1 Introducción

El Modelado Específico de Dominio o Domain Specific Modeling (DSM) es una metodología de desarrollo de software en la cual se busca incrementar el grado de abstracción en la representación de las aplicaciones o sistemas más allá de los lenguajes de programación convencionales, utilizando conceptos y reglas tomados directamente del dominio del problema de software a resolver. La metodología propone comenzar el desarrollo con una definición formal de los conceptos y reglas que caracterizarán a las representaciones de alto nivel (modelos), las cuales serán el artefacto principal de especificación de software. La obtención de representaciones de un menor nivel de abstracción como, por ejemplo, código fuente, son logradas a través de un proceso de derivación, el cual toma como entrada los modelos formales confeccionados de acuerdo a los conceptos y reglas definidos. Esta derivación es posible gracias a la cota que se pone en la cantidad de aspectos considerados del problema de software a tratar, la cual conforma el *dominio específico* de los modelos y promueve un incremento en la semántica de los conceptos y reglas utilizadas en las descripciones. Tal incremento abre paso a una generación de código completa, en el sentido de que los desarrolladores no requieren proveer más información que los mismos modelos a fin de obtener la implementación final, a costas de acotar las soluciones que se pueden obtener a un dominio particular.

En contraste con otros procesos de desarrollo de software, la arquitectura de la metodología DSM requiere, por parte de unos pocos desarrolladores con experiencia en el dominio particular, la creación de 3 elementos principales que serán la herramienta fundamental del equipo que llevará a cabo la construcción del software:

1. Un lenguaje de dominio específico formal a través del cual se definen las soluciones.
2. Un generador de código para el lenguaje anteriormente especificado.
3. Un framework de dominio que sirva como base al código generado en el punto anterior para que la traducción sea más sencilla.

Un dominio, según la metodología, es definido como un área de interés en relación al desarrollo de software, y puede tratarse de situaciones comunes y acotadas a una problemática particular en dicho desarrollo como, por ejemplo, persistencia, diseño de

interfaces de usuario, comunicación, etc., así como también puede referirse a un campo específico para el cual el mismo está destinado como telecomunicaciones, procesos de negocios, control de robots, manejo de ventas, etc. Cuanto más se acote el dominio, más posibilidades habrá de generar aplicaciones funcionales directamente de las representaciones de alto nivel, dado que mayor será el contenido semántico de los componentes y reglas que las conforman.

En el desarrollo basado en modelos de dominio específico, los modelos son los principales artefactos de especificación de software, permitiendo a través de los mismos incrementar la abstracción y ocultar la complejidad, obteniéndose las implementaciones finales a partir de ellos. Los elementos con los que se construyen los modelos están definidos a través de un modelo particular denominado meta-modelo, necesitando éste a su vez ser definido usando un lenguaje particular. Este último es conocido como meta-meta-modelo.

4.2 Motivación

La historia del desarrollo de software ha mostrado que los incrementos en la productividad en el mismo han estado asociados mayormente a incrementos en la abstracción de las especificaciones utilizadas para describirlo. Sin embargo, a comparación de cómo la aparición de los lenguajes de tercera generación (3GLs) contribuyeron a incrementar la productividad hace algunas décadas, los lenguajes de modelado y programación tradicionales en la actualidad están contribuyendo relativamente poco en este campo[1][28]. Es en este punto en donde la metodología de Modelado Específico de Dominio intenta innovar, persiguiendo un incremento en la abstracción más allá del provisto por los lenguajes de programación y técnicas de modelado genéricas actuales, utilizando para describir la solución conceptos del dominio del problema de software a tratar.

DSM surge como una solución a la *desconexión* entre la etapa de modelado y de implementación de las metodologías actuales. En la primera, se generan un conjunto de modelos para abstraer aspectos de implementación que son complejos o muchas veces desconocidos en las momentos tempranos del desarrollo. Luego, en la etapa de implementación subsiguiente, los modelos son interpretados por los programadores, quienes manualmente convierten las descripciones abstractas en implementaciones concretas. Esta traducción manual e informal, es propensa a errores frecuentes en el desarrollo, los cuales resultan en una disminución de la calidad del software y en la productividad alcanzada durante su construcción.

La separación entre las etapas de modelado e implementación implica también el mantener la consistencia entre modelos e implementaciones: siendo manual la tarea de conversión entre ambos, ocurre que cualquier cambio en alguna de ellas implica una modificación en su contraparte, lo cual nuevamente deberá ser una tarea manual y, por lo tanto, propensa a errores. Dado que el costo de mantener los modelos actualizados es mayor que el beneficio inicial obtenido por los mismos, usualmente las actualizaciones no son efectuadas al cambiar aspectos en la implementación, lo cual a los torna inservibles como documentación.

Mantener actualizados los modelos y la implementación automáticamente rara vez es posible en su totalidad y el grado en que esto puede llevarse a cabo depende de cuán

parecido sean los modelos e implementaciones entre sí. Dada la separación ya mencionada, ocurre que los modelos contienen aspectos demasiado abstractos que no pueden ser inferidos directamente de la implementación, teniendo esta última a su vez definiciones que no pueden ser descriptas en los modelos o pueden representarse con más de una estructura distinta en el mismo. Por otro lado, filtrar y organizar información de la implementación y mostrarla de una manera distinta (por ejemplo, gráficamente) no agrega ninguna semántica a la representación; en todo caso, la hace más amena a la vista.

En la Figura 2 se describen esquemáticamente diferentes maneras de relacionar el modelo con el código fuente y la aplicación final, incluyendo la metodología DSM. Las alternativas más usuales en la actualidad son:

(a) No incluir ningún modelo, lo cual es razonable para software de tamaño reducido.

(b) Se incluyen modelos que luego son descartados, debido a que el costo de mantenerlos actualizados es mayor a los beneficios que aportan.

(c) Se utilizan modelos para visualizar el código mediante alguna técnica de ingeniería inversa, lo cual puede facilitar su comprensión, pero no agrega semántica alguna al mismo.

(d) Ida y vuelta entre modelos y código, intentando actualizar los cambios producidos en cualquiera de los dos en su contraparte. Esto es posible de manera automática sólo cuando los formatos son estructuralmente parecidos o cuando se actualizaran los aspectos comunes entre ambas partes.

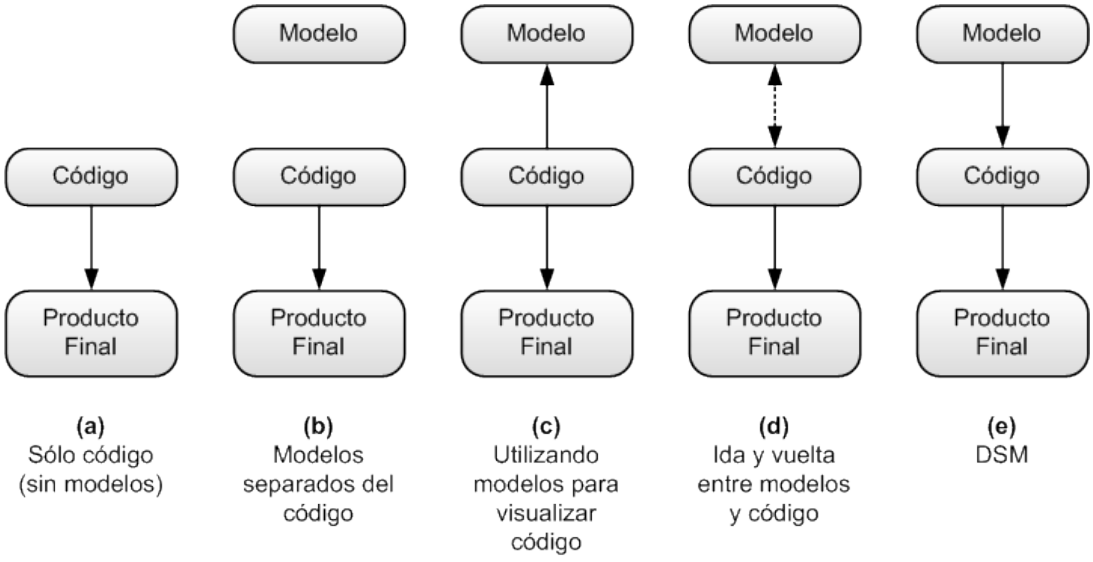


Figura 2. Distintas formas de relacionar código y modelos, entre ellas, la promovida por DSM

La metodología DSM (e), a diferencia de los casos anteriores, tiene a los modelos como la fuente de información fundamental para derivar el código, obteniéndose luego la implementación final a partir de este último. Al generarse el código fuente desde los modelos, los problemas anteriormente comentados a causa de la separación entre las etapas de modelado e implementación no se presentan.

En la actualidad, el hecho de que los ampliamente usados lenguajes de Tercera Generación permitan, a través de compiladores, generar código de bajo nivel confiable el cual no tiene que ser modificado por los desarrolladores, muestra que esta derivación es factible y además conlleva a un incremento considerable en la productividad.

4.3 Características

A continuación, se enumerarán y describirán brevemente las características principales que definen a la metodología.

4.3.1 Dominio acotado

El hecho de que un lenguaje se base en representar soluciones con un conjunto acotado de conceptos, relaciones y reglas provenientes del dominio del problema, permite que especificaciones y dominio estén más próximos. Dada esta proximidad, se dispone de mucha más información acerca de los elementos y relaciones utilizados para describir la solución, lo cual incrementa la semántica de las representaciones de alto nivel concebidas. Los generadores de código que utilizan como entrada estas representaciones disponen, por ende, de mucha más información, la cual puede ser utilizada para obtener la implementación final de un modo directo, sin etapas intermedias ni manuales, evitando los inconvenientes inherentes a éstas.

Cuanto más acotado sea el dominio del lenguaje, mayor será la semántica que los modelos confeccionados con el mismo contendrán y, por ende, mejores posibilidades habrá de derivar implementaciones funcionales completas desde los mismos. Como contrapartida, la cota que debe ponerse en el dominio del problema a tratar hace difícil utilizar los lenguajes y generadores fuera del contexto de la compañía u organización para la cual se lleva a cabo el desarrollo. Sin embargo, aquellos meta-modelos orientados a expresar soluciones a problemáticas recurrentes en la construcción de software (por ejemplo, especificación de interfaces de usuario para aplicaciones de escritorio) brindan una mayor posibilidad de reuso en diferentes proyectos de desarrollo, estando igualmente acotados a un dominio particular.

La cercanía entre dominio y especificación también permite guiar a los desarrolladores y validar errores en la etapa de modelado, utilizando para esto reglas derivadas del mismo dominio. Estas reglas pueden ser expresadas directamente en el meta-modelo, lo cual permitiría controlar su validez en tiempo de modelado. Alternativamente, las mismas pueden ser incluidas como parte de los generadores de código, los cuales corroborarán su cumplimiento en tiempo de derivación.

4.3.2 Alto nivel de abstracción

Como ya se dijo con anterioridad, la metodología de DSM incrementa el grado de abstracción en las representaciones de software, al especificar la solución al problema a tratar con conceptos tomados del dominio inherente al mismo. Esto se traduce en un incremento en la productividad al largo plazo. La productividad alcanzada implica menor tiempo y recursos para desarrollar el software, así como también facilita el mantenimiento.

Los generadores permiten derivar modelos a especificaciones en el dominio de la

solución al problema como código fuente, archivos de configuración, scripts de compilación, otros modelos, etcétera. Esto implica la posibilidad de no solo generar el código fuente que implementará el software, sino también de derivar especificaciones relativas a distintas tareas o etapas del desarrollo como testing, prototipado, etc.

4.3.3 Lenguaje (meta-modelo)

Los meta-modelos especifican reglas sintácticas que deberán respetar los modelos, así como también definen aspectos semánticos, los cuales son cercanos al dominio y son representados por los elementos utilizados en los modelos y sus relaciones. En los modelos de dominio específico, la asociación entre conceptos del dominio del problema y conceptos del lenguaje confeccionado es sencilla y usualmente directa. Para representar todos los aspectos concernientes al software, los meta-modelos deben permitir especificar tanto características estáticas y estructurales como dinámicas, basadas estas últimas generalmente en algún modelo de cómputo subyacente ya existente como, por ejemplo, Diagramas de Transición de Estados.

El lenguaje de modelado puede estar compuesto por dos o más lenguajes separados que consideren distintos aspectos del software a especificar. Los diferentes modelos generados con los distintos lenguajes son integrados al momento de la derivación. La vinculación entre modelos puede llevarse a cabo utilizando en cada uno conceptos definidos en otros modelos relacionados (el meta-modelo en este caso, aunque fragmentado en distintos aspectos, es único) o bien utilizando conceptos a modo de *proxy* o *puente*. La fragmentación de distintos aspectos del lenguaje y su integración posterior facilita la modularización y el trabajo concurrente de los desarrolladores.

Los meta-modelos describen formalmente los conceptos que los modelos podrán incluir, sus relaciones, propiedades, jerarquías y reglas de corrección asociadas. Según ha sido probado, como se menciona en [1], al menos 4 niveles de instanciación son necesarios en el Modelado Específico de Dominio: el nivel más concreto es representado por la aplicación funcional corriente, la cual es una instancia de un modelo particular, el cual es definido a través de un meta-modelo, representando este último una instancia concreta de un meta-meta-modelo. Cada *instanciación* puede verse como la definición de un modelo que utiliza los conceptos y relaciones y respeta las reglas definidas por el modelo superior (su meta-modelo). En la Figura 3, pueden apreciarse gráficamente la relación entre estos 4 niveles de instanciación en meta-modelos y modelos particulares comúnmente utilizados. La filosofía de DSM requiere que los modelos presentes en cada uno de estos niveles sean formales, es decir, que estén definidos a través de algún meta-modelo formal. La existencia de herramientas que permitan el desarrollo de estos meta-modelos de manera sencilla y sin imponer restricciones particulares a algún dominio o área es también un requerimiento de base impuesto por la metodología. Estas herramientas son creadas y/o configuradas por expertos en el dominio y utilizadas por la mayor parte de los desarrolladores.

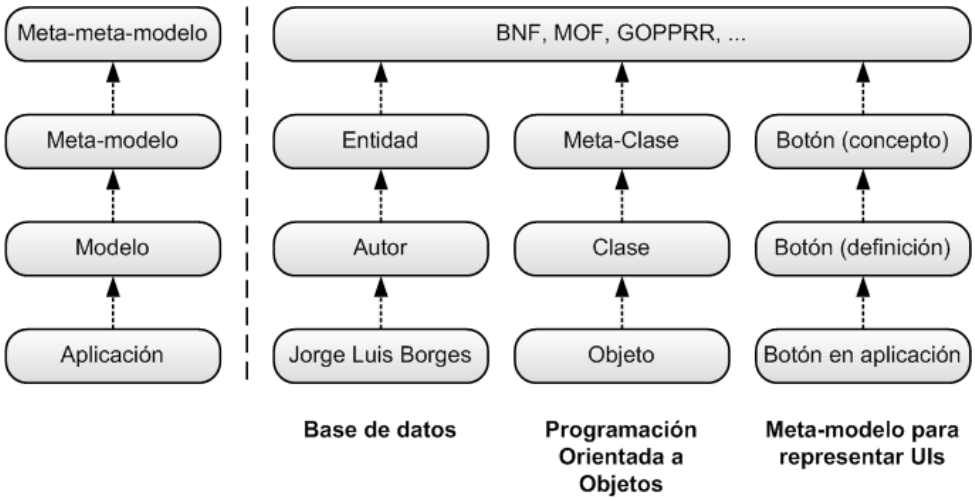


Figura 3. Los 4 niveles de instanciación, ejemplificados en distintas circunstancias. Las flechas punteadas indican una relación del tipo *es instancia de*.

4.3.4 Generación de código

El objetivo de DSM es la obtención de la aplicación final completa a través de la generación de código, siendo la misma posible gracias al alto nivel de semántica que los modelos poseen. Sin embargo, esto no quiere decir que la escritura de código se evada en un ciento por ciento, sino que los desarrolladores deberán indefectiblemente codificar a fin de construir el framework que servirá de soporte al código generado.

La metodología no considera correcto el comprender y modificar el código obtenido desde los generadores, por los motivos ya comentados en relación a las dificultades asociadas a mantener la consistencia entre implementación derivada y modelos de alto nivel. El efectuar estos cambios sería análogo a modificar el código máquina o ensamblador obtenido desde compiladores en los lenguajes de Tercera Generación. Si el dominio es capturado de manera correcta y completa, y tanto generadores como framework de dominio son escritos adecuadamente, cualquier personalización extra puede ser conseguida a través de cambios sobre estos dos últimos componentes. Las modificaciones efectuadas serán plasmadas finalmente en la implementación al ser la misma regenerada desde los modelos que especifican al software.

Los generadores deben considerar tanto la producción de código estático o de estructura así como también de comportamiento, siendo este último el más difícil de derivar. Hacer portable un modelo a diferentes dominios de solución (por ejemplo, plataformas de ejecución) implica la escritura de distintos generadores, uno para cada dominio destino particular, la cual deberá estar a cargo de programadores expertos en el área. Hacer efectiva esta portabilidad usualmente implica la elaboración de un framework que facilite la ejecución y traducción de las soluciones generadas a partir de modelos para cada plataforma destino deseada.

De acuerdo a la metodología, los generadores deberán derivar código de producción, lo cual es posible gracias al ya comentado contenido semántico presente en las especificaciones, en conjunción con la gran experiencia en el dominio requerida por parte de los creadores de la solución DSM, la cual es volcada directamente en generadores y

framework de dominio. Estos dos componentes, en conjunto con la plataforma de ejecución, son invisibles a los desarrolladores de igual manera en que lo es el código máquina generado por los compiladores en los 3GLs actuales.

El funcionamiento de los generadores consiste básicamente en acceder y navegar los modelos, extraer información de los mismos (transformándola de ser necesario) y producir una salida correspondiente. Si se utilizaran varios lenguajes (meta-modelos) que consideraran distintos aspectos del dominio en la especificación, la integración entre los diferentes modelos puede ser llevada a cabo en última instancia por los generadores.

Finalmente, los derivadores son capaces de generar no sólo representaciones de menor abstracción como, por ejemplo, código fuente, sino también pueden realizar variadas tareas relativas al desarrollo como creación de build scripts, deploying, invocación de compiladores, generación de archivos de configuración, packaging, toma de métricas, etc. Los chequeos de validez finales en los modelos también pueden ser tareas adicionales que los generadores pueden llevar a cabo. Si bien la validación en tiempo de modelado ahorra muchas de las tareas de prueba de la aplicación final, aún así puede derivarse código de testing si el mismo fuera necesario.

4.3.5 Representaciones no necesariamente textuales

Bajo la metodología de Modelado Específico de Dominio se consideran una variedad de representaciones posibles además de las textuales como, por ejemplo, diagramas, tablas y matrices. Las primeras 3 pueden ser igual de formales que las representaciones en texto, dado que establecen conceptos y reglas concretos y no ambiguos para componer especificaciones, siendo usualmente más amenas para construir desde el punto de vista del desarrollador y brindando mayor protección frente a errores semánticos (por ejemplo, utilizando conexiones gráficas para referencias en lugar de hacer las mismas utilizando identificadores). Características de las representaciones gráficas como conexiones entre componentes, por ejemplo, permiten alcanzar especificaciones de mayor riqueza, haciendo que las mismas sean más seguras y llevando a cabo un mejor reuso de componentes. La representación a escoger es aquella que se considere más conveniente para el tipo de especificación que se busca definir.

4.3.6 Framework de dominio

El framework de dominio funciona a modo de interfaz entre el código generado y la plataforma subyacente. Su objetivo principal es incrementar la abstracción del ambiente en donde se ejecutará la aplicación, de manera de que la generación de código sea más simple. Su tamaño puede variar desde ser prácticamente inexistente, hasta proveer una abstracción importante al entorno de ejecución del software derivado, dependiendo de cuánto se busque simplificar el trabajo de los generadores y cuán alto sea el grado de la abstracción que el mismo entorno brinde por defecto. Dentro de las ventajas que el uso de un framework de dominio implica se encuentran el hecho de forzar ciertos aspectos de diseño que el código generado debe respetar, ocultamiento del entorno de ejecución subyacente y el eliminar la duplicación de código al derivar.

Es recomendable que para cada plataforma de ejecución existan varios frameworks, donde cada uno esté orientado a la implementación de distintos tipos de software.[1]

4.4 Ventajas provistas por la metodología

La metodología DSM ofrece diferentes ventajas de acuerdo a cómo descompone y organiza el proceso de desarrollo de las aplicaciones. A continuación, se enumeran las mejoras que la filosofía de Modelado Específico de Dominio ofrece en distintas áreas y tareas del desarrollo.

4.4.1 Uso de modelos específicos

Dado que los modelos son un artefacto de especificación de software necesario e inevitable en la mayoría de los proyectos de desarrollo, la metodología DSM permite evitar la necesidad de comprender la semántica particular de otros lenguajes de modelado (por ejemplo, UML, E-R, etc.), y su consecuente requerimiento de asociar conceptos del dominio del problema a conceptos particulares del lenguaje. La proximidad del lenguaje de modelado al dominio del problema hace que esta asociación sea natural, menos ambigua y más directa que en los lenguajes de modelado genéricos. Sin embargo, es pertinente mencionar que el diseñador del lenguaje de modelado tendrá a cargo la tarea de analizar y representar los conceptos del dominio, utilizando constructores del meta-lenguaje elegido para describir el meta-modelo, siendo sí esta tarea de asociación inevitable para la metodología.

Las reglas de validez estipuladas en los modelos provienen del dominio del problema a tratar y permiten detectar errores en la etapa de modelado, donde es más sencillo corregirlos. Estas reglas son independientes de los generadores, por lo que cualquier generador que se utilice recibirá como entrada modelos válidos, lo cual tiene como consecuencia una reducción de la complejidad y lógica necesaria en los mismos. A su vez, el hecho de que los generadores sean más sencillos facilita su confección para distintos propósitos durante el desarrollo como, por ejemplo, prototipado, en pos de alcanzar una mayor productividad. Las reglas gramáticas guían el modelado, no haciendo posible a los desarrolladores la construcción de modelos inválidos. El hecho de que estas reglas provengan directamente del dominio también facilita su comprensión.

4.4.2 Código fuente

Los generadores de código facilitan la ejecución de tareas rutinarias, de manera que los desarrolladores puedan focalizarse en aspectos más interesantes del desarrollo en lugar de en detalles de implementación. Los encargados de los aspectos de implementación serán los autores de los generadores de código, los cuales serán expertos en el área y trasladarán implícitamente sus conocimientos a toda la implementación final derivada por medio de los generadores.

Un alto nivel de abstracción en las especificaciones se traduce en menos esfuerzo para describir el software, aunque es necesario invertir trabajo extra en el desarrollo de los generadores de código, meta-modelos y frameworks. Este esfuerzo inicial puede ser amortizado considerablemente debido al incremento en la productividad en el desarrollo alcanzado.[1]

4.4.3 Calidad, evolución y mantenimiento

Debido a que el código fuente es producido por los generadores elaborados por el equipo de algunos pocos expertos en el área, los cuales conocen tanto el dominio del problema como la tecnología en la que se implementará, el código derivado se supone libre de errores y eficiente en el uso de recursos de ejecución y memoria, por lo que no necesita ser modificado u optimizado. Adicionalmente, todo el código generado seguirá un único estilo de programación y diseño.

Es necesario menos testing, al contener los lenguajes definidos para describir el software reglas que realizan verificaciones antes de derivar implementaciones ejecutables. Estos controles aseguran una detección y solución temprana de errores, dado que son efectuadas en la etapa de modelado. Al utilizar lenguajes no textuales, se pueden evitar errores tipográficos, de referencias entre componentes, variables no declaradas o inicializadas, etc. Los detalles de código fuente son responsabilidad de los pocos expertos involucrados en la construcción de la solución DSM. La documentación, código relativo a testing, debugging, etc., pueden derivarse como artefactos adicionales a partir de los modelos confeccionados y cumplirán con los preceptos de calidad buscados, por la misma razón que el código fuente lo hará.

Dada la cercanía de los modelos con el dominio del problema a resolver, la metodología DSM soporta naturalmente la evolución de la arquitectura subyacente (plataforma de ejecución y framework de dominio) sobre la cual se implementa el sistema, reutilizando en su totalidad o en gran parte los modelos generados. De este modo, el software especificado es atemporal en términos tecnológicos y las representaciones de alto nivel que lo definen siguen siendo válidas mientas el dominio del problema siga siendo el mismo.

La proximidad modelo-dominio también tiene como ventaja el hecho de que los modelos funcionen como artefactos tanto de captura de requerimientos como de especificación de software. En consecuencia, los clientes participan con mayor actividad en el desarrollo, pudiendo inclusive colaborar directamente en la construcción de las representaciones que lo definen. Esto último, en conjunto con el rápido prototipado a partir de modelos que la metodología facilita, contribuye de manera importante a asegurar que el software especificado sea válido en término de requerimientos.

4.4.4 Un conjunto mayor de potenciales usuarios

El alto nivel de abstracción que brindan los modelos para componer software amplía el conjunto de usuarios que pueden participar del desarrollo, incluyendo a personas que pueden no tener conocimientos específicos de programación como, por ejemplo ingenieros o directivos. Estos tipos de usuarios pueden utilizar lenguajes de modelado que consideren los elementos fundamentales de su tópico y reglas para relacionarlos y validar su consistencia. La posibilidad de que analistas y diseñadores puedan crear especificaciones completas de software facilita también la interacción entre ambos. En consecuencia, se obtiene una mejor intercomunicación entre los diferentes agentes involucrados en el desarrollo de la aplicación o sistema. Finalmente, los únicos pocos desarrolladores que necesitan conocer aspectos de bajo nivel relativos a generadores, plataforma y frameworks subyacente son los expertos en el dominio a los cuales se les asigna la tarea de crear la

solución DSM.

4.4.5 Productividad

Un mayor grado de abstracción en la especificación del software conlleva a un incremento en la productividad durante su desarrollo. La metodología DSM intenta explotar al límite la misma productividad que ha proporcionado la abstracción presente en los lenguajes de Tercera Generación actuales frente a la programación directa en código máquina, haciendo que las especificaciones de software sean construidas con conceptos del dominio del problema a tratar. Dado el dominio acotado y el consecuente alto grado de semántica contenido en los modelos, se necesita menos cantidad de estos para especificar software a comparación de aquellos confeccionados con lenguajes de modelado genéricos como UML. Estos lenguajes buscan ser aplicables a un amplio espectro de dominios, lo cual obliga a que la semántica de sus conceptos y reglas sean lo suficientemente débiles como para cubrir una gran cantidad de áreas de aplicación. Esta misma debilidad semántica tiene como resultado modelos que aportan poca información significante para los generadores de código, los cuales se ven limitados en su tarea y requieren que las representaciones de bajo nivel que derivan sean completadas manualmente por desarrolladores. Como resultado, los programadores tienen que interpretar y refinar código fuente que no fue escrito por ellos mismos y, además, deben mantener actualizados los modelos si las modificaciones en las representaciones de bajo nivel de abstracción que efectuaran implicaran cambios en estos. En [1] se resumen distintos casos reales en donde fue posible la toma de métricas, los cuales reportan ganancias de entre 300% y 1000% en la productividad utilizando soluciones DSM en dominios específicos correctamente capturados.

A diferencia de otras metodologías, en el Modelado Específico de Dominio, la productividad que puede alcanzarse se incrementa durante el desarrollo, a medida que los desarrolladores se van familiarizando con las herramientas de modelado y los meta-modelos. En las etapas finales, los bugs evitados de manera anticipada con las validaciones llevadas a cabo en modelos y con la generación automática de código, la facilidad para implementar cambios de requerimientos con sólo modificar modelos existentes y la posibilidad de portar el software a distintas plataformas a través de diferentes generadores implican incrementos extra en la productividad en etapas ya avanzadas del desarrollo. Para que este incremento en la abstracción sea factible, se hace notar nuevamente la necesidad de que la definición de la arquitectura DSM (meta-modelos, herramientas y framework de dominio) sea llevada a cabo por expertos en el área, de manera de reducir al mínimo posible la necesidad de cambios estructurales o de gran envergadura durante la especificación.

La agilidad de la metodología, proveniente de la cercanía entre modelos y requerimientos, facilita la adaptabilidad del producto y la especificación de cambios en los requerimientos. El incremento en la productividad implica un tiempo de desarrollo menor, cuyas consecuencias son una reducción de costos y del ciclo de retroalimentación con clientes y una mejora en términos de competitividad, dada la posibilidad de lanzar productos al mercado más rápidamente.

Aspectos relativos a la calidad y mantenimiento del producto (ya descriptos en detalle en la sección *Calidad, evolución y mantenimiento*) y a mejoras asociadas a los costos de capacitación necesarios y aprovechamiento del conocimiento de expertos en el

área (enunciados en la siguiente sección, *Desarrolladores y experiencia*), tienen un impacto adicional en el incremento de la productividad.

4.4.6 Desarrolladores y experiencia

En la mayoría de los proyectos de desarrollo de software, la cantidad de desarrolladores no es directamente proporcional a la magnitud de productividad alcanzada debido a diversos factores. Un ejemplo usual de este tipo de factores es la necesidad de capacitación inicial a desarrolladores novatos en las tecnologías utilizadas en la construcción de la implementación y en aspectos internos concernientes a cómo el dominio es representado a través de las mismas. Bajo la filosofía impuesta por DSM, son los expertos en el área quienes definen los meta-modelos y construyen los derivadores que realizarán la asociación entre conceptos de dominio e implementación. Los meta-modelos obligan a los desarrolladores a seguir preceptos de diseño como guía en la construcción del software, y su semántica resulta más fácil de comprender que aquella presente lenguajes de modelado no específicos como UML. Los derivadores, por otro lado, generarán código eficiente y correcto que los mismos expertos especificarán. Por esto último, se tiene una mayor cantidad de influencia implícita de los expertos en el producto final en comparación a otras metodologías clásicas que requieren comunicación directa entre integrantes experimentados del equipo y desarrolladores en entrenamiento. El Modelado Específico de Dominio impulsa también una inserción y capacitación más rápida de los integrantes del equipo de desarrollo, dado que los desarrolladores tienen que involucrarse sólo con aspectos del dominio del problema a resolver y, además, son asistidos en sus tareas por las herramientas de modelado creadas por los autores de la solución DSM.

4.4.7 Aspectos económicos

La decisión de utilizar la metodología DSM dentro de una organización involucra tanto aspectos económicos como tecnológicos. Las alternativas más usuales son el crear una solución DSM desde cero, o bien utilizar una ya existente y pública, adaptándola a las necesidades que se requieran si fuera necesario y posible.

A pesar de que las soluciones DSM son generalmente un capital importante de la organización que la concibe y, por lo tanto, no se busca que la misma adquiera carácter público, existen casos en los cuales su difusión es de utilidad o importancia. Algunos de estos casos son el facilitar la comunicación e intercambio de información entre diferentes sectores de la organización o con subsidiarias, o el simplificar el desarrollo de software bajo plataformas o HW particulares que la misma empresa produce. El uso de soluciones DSM existentes, aunque implica ahorro de costos en la creación de una solución *in-house*, puede no llegar a brindar la flexibilidad necesaria que se requiere.

El introducir la metodología en una empresa u organización representa una inversión, dado que implica un costo inicial sin obtención de ganancia alguna en primera instancia, obteniéndose esta tiempo después a través de las ya mencionadas ventajas alcanzadas en términos de productividad, calidad del producto obtenido, comunicación, etc. El costo inicial a afrontarse es representado por la confección de los componentes de la arquitectura de DSM (meta-modelos, herramientas, generadores, framework de dominio, etc.).

La viabilidad de la introducción de una solución DSM en una empresa aumenta proporcionalmente a la cantidad de *repetición* que se tenga en los desarrollos de software llevados a cabo dentro de la misma, ya sea en términos de distintos productos con características en común o bien en versiones, configuraciones o funcionalidades similares de un producto en particular. Esta viabilidad proviene del hecho de que, mientras más tareas similares haya dentro del mismo dominio considerado por la solución DSM, más se aprovecharán sus ventajas y, por ende, más ganancias resultarán de la inversión inicial efectuada.

Dadas las mejoras que se pueden obtener en términos de comunicación y capacitación en el equipo de desarrollo, la existencia de gran cantidad de desarrolladores es también un incentivo extra para introducir la metodología. En estos casos, la distribución de trabajo puede ser distinta en comparación a metodologías clásicas, debido a que los desarrolladores no necesitan conocer aspectos técnicos del software, los cuales son debidamente considerados por los creadores de la solución DSM. La introducción de la solución puede ser incremental, intercalando la misma con otras metodologías asociadas a la programación manual, a fin de no incurrir en pérdidas relativas a la espera de que las herramientas adecuadas estén disponibles.

Según se menciona en [1], es usual que ocurran cambios en la misma solución DSM a medida que el desarrollo avanza. Sin embargo, los mismos son introducidos con mayor facilidad y productividad que en otras variantes de desarrollo, dado que involucran esfuerzo por parte de unos pocos desarrolladores (los expertos en el dominio), mientras que el resto puede continuar con sus tareas. Adicionalmente, dado que el mantenimiento es efectuado sobre la solución DSM, el mismo no depende de la cantidad de desarrolladores ni de el tamaño del producto. No obstante, es pertinente aclarar que cambios de envergadura en el meta-modelo pueden dejar obsoletos gran parte de los modelos confeccionados hasta el momento, los cuales deberán ser ajustados o redefinidos adecuadamente. Por esta razón, la metodología considera indispensable que la definición de los meta-modelos sea llevada a cabo por desarrolladores expertos en el dominio, de manera de asegurar la menor cantidad posible de cambios en los mismos a futuro.

Finalmente, si no se dispone de expertos en el área dentro de la organización, una opción viable es el contratar a un equipo de consultores que hagan un análisis del dominio y confeccionen la solución DSM adecuada. Sin embargo, por lo ya dicho, es preferible que los creadores de dicha solución sean especialistas en el dominio a tratar.

4.5 Preceptos implicados por la metodología

4.5.1 Modelos

Los modelos en DSM deben abstraerse de la estructura del código que generan, por lo cual no debe pensarse en representar código fuente al diseñar el lenguaje de modelado, sino en plasmar aspectos del dominio del problema que luego serán trasladados a una implementación particular. A su vez, los modelos deberán ser lo suficientemente formales y ricos en semántica como para permitir generar, a partir de su contenido, distintos artefactos de software como, por ejemplo, código fuente, scripts de compilación, código de testing, etc. Más allá de su formalismo y semántica inherente, al igual que en las metodologías

clásicas, los modelos permiten representar de manera abstracta la aplicación desarrollada y sirven como documentación y como herramienta de comunicación.

4.5.2 Generación de código

Como ya se comentó y justificó adecuadamente, la generación de código debe ser total y el código obtenido no debe tener que modificarse, al igual que ocurre al compilar programas escritos en lenguajes de alto nivel en representaciones ejecutables en la actualidad. Además, el código generado debe ser de calidad y eficiente, características las cuales son alcanzables gracias al dominio acotado del lenguaje y al conocimiento especifico del mismo por parte de los desarrolladores, plasmado en los generadores.

4.5.3 Herramientas de meta-modelado: necesidad y características

Las herramientas de meta-modelado son un elemento fundamental para la metodología DSM, ya que hacen posible la definición formal de los meta-modelos y la automatización de la generación de código. Como requerimiento primario, las mismas deben proveer total libertad a los desarrolladores para definir meta-modelos, sin asunción o restricción alguna, permitiendo la definición de lenguajes que puedan ajustarse por completo a las necesidades del dominio. Para esto, debe brindarse un lenguaje de meta-modelado que haga posible el definir componentes y sus relaciones y restricciones de manera totalmente abstracta.

La inclusión de lenguajes que no son extensibles o personalizables, característica presente en la mayor parte de las herramientas CASE existentes, no permite la flexibilidad que la metodología DSM requiere, dado que no facilita el ajustar el lenguaje al dominio específico. La diferencia entre las herramientas CASE convencionales y las orientadas al meta-modelado se esquematizan en la Figura 4.

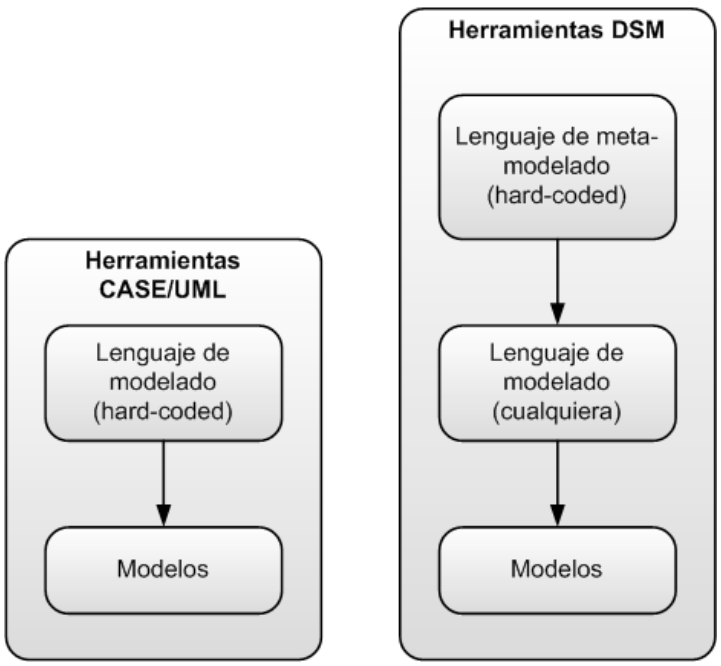


Figura 4. Comparación entre herramientas CASE/UML convencionales y herramientas para DSM

Adicionalmente, es deseable que las herramientas de meta-modelado sean seguras, es decir, que protejan la integridad de los modelos siempre que sea posible (por ejemplo, frente a modificaciones en el meta-modelo).

4.6 Diferenciación con otras metodologías centradas en modelos

Existen en la actualidad una gran cantidad de lenguajes de modelado de propósito general los cuales han sido estandarizados y han logrado amplia difusión, siendo el más conocido UML. Lamentablemente, ninguno de estos lenguajes fueron concebidos para la generación automática de código o representaciones de más bajo nivel a partir de modelos, dado que no responden a un incremento suficiente en el grado de abstracción de las especificaciones que permiten obtener[1]. La causa de este bajo nivel de abstracción se debe a que sus conceptos y relaciones son demasiado generales, lo cual hace que su semántica sea débil.

El lenguaje de modelado UML surge como una manera de acordar conceptos de modelado, su notación y símbolos utilizados, por lo cual nunca estuvo en sus planes la generación automática de código ejecutable. Los conceptos como clases, métodos, atributos, etc., son tomados del dominio del código fuente, no del dominio del problema a resolver. Esto deriva en el hecho de que, aún usando por completo y adecuadamente los conceptos que UML provee, solo una porción muy pequeña del código puede ser generada a partir de los modelos confeccionados en este lenguaje. Existen también casos en los cuales se ha podido alcanzar una mejor generación de código a partir de UML, pero en estos la estructura del lenguaje (meta-modelo) y el significado de sus conceptos han tenido que ser modificados[1].

La especificación de lenguajes de modelado ejecutables también ha sido considerado desde la década pasada. En este tipo de meta-meta-modelos, se utiliza usualmente un lenguaje textual en conjunto con modelos específicos para describir restricciones, cambios de estado o acciones, incluyéndose en algunos casos el uso directo de un lenguaje de programación. Todo esto obliga a los desarrolladores a comprender no sólo el lenguaje de modelado, sino todos los adicionales, incluyendo alguno extra que permita especificar acciones con mayor grado de refinamiento, si con todo lo anteriormente mencionado no puede representarse lo que se desea. Finalmente, en estos casos la abstracción lograda por los modelos es pobre, debido a que los mismos deben incluir indefectiblemente aspectos menos abstractos y más relacionados con el código que con el dominio del problema de software a tratar.

En el año 2003, la OMG lanza la Arquitectura Dirigida por Modelos (Model-Driven Architecture o MDA), asumiendo directamente que la generación completa de código a partir de UML no es posible. MDA utiliza tres tipos de modelos: modelos independientes del cómputo (CIM), modelos independientes de la plataforma (PIM) y modelos específicos a una plataforma de ejecución (PSM). En su forma más básica, la metodología implica la transformación, posiblemente automática, de uno o más modelos UML a otros modelos UML, generando código a partir del último modelo obtenido en la secuencia de transformaciones (el menos abstracto). Sin embargo, la metodología usualmente implica el hecho de que en cada transformación sea necesario la extensión de los modelos obtenidos por parte de los desarrolladores, lo cual atenta claramente contra la generación de código (o

modelos intermedios) total y sus problemáticas asociadas. Como solución a este problema, se plantea el uso de un mismo lenguaje en todos los niveles de abstracción. Esto último corre con la desventaja de forzar el mismo grado de abstracción en cada uno de los distintos niveles, resultando en un decremento de la abstracción general que proporciona el lenguaje de meta-modelado.

Otra opción para definir meta-modelos bajo UML es enriquecer a este último a través de profiles que permitan clasificar con estereotipos y agregar nuevos tipos de atributos a los elementos de modelo, pudiéndose también especificar restricciones a través de algún lenguaje ideado a tal fin como, por ejemplo, OCL[32]. Aunque esta posibilidad de enriquecimiento hace más ameno a UML para definir modelos, no permite la definición de nuevos tipos completamente diferentes al lenguaje. Debido a estas limitaciones, la OMG ha propuesto una nueva manera de generar modelos, a través de la introducción de un lenguaje de meta-modelado denominado MOF (Meta-Object Facility).

Finalmente, se hace notar que la estandarización de meta-modelos es usualmente de poca utilidad e incluso contradictoria con el propósito para el cual los lenguajes son creados. Esto se debe a que los lenguajes de modelado a partir de los cuales se derivan las implementaciones finales están adaptados a un problema de software particular de una organización o empresa. Sin embargo, en el contexto de las ventajas en cuanto a abstracción y facilidad para comunicar conceptos que el Modelado Específico de Dominio brinda, han surgido meta-modelos estándar aplicados a dominios particulares como, por ejemplo, AUTOSAR en la industria automotriz.

4.7 Conclusión

En esta sección se han comentado brevemente los aspectos característicos de la metodología de Modelado Específico de Dominio (DSM), su motivación subyacente, ventajas y desventajas inherentes, preceptos a respetar para su correcta adopción y su aplicabilidad.

En primer lugar se introdujo el objetivo principal de la metodología, el cual consiste en alcanzar la elaboración de especificaciones de software de alto nivel de abstracción, acercando las mismas al dominio del problema a tratar. Acto seguido, se introducen sus componentes principales: lenguajes de dominio específico, generadores de código y frameworks de dominio.

En cuanto a su motivación subyacente, DSM invoca a la poca contribución aportada por los lenguajes de programación modernos en términos de productividad alcanzada en el desarrollo. Como solución, busca fortalecer la conexión entre las etapas de modelado e implementación a través del pasaje automático de la primera a la segunda, intentando emular el incremento de productividad logrado con la aparición de los 3GLs frente a los lenguajes de menor abstracción presentes en la época de su surgimiento.

Luego de estos aspectos introductorios, se mencionaron características más detalladas de la metodología: el establecimiento de un dominio acotado, la confección de un lenguaje de modelado de alto nivel de abstracción y sus características, la obtención directa de la aplicación final (y otros artefactos de software) y el rol del framework de dominio en la metodología.

En la sub-sección siguiente se comentaron aspectos positivos de la metodología. Entre ellos, los más relevantes son el rápido aprendizaje del lenguaje específico por parte de los desarrolladores gracias a su cercanía con el dominio y a la asistencia de las herramientas de modelado, mayor incidencia de los conocimientos de los expertos en el producto final, menor esfuerzo requerido para construir software con calidad de producción (más productividad) e independencia tecnológica de la aplicación o sistema especificado. También se relataron ventajas que la metodología provee en términos de la relación con clientes: más participación de los mismos en el desarrollo gracias al alto nivel de abstracción de los modelos que describen al software y la capacidad para desarrollar prototipos más rápidamente, implicando esto último una mejora consecuente en la validación de la aplicación o sistema en lo concerniente a requerimientos.

Se comentan también, durante el transcurso de las sub-secciones mencionadas, desventajas, características negativas o restricciones de aplicabilidad relativas a DSM. La necesidad de un equipo de expertos que asegure su correcta aplicación, el requerir una inversión inicial en la confección de las herramientas e infraestructura que la metodología requiere, y la imposibilidad de reuso directo en otras áreas debido a la cota impuesta en el dominio que considera el lenguaje son las más importantes.

Finalmente, se relató de manera breve la evolución de distintas metodologías basadas en modelos y su relación con DSM. Se resumieron también los preceptos principales que la metodología de Modelado Específico de Dominio promueve y requiere para su correcta implementación: uso de lenguajes de alto nivel de abstracción para especificar software, generación completa de código y el uso de herramientas que asistan a la construcción de modelos y faciliten la confección de los lenguajes específicos de dominio y de los entornos de modelado.

5 DEFINICIÓN DE RICH INTERNET APPLICATIONS A TRAVÉS DE MODELOS ESPECÍFICOS DE DOMINIO: MOTIVACIÓN Y ASPECTOS A CONSIDERAR

5.1 Introducción

En esta sección se mencionarán, por un lado, las principales motivaciones que subyacen como ventajas de especificar Rich Internet Applications a través de Modelos de Dominio Específico. Estas ventajas se adicionan a las ya provistas por la metodología de Modelado Específico de Dominio (implementada correctamente y con un dominio adecuadamente acotado).

A posteriori, se tratarán aquellos aspectos de las Rich Internet Applications deseables de ser especificados en un alto nivel de abstracción. Estos serán determinados a partir de las tecnologías orientadas a su construcción que han sido analizadas y considerando patrones de diseño usualmente presentes en su definición. También serán tenidos en cuenta metodologías de modelado orientadas a la definición de aplicaciones web tradicionales y se considerarán Aplicaciones Ricas implementadas y utilizadas intensivamente en la actualidad.

5.2 Motivación

5.2.1 Portabilidad tecnológica

Como ya se comentó, el concepto de Rich Internet Application engloba a cualquier aplicación web capaz de efectuar algún tipo de interacción *rica*, es decir, que pueda llevar a cabo una o más tareas en el client-side las cuales requerirían indefectiblemente de interacción cliente-servidor en una aplicación web tradicional. La amplitud de esta definición engloba a un gran número de tecnologías que han surgido en la actualidad para satisfacer este propósito.

Por un lado, existen hoy en día un gran número de librerías que pueden adosarse a aplicaciones web tradicionales a fin de enriquecer parte de su interfaz de usuario. Las más utilizadas son las confeccionadas en lenguaje JavaScript, y algunos ejemplos están dados por jQuery, DOJO[12] y MooTools[13]. Frameworks como ZK y Echo2 brindan un mayor nivel de abstracción en este campo, al encapsular detalles de bajo nivel relativos a comportamiento de la UI e interacción, a través un modelo de componentes y eventos a tal fin (teniendo esto un costo asociado que ya se ha mencionado). GWT permite una abstracción similar, aunque brinda mayor detalle a la hora de especificar la interacción cliente-servidor, teniendo que ser la misma explícitamente codificada. Finalmente, otros frameworks como OpenLaszlo, Microsoft Silverlight y Adobe Flex ofrecen lenguajes, *runtimes* y entornos propios para desarrollar RIAs.

Aquellas herramientas diseñadas para la especificación sólo del client-side como,

12 http://www.dojotoolkit.org/
13 http://mootools.net/

por ejemplo, OpenLaszlo o las librerías JavaScript mencionadas, implican que la tecnología utilizada para implementar el server-side deberá ser escogida independientemente de la relativa a la faceta cliente. Esto hace aún más amplio el espectro de herramientas sobre el cual puede optarse al llevar a cabo el desarrollo de Rich Internet Applications.

El disponer de un modelo de alto nivel capaz de capturar las características primordiales del desarrollo de RIAs bajo un dominio particular permite al equipo encargado de tal tarea abstraerse de las tecnologías subyacentes que implementarán las especificaciones confeccionadas. Serán los expertos en el área de implementación quienes tratarán los aspectos tecnológicos y quienes crearán, en base a su conocimiento en el campo, los derivadores que permitirán obtener las Rich Internet Applications finales con calidad de producción. Los modelos confeccionados serán artefactos de software que permanecerán intactos ante cambios en las tecnologías de implementación, lo cual permite escoger la herramienta más adecuada según los requerimientos de bajo nivel, sin que esta última condicione las especificaciones elaboradas. Esto permite incluso que pueda cambiarse completamente la plataforma de ejecución, sin necesidad de especificar nuevamente el software requerido. En un área de desarrollo con tecnologías heterogéneas, diversas y cambiantes como el de las RIAs, esta abstracción supone mayores ventajas que en áreas en donde la variedad tecnológica es menor.

5.2.2 Multimedia

En [30] se analizan las carencias de las metodologías de modelado de aplicaciones web actuales a la hora de especificar aspectos requeridos en la construcción de Rich Internet Applications. Si bien algunas de las mismas como, por ejemplo, WebML[11], han sido probadas exitosamente en la industria para construir aplicaciones web tradicionales, el artículo resume la incapacidad general de las metodologías de modelado web modernas para representar características concernientes a RIAs.

Las tecnologías utilizadas para implementar aspectos multimediales tales como, por ejemplo, streaming de video on-demand, pueden llegar a ser muy diferentes de las asociadas a transferencia de datos convencionales, tanto en lo relativo al cliente como en lo que concierne al servidor. Desde el punto de vista del client-side, tomando como ejemplo el caso particular de video bajo demanda, esto puede observarse en la necesidad de los browsers modernos de incluir componentes de terceros tales como Adobe Flash Player[14], Windows Media Player[15] o Microsoft Silverlight, que hagan posible la implementación de este tipo de funcionalidad. Nuevamente, se tiene diversidad tecnológica la cual es deseable abstraer al especificar Rich Internet Applications. Debido a esto, los componentes de un lenguaje de alto nivel que permitan modelar RIAs deberán entonces considerar contenidos multimediales tales como imágenes, videos y sonido, los cuales luego serán derivados a una implementación particular.

5.2.3 Abstracción en componentes interactivos

Las mejoras en interactividad factibles de ser alcanzadas por las RIAs han llevado al surgimiento de nuevos paradigmas de interacción y con ello, a nuevos *widgets* o

14 http://www.adobe.com/products/flashplayer/

15 www.microsoft.com/windows/windowsmedia/players.aspx

componentes de interfaz de usuario que ofrecen riqueza tanto en términos de interactividad como de presentación[16]. Aunque con características particulares, muchos de estos componentes están orientados a una misma tarea, lo cual los hace semánticamente equivalentes, a pesar de sus diferencias en términos de presentación e interacción. Un ejemplo clásico está dado al mostrar en la UI una colección de elementos de datos: la misma puede ser efectuada por una lista paginada convencional, por un *carousel*[29], un componente de lista desplegable, una tabla interactiva, entre otros. Si la colección de elementos permitiera elegir un subconjunto de los mismos con algún fin determinado, esta selección también es un aspecto que puede implementarse de distintas maneras. Un modo usual de hacerlo es proveyendo gráficamente dos listas: una que incluya a aquellos elementos disponibles para ser seleccionados y la otra que liste a aquellos que ya han sido elegidos. Transfiriendo elementos entre ambas listas el usuario puede establecer cuales conformarán la selección. Esta misma tarea puede ser efectuada sin mayores inconvenientes a través de un componente especial presente por cada elemento en la colección que permita denotarlo como seleccionado.

La abstracción de estos aspectos en conceptos de alto nivel resulta provechosa, dado que evita la redundancia semántica entre los componentes de interfaz de usuario que conformarán las interfaces ricas. Adicionalmente, aspectos detallados de presentación y comportamiento pueden ser especificados en el meta-modelo en etapas avanzadas del desarrollo, o bien pueden ser forzados en los generadores según se desee.

5.2.4 Balance de carga entre cliente y servidor

Como ya se ha mencionado, las Rich Internet Applications definen un punto intermedio entre las aplicaciones de escritorio, las cuales se ejecutan en su totalidad en el client-side, y las aplicaciones web tradicionales, cuya ejecución se lleva a cabo en la faceta servidor. El punto en el que una RIA se sitúa entre estos límites depende de cuánto se distribuya la ejecución en el client- y server-side, diferencia la cual ya ha sido apreciada entre las llamadas tecnologías RIA de *cliente pesado* como, por ejemplo, GWT, en contraste con aquellas de *cliente liviano* como ZK y Echo2, entre otras.

La posibilidad de definir el grado de dependencia del server-side de la Aplicación Rica es, por lo tanto, un factor de interés a tener en cuenta al especificar este tipo de aplicaciones, el cual no es considerado por las metodologías de modelado de aplicaciones web actuales. Por este motivo, han sido propuestas extensiones a estos lenguajes las cuales sean capaces de especificar la distribución de datos y procesamiento entre las facetas cliente y servidor como, por ejemplo, en [19].

5.2.5 Definición declarativa de interfaces de usuario en RIAs

Como se ha observado en el análisis llevado a cabo, varias de las tecnologías modernas para implementar RIAs incluyen como característica principal un lenguaje de dominio específico a través del cual puede llevarse a cabo la declaración de la interfaz de usuario rica y aspectos de interacción con el server-side: ZK con ZUML, OpenLaszlo con LZX, Microsoft Silverlight con XAML, Adobe Flex con MXML, entre otros. A pesar de las diferencias entre los distintos lenguajes, todos persiguen el objetivo de facilitar la

16 Un catálogo notable de patrones puede hallarse en `http://developer.yahoo.com/ypatterns/`

confección de interfaces de usuario ricas de una manera declarativa.

Esta característica común entre distintas tecnologías modernas orientadas al desarrollo de RIAs sugieren que no solo es factible, sino también deseable en términos de productividad, el contar con un lenguaje de dominio específico que facilite la definición de aspectos estáticos y dinámicos relativos a las Rich Internet Applications. La tendencia de la industria a adoptar lenguajes con estas características, en adición a la portabilidad tecnológica anteriormente mencionada facilitada por la metodología DSM, motivan el interés del desarrollo de un lenguaje que resuma los aspectos relativos al desarrollo de RIAs y que, a su vez, permita derivar implementaciones funcionales con calidad de producción.

5.3 Aspectos relevantes de RIAs a tener en cuenta

Como se ha observado, las tecnologías analizadas comparten ciertos aspectos en común los cuales caracterizan a las Aplicaciones Ricas. A su vez, la gran cantidad de este tipo de aplicaciones que han surgido en la actualidad, ha conllevado a la detección y posterior catalogación de patrones de diseño comúnmente aplicados en su construcción como, por ejemplo, los mencionados en [3].

En esta sección se describirán los principales aspectos concernientes al dominio de las Aplicaciones Ricas y, por lo tanto, aquellos elementos del mismo que son de interés representar en un meta-modelo que las caracterice. La definición de estos aspectos tendrá como base al análisis de patrones de diseño mencionados en la bibliografía (especialmente en [3] y [14]), adicionándose las características obtenidas en lo relativo a tecnologías orientadas a su desarrollo investigadas. Asimismo, los lenguajes de modelado ya existentes para la definición de aplicaciones web tradicionales serán considerados también como base para definir los componentes que a futuro formarán parte del meta-modelo.

5.3.1 Carga del client-side

En la actualidad, y a partir del surgimiento y consolidación de distintas variantes del paradigma cliente/servidor (siendo las RIAs un caso particular y el más recientemente adoptado), la arquitectura de las aplicaciones o sistemas comenzaron a apreciarse divididos en tres capas fundamentales: interfaz de usuario o presentación, lógica del dominio o de negocios y fuentes de datos [3][7]. Los clientes *pesados*, categoría de la cual usualmente forman parte las Aplicaciones Ricas, consideran no sólo aspectos de la capa de presentación como ocurre en los clientes *livianos*, sino también aspectos de la lógica del dominio o de negocios. La representación de estos aspectos por parte del meta-modelo se torna entonces fundamental, así como también la inclusión de aspectos ya presentes en las aplicaciones web tradicionales.

Ejemplos reales

Cualquier Aplicación de Internet que sea clasificada como Rica deberá contener una o más características de interfaz de usuario y comportamiento que permitan alguna interactividad de mayor riqueza que la provista por las aplicaciones web tradicionales. Estas características pueden estar presentes en pequeñas partes de la aplicación, o bien pueden llegar a extenderse por completo a toda la interfaz de usuario.

Algunos ejemplos actuales de este tipo de aplicaciones son GMail[17],Facebook[18], el portal Yahoo![19], entre muchas otras.

5.3.2 Interactividad y performance en comunicación

Como se dijo en la sección correspondiente, las RIAs representan una variante de la arquitectura cliente/servidor, en la cual se utiliza Internet como medio de comunicación entre client- y server-side. Los aspectos concernientes a la arquitectura misma, sumados a las limitaciones en cuanto a ancho de banda y latencia, deben tenerse en cuenta a fin de brindar una interfaz de usuario rica en interactividad. Las capacidades mejoradas en términos de comportamiento en el client-side, característica fundacional de las las RIAs, puede ser explotada para alcanzar este fin.

Caching, prefetching y multi-stage downloading

El *caching*, la duplicación de datos en el client-side a fin de evitar repetir peticiones al servidor para solicitar datos que ya fueron pedidos con anterioridad, es una técnica que se utiliza exitosamente para obtener mayor velocidad de respuesta y menor ancho de banda en los browsers desde su surgimiento. El *prefetching* o *Predictive Fetch*[3], la solicitud automática de datos o recursos extra que probablemente se necesiten a la brevedad, representa otra manera de reducir la latencia, aunque conduciendo a un mayor uso de ancho de banda. Finalmente, si los contenidos de interfaz de usuario son considerables, por ejemplo, debido a que contienen imágenes u otros elementos que conllevan un alto tiempo de transferencia, la carga de los mismos en su totalidad de una sola vez puede llegar a significar un tiempo de espera considerable para el usuario. Una mejora frente a esta situación es lo que se denomina *Multi-Stage Downloading* [3] o *Deferred Content Loading* [14]: la aplicación se descompone en partes las cuales son cargadas de a etapas, ya sea en paralelo o en algún orden serial, en orden de relevancia.

Cuando el flujo de datos se produce desde el cliente hacia el servidor, puede igualmente aplicarse técnicas para mejorar la performance. Si el envío consiste en pocos datos enviados con escasa diferencia temporal entre sí, pueden agruparse varios mensajes en uno solo, conformando una suerte de buffer de mensajes que es vaciado y enviado como un mensaje único. El envío puede efectuarse cada cierto período de tiempo o bien al acumularse una cantidad de información razonable a ser enviada. Esta técnica, definida en [3] como *Submission Throttling*, es utilizada usualmente en las sugerencias o auto-completación de datos en cuadros de texto, y reduce el overhead de comunicación y la cantidad de respuestas que serán descartadas debido a que ya no son necesarias. La capacidad para confeccionar interactivamente una lista de elementos la cual será dada de alta en una sola petición al servidor, conforma una aplicación similar de la lógica que el patrón describe. Para lograr la misma interactividad en una aplicación web tradicional, sería necesario el envío de una petición por cada elemento de datos que se quiera crear, o bien se requiere fijar la cantidad de datos que pueden ser enviados de una sola vez en la UI. Dada la capacidad de modificar la UI en tiempo real y sin involucrar la server-side, estas limitaciones pueden sobrellevarse en las RIAs.

17 http://mail.google.com
18 http://www.facebook.com
19 http://www.yahoo.com

Paginación

Cuando se dispone de un listado de elementos de magnitud considerable, es usual en las aplicaciones web el dividir el mismo en páginas, brindándole al usuario la capacidad de navegar entre las mismas con componentes de UI definidos a tal fin. Esto evita un tiempo de carga inicial excesivo y le permite al usuario buscar de manera más eficiente entre los datos, cuando estos son muy numerosos. En las RIAs, la capacidad para efectuar peticiones asincrónicas al server-side desde el cliente permite solicitar datos relativos a otras páginas sin necesidad de una recarga completa de la interfaz de usuario, situación la cual es inevitable en aplicaciones web tradicionales. Los nuevos datos obtenidos son plasmados en la UI al ser recibidos, explotando la posibilidad de modificación de la misma en el client-side. Esto se traduce en un menor ancho de banda utilizado para la navegación entre páginas, dado que sólo se transfieren los datos, no la UI, lo cual tiene como consecuencia un menor tiempo de respuesta.

Una variante de este esquema de paginación es lo que se denomina *infinite scrolling*: la carga inicial de una primer fracción de la lista, requiriéndose sus elementos siguientes de ser necesarios a medida que el usuario se desplaza sobre la misma. Un concepto más general de este comportamiento es capturado en el pattern *Virtual Workspace* [3], el cual versa acerca de un área de trabajo virtual cuyo contenido completo se encuentra en el server-side, siendo el mismo cargado de a fracciones en el client-side a medida que se va necesitando. De esta manera, la posibilidad de carga bajo demanda ejemplificada a través de una lista, puede aplicarse a cualquier estructura de información la cual pueda ser fraccionada en partes y entregada a medida que se necesite.

Conclusión

Todas estos detalles relativos a interactividad y comunicación que han sido mencionados, son frecuentemente utilizados en RIAs y deben ser implementados desde cero o bien con ayuda de algún framework o librería (detalles de implementación bajo DHTML/ Ajax pueden hallarse en [3]). Su importancia no guarda relación con otros aspectos más que con la eficiencia en cuanto a uso de ancho de banda y reducción de la espera por latencia, por lo que el meta-modelo no considerará su inclusión si las optimizaciones puedan hacerse automáticamente sin costo extra. En caso de que el desarrollador pueda elegir si utilizarla o no con algún costo o consecuencia adicional, se buscará que el lenguaje le brinde la posibilidad de hacerlo en los componentes que las soporten.

Aplicación en casos reales

La aplicación interactiva de mapas Google Maps[20] contiene fotografías satelitales y mapas de todo el mundo, pudiéndose con una simple búsqueda ubicarse en un lugar particular del planeta y, mediante una interfaz *drag and drop*, moverse desde el punto inicial hacia cualquier otro sitio (Figura 5). Debido a la gran magnitud de información y su consecuente tiempo de descarga desde el servidor, las imágenes que conforman los mapas o fotos satelitales son divididas en fracciones y son solicitadas por la interfaz rica a medida que son necesitadas, así como también sus aledañas, dado que es muy probable que sean requeridas a la brevedad. Esta situación es un claro ejemplo de *prefetching*, así como

20 http://maps.google.com

también de *caching*, dado que las imágenes navegadas se mantienen almacenadas momentáneamente por si el usuario vuelve a sitios que ya exploró. Asimismo, esta aplicación representa un ejemplo clásico del pattern *Visual Workspace*.

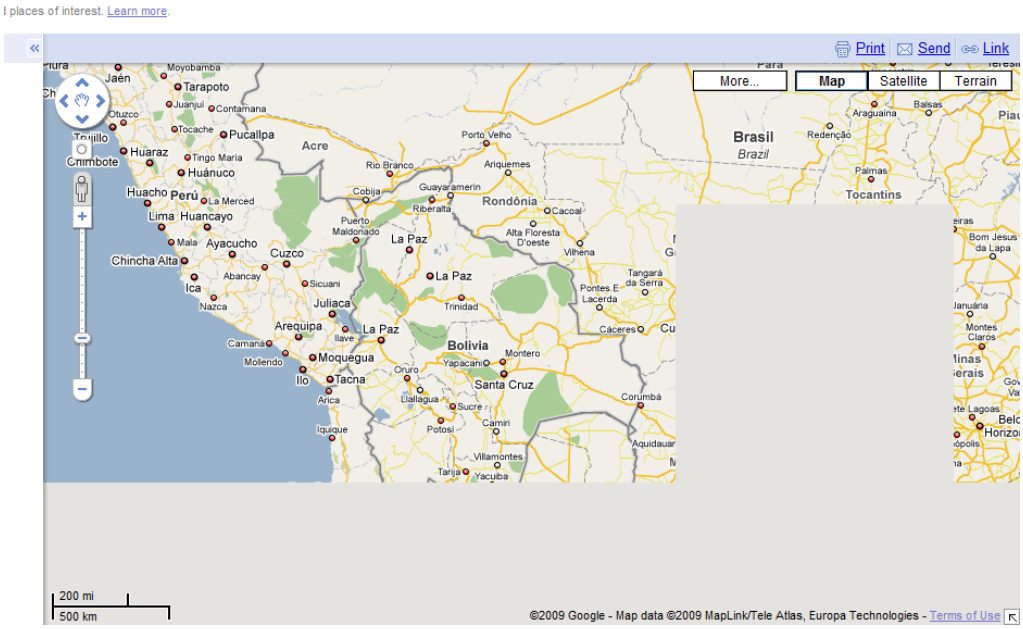


Figura 5. Visualización de un mapa en Google Maps. Se puede apreciar la carga bajo demanda de fracciones de contenido de la UI y la aplicación del pattern *Visual Workspace*.

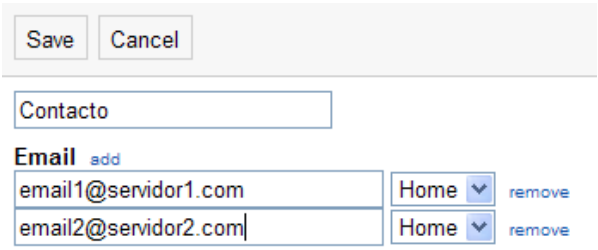


Figura 6. Edición de contactos en GMail: se permite agregar o quitar direcciones de correo para un contacto dinámicamente, enviándose los nuevos datos al seleccionar *Save*.

En GMail es usual en distintas ocasiones el agregar o quitar conjuntos de componentes que representan instancias de datos determinadas a ser creadas o modificadas, ejecutándose el alta o modificación de modo diferido cuando el usuario lo desea. En particular, la edición de datos de contactos, como se observa con las direcciones de email en la Figura 6, es un ejemplo de cómo puede reducirse la cantidad de peticiones efectuadas al servidor a una sola. Esto se logra permitiendo especificar múltiples instancias de datos replicando los componentes de UI que permiten construirlas y efectuando una sola invocación al servidor para efectuar las múltiples altas. El resultado de esta técnica es una menor carga para el server-side, así como también una mejora en la interactividad en el client-side, dada la no necesidad de espera por que un alta se complete a fin de poder efectuar la siguiente. En una aplicación web tradicional, o bien debería forzarse a un número fijo de instancias que pueden crearse a la vez, prefijando la cantidad de componentes de construcción en la interfaz de usuario, o bien debería efectuarse una petición por cada

instancia de datos agregado. Este patrón no ha sido descripto aún por la bibliografía analizada, por lo que en este documento se lo catalogará con el nombre *Merged Uploads*, y su fundamento de base es el mismo que el pattern *Submission Throttling*[3], aunque aplicado a mensajes enviados al servidor los cuales no tienen restricciones temporales acotadas.

El concepto de paginación y scrolling es tan frecuente en aplicaciones web centradas en datos (ya sean ricas o no), que la metodología WebML[11] la considera como un elemento de primer orden en su meta-modelo.

5.3.3 Definición de la interfaz de usuario

Como se comprobó al analizar las distintas tecnologías para implementar RIAs, la confección de la interfaz de usuario de una manera declarativa, por ejemplo, a través de algún tipo de descripción jerárquica como un archivo XML, resulta más natural frente a la alternativa programática. Este tipo de descripciones se ajustan naturalmente a la estructura jerárquica de inclusión de componentes, pudiendo realizarse en tiempo de diseño las validaciones correspondientes para evitar errores semánticos al definir la estructura de la interfaz de usuario. Un lenguaje gráfico no sólo puede llevar a cabo estas validaciones, sino que puede directamente no hacer posible la construcción de UI inválidas en el entorno o herramienta que se utilice, guiando al desarrollador en su correcta confección. Por otra parte, los desarrolladores no necesitan definir programáticamente la inclusión de componentes en otros componentes compuestos, dado que la misma es expresada naturalmente a través de la estructura jerárquica de la especificación. De esta manera, errores como el instanciar componentes y olvidar el agregarlos a su componente contenedor son directamente evitados.

Finalmente, las definiciones declarativas de UI son fácilmente representables en meta-modelos debido a su naturaleza estructural, y en la actualidad cada una de las tecnologías emblemáticas para desarrollo de Aplicaciones Ricas cuenta con su propio lenguaje específico para declarar UIs [6], lo cual fortalece su viabilidad y conveniencia. Las RIAs implementadas bajo DHTML/Ajax corren con un problema adicional: a pesar de la existencia de estándares como DOM, en la práctica se dan diferencias de implementación entre distintos navegadores. A su vez, dado que esta plataforma para correr Aplicaciones Ricas está presente en cualquier browser moderno convencional, existe un gran interés en desarrollar software sobre la misma. Tanto es así, que el hecho de proveer funciones y componentes portables entre browsers es considerado en [3] como un patrón particular, denominado *Cross-Browser Component*. En este contexto y para solucionar estos inconvenientes, surgieron en la actualidad una gran cantidad de librerías y frameworks (por ejemplo, Dojo, Prototype, jQuery, entre otros) que, entre otras cosas, abstraen los susodichos problemas de compatibilidad, facilitando la tarea de los desarrolladores. Frente a esta situación, poseer descripciones de alto nivel para especificar la UI de RIAs permite evitar las problemáticas introducidas por aspectos de implementación como los mencionados.

Como fue observado en el framework OpenLaszlo, la posibilidad de seleccionar declarativamente la fuente de datos que los controles de interfaz de usuario utilizarán evita la necesidad de escribir código que realice esta asociación manualmente. Además, esto

enriquece la semántica de los datos que los componentes referencian, dado que asocia los mismos con su estructura y permite realizar controles de consistencia. Finalmente, las dependencias de datos entre componentes de UI se dan naturalmente y las actualizaciones correspondientes pueden inferirse y llevarse a cabo sin necesidad de programarse manualmente a través de mecanismo de observación.

5.3.4 Arquitectura Orientada a Servicios y Web Services

La Arquitectura Orientada a Servicios (Service Oriented Architecture o SOA) define un sistema como un conjunto de componentes denominados *servicios*, los cuales interactúan entre sí. Cada uno de estos *servicios* debe especificarse siguiendo un estándar acordado y es capaz de llevar a cabo cierta función para la que es definido, encapsulando dentro de sí toda la lógica necesaria para hacer efectiva su ejecución. Cada *servicio* es independiente del estado o contexto particular del sistema y su interoperabilidad se logra utilizando protocolos estandarizados y preestablecidos de comunicación. Para preservar el encapsulamiento, un *servicio* ofrece una *descripción*, la cual indica cómo el mismo debe ser invocado y el resultado que se obtendrá como consecuencia de dicha invocación. Adicionalmente, los servicios se comunican entre sí mediante el envío de *mensajes.* [4][15]

El objetivo de la arquitectura es estructurar un sistema a través de *servicios* con bajo grado de acoplamiento, los cuales interactúen en conjunto para llevar a cabo cada uno de los aspectos que implica la lógica de negocios del dominio. Los aspectos principales que la arquitectura promueve y que representan las principales ventajas de su adopción, son[4]:

- Bajo acoplamiento. Se minimizan las dependencias entre componentes -*servicios*-, limitándose a que cada uno conozca la existencia e interfaz de comunicación con los otros.
- Definición de Contratos entre servicios, estableciendo un protocolo de comunicaciones común y definiendo una o más *descripciones*.
- Autonomía y encapsulamiento. Cada *servicio* mantiene control sobre la lógica que lleva a cabo y su ejecución es independiente del contexto global del sistema.
- Abstracción. Cada *servicio* oculta su lógica interna y sólo muestra su interfaz.
- Reusabilidad. La lógica del sistema es dividida entre *servicios*, los cuales pueden ser reusados.
- Composicionalidad. Los *servicios* pueden ser coordinados y ensamblados a fin de componer otros nuevos *servicios*.
- Capacidad de descubrimiento. Se proveen mecanismos para que los *servicios* puedan ser detectados y se determine la manera en la que pueden ser invocados.

La opción mayormente adoptada en la actualidad para definir arquitecturas orientadas a servicios son los denominados *Web Services*, los cuales son definidos como *un sistema de software diseñado para soportar interacción máquina a máquina sobre una red*[16]. Los *Web Services* se caracterizan por la especificación de distintos estándares

utilizados para definir la arquitectura SOA: el estándar SOAP para intercambio de mensajes, descripciones de servicios basados en WSDL y métodos de registración y descubrimiento de servicios a través de UDDI.

En lo relativo a las Aplicaciones Ricas, la principal ventaja que ofrecen los *Web Services* es el evitar la necesidad de comunicación entre las facetas cliente y servidor a través de documentos HTML tradicionales, o protocolos propietarios o creados a medida, haciendo uso del conjunto de estándares ya mencionado en su lugar. Adicionalmente, proveen un marco sólido en cuestiones de validación estructural de la información transferida desde y hacia el server-side. Siguiendo las características fundacionales de SOA, los *Web Services* pueden estar concebidos para ser utilizados por otras entidades distintas a aquella que los definió inicialmente, y existen gran cantidad de sitios remarcables (por ejemplo, `amazon.com`) que los ofrecen para facilitar a los desarrolladores el acceso a distintas funcionalidades e información que los mismos utilizan internamente. Actualmente, en términos de implementación, se han propuesto varias maneras de invocar servicios para cada tecnología y plataforma RIA particular [3][4][5].

Los *Web Services* son usualmente utilizados y diseñados de acuerdo a una arquitectura de implementación y definición de capa se servicios denominada RPC o Remote Procedure Call. Bajo la misma, cada *servicio* representa una acción que puede ser invocada, cuya semántica depende exclusivamente de cómo se defina su funcionamiento y tareas que desempeña internamente. Dado este encapsulamiento, el invocador conoce sólo la interfaz del *servicio* que está invocando, es decir, cómo el mismo debe ser llamado y la respuesta que espera obtenerse al finalizar la invocación. En la actualidad, el interés en una arquitectura diferente a RPC denominada REST (Representational State Transfer)[3][13] ha crecido considerablemente, abogando mayor simplicidad e intentando emular aquellas características de escalabilidad que le han permitido a la Web crecer como lo ha hecho. A diferencia de la arquitectura RPC, REST busca alcanzar un modo uniforme para invocar servicios, de manera que los clientes de los servicios conozcan más acerca de la semántica de los mismos de manera implícita. La características principales de la arquitectura REST serán comentadas en la siguiente sección.

Servicios RESTful

En la arquitectura REST, los servicios son definidos como *recursos*, los cuales son descriptos como *cualquier aspecto que es suficientemente importante como para ser referenciado en la capa de servicios, el cual debe poder ser representado como una cadena de bits*[13]. Sobre los mismos es posible efectuar operaciones tales como altas, bajas, modificaciones, eliminaciones, búsqueda, etc. La primer característica de importancia de los recursos es que son *direccionables*, es decir, que se pueden referenciar unívocamente a través de una dirección.

Si bien la arquitectura admite el uso de cualquier protocolo de comunicación y método de direccionamiento de recursos, usualmente se asume al hablar de REST que el protocolo de comunicación es HTTP, y el método de direccionamiento utilizado para referenciar recursos son URLs. Más allá del protocolo de comunicación particular que se utilice, los recursos son transferidos desde y hacia el servidor a través de *representaciones*, las cuales se definen como una secuencia de bytes estructurados según un formato particular

que describen el estado de un recurso[13].

A diferencia de RPC, en donde los objetos centrales de la arquitectura son *operaciones* (verbos), la arquitectura REST tiene como elemento central los datos, abstraídos en *recursos* (sustantivos). Por lo tanto, se hace necesario determinar cuál es la acción particular que se desea llevar a cabo sobre el recurso referenciado al efectuar una petición al servidor. A este fin, se utilizan los métodos ya definidos en el mismo protocolo HTTP para comunicación, respetando su respectiva semántica:

- El método GET permite obtener una representación del recurso referenciado por la URL en la petición.
- El método DELETE elimina al recurso referenciado en la URL incluida en la petición.
- El método POST provoca la creación de un nuevo recurso, retornando su dirección, debiendo adjuntarse en la petición la representación que describa los datos de la nueva instancia a crear.
- El método PUT permite modificar un recurso existente referenciado por la URL en la petición, debiéndose adjuntar la representación que describa los datos actualizados de la instancia a modificar.

En la arquitectura REST, el servidor no contiene un estado particular de ejecución. Es decir, el cliente no necesita forzar a que el servidor entre en un estado particular para poder llevar a cabo la ejecución de un servicio adecuadamente. Esto implica, por ejemplo, que el solicitar el contenido de un recurso retornará el mismo resultado esperado, más allá de las peticiones que se hayan producido antes la misma.

De la misma manera, las operaciones GET, DELETE y PUT son idempotentes, es decir, la ejecución de dos o más operaciones idénticas no tiene efecto alguno luego de la ejecución de la primera. En redes no confiables como Internet, esta propiedad cobra mayor importancia, dado que una misma operación puede ser ejecutada múltiples veces de no recibirse en tiempo y forma adecuados la respuesta desde el servidor. A su vez, la operación GET es segura, en el sentido de que no causa modificación alguna en recursos. Esto representa una clara diferencia con otras implementaciones de SOA que utilizan el protocolo HTTP, las cuales usualmente utilizan el método GET para efectuar acciones varias además de consultas.

Al igual que las páginas HTML, los recursos conforman hipermedia, es decir, no incluyen sólo información, sino también referencias a otros recursos. La propiedad de direccionamiento antes mencionada hace posible esto sin mayores dificultades, y la propiedad de ausencia de estado asegura que el recurso referenciado represente siempre el mismo concepto y no cambie con el tiempo. De esta manera, se hace posible *navegar* de una manera natural entre los recursos.

Finalmente, la arquitectura REST persigue un diseño similar al de la Web, explotando las características que han atribuido a la misma de gran escalabilidad y robustez. La propiedad de ausencia de estado en el server-side permite que el balance de carga entre dos o más servidores puede implementarse de manera transparente, dada la no necesidad de almacenar o compartir el estado de ejecución entre los mismos. El uso de distintas técnicas de manejo y optimización de tráfico de datos como cachés, forwarding, proxys, entre otros,

aplicados exitosamente en la Web, pueden aplicarse igualmente a arquitecturas de servicios RESTful.

Como contrapartida, el hecho de que los servicios RESTful estén orientados a datos en lugar de a operaciones, tiene como efecto colateral la imposibilidad de implementar en la capa de servicios parte de la lógica de negocios. No obstante, las características ya mencionadas de las Aplicaciones Ricas en términos de capacidad de ejecución de operaciones complejas en el client-side contrarresta esta deficiencia, trasladando la lógica de negocios que no puede o no se desea ejecutar en el servidor hacia el cliente.

Conclusión

En esta sección se comentó brevemente el concepto de SOA, los *Web Services*, su implementación más utilizada, y sus variantes arquitectónicas principales: RPC y REST. En los frameworks RIA de *cliente liviano* analizados en este documento (ZK y Echo2), dado que uno de sus principales objetivos es el de ahorrarle al desarrollador la especificación manual de la comunicación entre el cliente y servidor, no se presenta la necesidad de construir una capa de servicios debido a que la misma ya es provista por las tecnologías en sí. No obstante, en las tecnologías RIA de *cliente pesado*, su diseño e implementación continúa siendo fundamental.

5.3.5 Eventos distribuidos

Si bien en la mayoría de los casos, siguiendo el patrón de las aplicaciones web tradicionales, la comunicación entre las facetas cliente y servidor en las RIAs es llevada a cabo desde el client- hacia el server-side, en ocasiones puede ser deseable que sea la faceta servidor quien inicie una comunicación con su contraparte para informarle acerca de la ocurrencia de algún evento que pueda ser de su interés. Un ejemplo de esta situación es advertir al client-side de cambios que se han producidos en la faceta servidor en lo relativo a información que actualmente ha almacenado y que puede estar manipulando localmente.

El llevar a cabo este tipo de funcionalidad implica la implementación de eventos distribuidos (patrón *Distributed Events*[3]), esquema de comunicación cuyo comportamiento es similar al conocido patrón de diseño *Observer*[8], pero en el contexto de una arquitectura cliente-servidor: la aplicación cliente se registra (ya sea explícita o implícitamente) a un aspecto particular del servidor frente al cual requiere ser notificada y, al darse un cambio en dicho aspecto, el servidor informa al cliente correspondientemente.

Una forma virtual de llevar a cabo esta notificación es el *polling* o *Periodic Refresh*[3], en el cual el cliente efectúa periódicamente solicitudes al servidor, las cuales serán respondidas indicándole si ha ocurrido o no uno o más eventos en los que está interesado. La segunda opción, conocida como *server push*, es que sea el servidor quien efectúe una petición al cliente frente a la ocurrencia de este tipo de eventos. Tecnológicamente, dada la naturaleza de *cliente liviano* presente en las aplicaciones web tradicionales, implementar esta última técnica bajo la plataforma DHTML/Ajax representa una problemática, siendo actualmente llevado esto a cabo a través de lo que se conoce como *HTTP Streaming*[3]. En el método de *HTTP Streaming*, el cliente establece una conexión permanente con el servidor, sobre la cual serán enviadas las notificaciones. La principal desventaja de esta técnica es la degradación de performance en el server-side causada por el

establecimiento de una conexión HTTP permanente con cada uno de los clientes ricos *online*. Su implementación requiere, en consecuencia, de software server-side diseñado específicamente para permitir este tipo de conexiones permanentes, como Glassfish[21], Pushlets[22], entre otros.

Finalmente, resta comentar que en las RIAs no es posible la comunicación cliente-cliente, entre otras razones, debido a que las restricciones de seguridad usualmente impuestas por el *runtime* sobre el que se ejecutan permiten sólo la comunicación desde el client-side hacia el servidor desde el cual fueron descargadas. Sin embargo, la distribución de eventos entre distintos clientes puede ser implementada utilizando al server-side como intermediario o *broker*.[33]

5.3.6 Interacción

Muchas de las capacidades que brindan las Aplicaciones Ricas se basan en la posibilidad de crear, destruir, mostrar u ocultar elementos de la interfaz de usuario sin intervención alguna del servidor. Estas capacidades pueden ser explotadas para llevar a cabo gran cantidad de tareas de distinta índole como, por ejemplo, mostrar mensajes de error en tiempo real al validar contenido ingresado por el usuario antes de que el mismo se envíe al servidor. A continuación se mencionan ciertos patrones de interacción y de diseño comunes los cuales implican modificaciones de la interfaz de usuario, de manera que puedan ser abstraídos luego incluidos bajo conceptos del meta-modelo a definir.

Multi-page vs. *single-page web applications*

Existiendo la ya mencionada posibilidad de modificar en tiempo real la estructura de la UI, pueden plantearse en general en las Aplicaciones Ricas dos maneras diferentes de mostrar nuevos contenidos, cuando estos sean solicitados. Siguiendo la manera tradicional de las aplicaciones web convencionales, el primer modo de efectuar esta tarea es navegar hacia una segunda página que conforme el contenido a mostrar, lo cual induce a una carga completa de la misma y a abandonar la UI actual. Este esquema relaciona fuertemente la interactividad con la navegación y, al estar la primera basada en el flujo entre diferentes nodos navegacionales o páginas, se denomina a las aplicaciones web diseñadas bajo este esquema *Multi-page web applications*[17]. Los meta-modelos planteados por las metodologías mayormente conocidas para modelar aplicaciones web ofrecen conceptos particulares de su lenguaje para representar páginas (nodos navegacionales) y la navegación entre las mismas [11][10][9].

Explotándose la capacidad de modificar la UI desde el client-side, el mismo contenido que en las aplicaciones web tradicionales es distribuido en páginas y links entre las mismas, puede mostrarse aplicando transformaciones a la interfaz de usuario actual, sin necesidad de efectuar la carga de una nueva página ni de abandonar la interfaz de usuario sobre la cual se está trabajando. Esta técnica permite interacción, sin que la misma implique obligatoriamente una acción de navegación y, por esta razón, las aplicaciones cuya interfaz de usuario respeta este diseño son denominadas *single-page web applications*. Es necesario aclarar que las aplicaciones web de tipo *single-page* no necesariamente están definidas en

21 http://glashfish.dev.java.net
22 http://www.pushlets.com

una sola página o nodo navegacional. Más específicamente, son caracterizadas por la capacidad implementar en una sola página, aspectos de interfaz de usuario que en las aplicaciones web convencionales conllevarían dos o más.

Las Aplicaciones Ricas actuales explotan al máximo la posibilidad de proveer interfaces de tipo *single-page*, siempre que sea posible y mejore la capacidad de interactuar con la UI. La introducción de este tipo de diseño mejora la interacción, al no requerir una recarga completa de la UI por cada petición efectuada al servidor y gracias a la posibilidad de reaccionar asincrónicamente frente a respuestas provenientes del mismo. GMail es un ejemplo claro de la ventaja obtenida en términos de interacción haciendo visibles UIs en modo *popup* al etiquetar mensajes de correo. Al realizarse este etiquetado, el usuario puede optar en una lista por una etiqueta ya existente, o bien por una última opción denominada *New label...*, la cual abre un cuadro de diálogo generado dentro de la misma UI, siguiendo el paradigma *single-page* (Figura 7), en el cual el usuario puede ingresar la nueva etiqueta que quiere asignarle a los correos seleccionados. De no existir esta capacidad, el usuario debería cargar otra UI completa a fin de ingresar el nombre de la etiqueta a generar, para luego retornar a la página de la que provino.

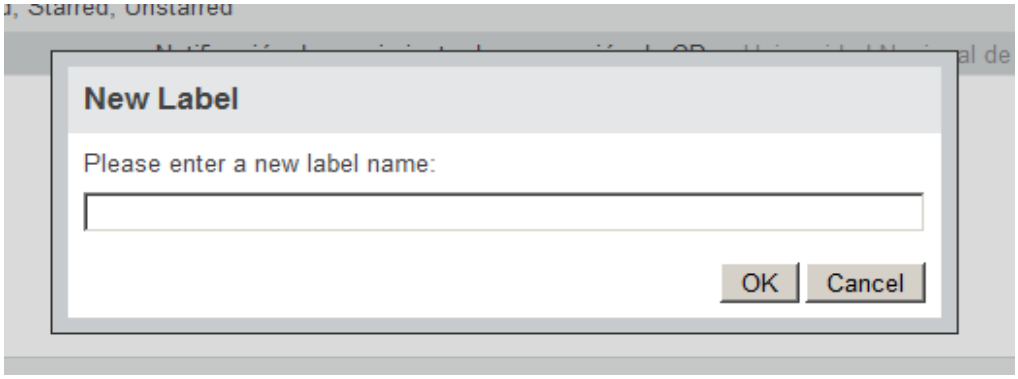


Figura 7. Etiquetado de múltiples correos en GMail, al seleccionarse la opción *New label...*

Detalles

Cuando se está observando cierta información en una aplicación web, dado que el límite en el espacio visible por el usuario es acotado, usualmente se hace necesario mostrar primero los datos más relevantes en relación al objeto que se está observando, ofreciendo luego la posibilidad de solicitar más información si así se desease. Asimismo, si el usuario deseara efectuar alguna acción con el contenido del objeto en cuestión, los elementos visuales que disparan dichas acciones no necesariamente tienen que estar visibles todo el tiempo, sino que pueden mostrarse sólo cuando el visitante decida ejecutar alguna tarea con respecto a los datos que observa.

Si la cantidad de datos extra a presentar es razonable, puede plantearse la solución de mostrarse en una nueva página dedicada todos los datos del objeto que se encuentra bajo observación. Por otro lado, si dicha cantidad fuera razonablemente pequeña, los mismos podrían mostrarse en la UI ya construida y ofrecerse al usuario algún método para hacerla visible u ocultarla según su preferencia. Bajo este esquema, el único contenido necesario a transferir desde el servidor hacia el cliente es la información adicional a mostrar, por lo que

pueden aplicarse las optimizaciones ya mencionadas en lo relativo a tráfico con el servidor. En particular, los datos pueden precaptarse, explotando la técnica de prefetching, y también pueden mantenerse en memoria si se necesitaran rever a futuro, efectuándose caching de los mismos. Por supuesto, esta forma de interacción requiere que la UI pueda modificarse en tiempo real y desde el client-side.

La capacidad para brindar más detalles acerca de la información que se está observando está presente en gran cantidad de Aplicaciones Ricas actuales. En particular, YouTube[23], el reconocido sitio de Video On-Demand, utiliza este tipo de mecanismo para brindar información más detallada acerca de los autores de los videos (Figura 8), videos relacionados, entre otros. A su vez, esta funcionalidad ha sido abstraída como patrones particulares de RIAs, siendo catalogada bajo el nombre de *Information Summary* en [12], y definiéndose como *Microlink* en [3].

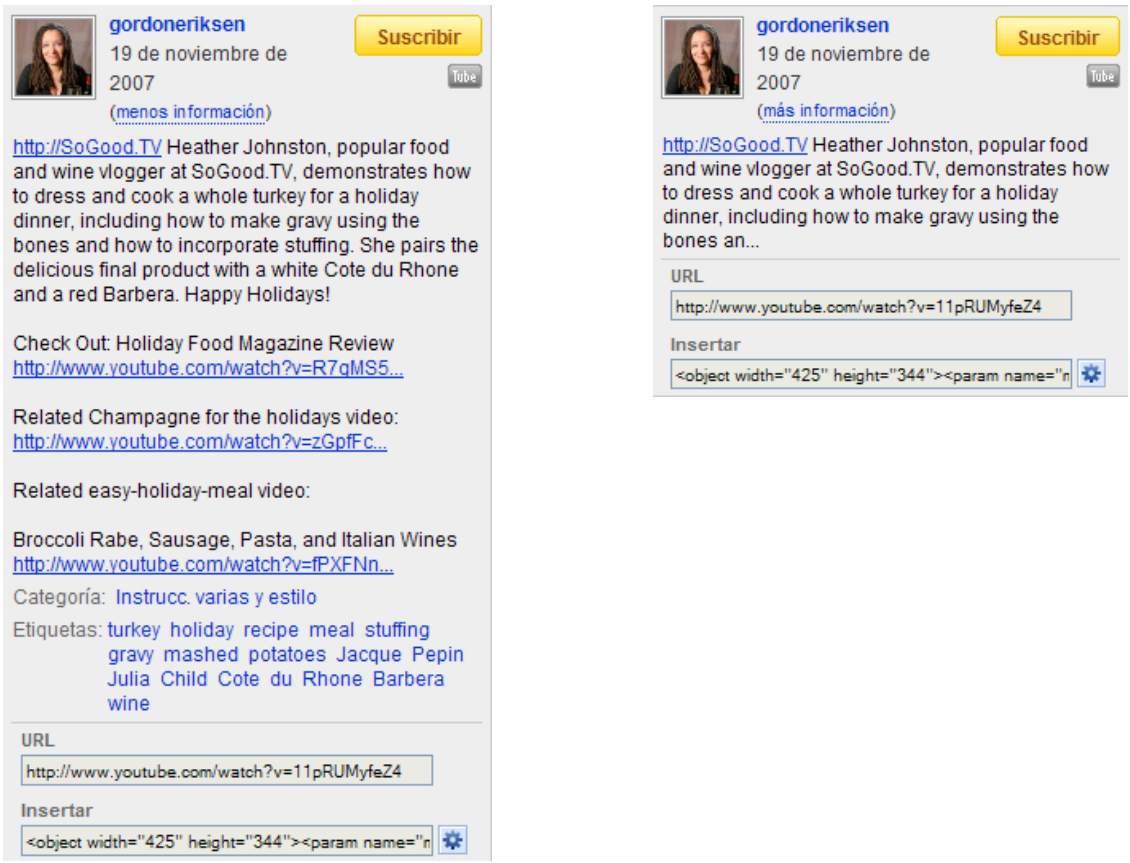


Figura 8: Ejemplo proveniente de YouTube, en el cual se utiliza un mecanismo para mostrar información detallada acerca del objeto de datos que se está observando.

Como se dijo anteriormente, esta técnica no sólo está ligada a mostrar información detallada del objeto de datos mostrado en la UI, sino también a hacer visible componentes de UI que faciliten el ejecutar acciones sobre el mismo (por ejemplo, realizar una modificación, etiquetarlo, etc.). Un uso alternativo de este mismo patrón, idéntico desde el

23 http://www.youtube.com

punto de vista funcional aunque distinto semánticamente, puede ser el mostrar u ocultar componentes de UI dinámicamente, sin que los mismos guarden relación con la información que se observa.

Tooltips

Es usual que en las interfaces de usuario de aplicaciones desktop se muestren *tooltips* (información asociada a componentes luego de señalar el mismo por algunos segundos) a modo de ayuda contextual. Esta funcionalidad puede extenderse a fin de mostrar uno o más subcomponentes los cuales permitan no sólo proveer una breve información adicional, sino una sub-interfaz de usuario completamente funcional. El pattern *Hover Detail*[14] captura precisamente este tipo de comportamiento.

Un ejemplo real de la aplicación de este patrón puede observarse en la interfaz de chat de GMail. En la misma, dejando el cursor estático unos segundos sobre un contacto particular, se muestra información más detallada del mismo, así como también acceso a funcionalidades y acciones disponibles a ejecutar.

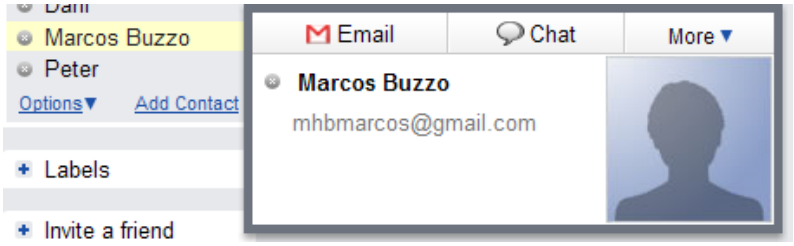


Figura 9. Información detallada de contactos mostrada a modo de *tooltip* en la UI de chat de GMail.

Lo anteriormente descripto puede ser aplicado para modelar e implementar distintos elementos de UI como, por ejemplo, menúes, aunque es deseable que el meta-modelo incluya conceptos, reglas y relaciones específicas para esta clase de situaciones particulares.

Live search

En las aplicaciones web tradicionales, como ya se mencionó con anterioridad, toda interacción desde el cliente hacia el servidor se lleva acabo enviando datos estructurados a través de formularios o bien a través de navegación. La búsqueda, una tarea frecuente en este tipo de aplicaciones, debe efectuarse indefectiblemente a través del envío de la(s) condición(es) de búsqueda al servidor, obteniéndose como respuesta una segunda página que contenga los resultados.

Sin embargo, en el contexto de las RIAs, las cuales tienen mayor posibilidad de captar eventos de UI, efectuar peticiones al servidor y modificar la estructura de la interfaz de usuario dinámicamente, las capacidades para retornar los resultados de las búsquedas se ven mejoradas. En particular, siguiendo lo descripto en el pattern *Live Search*[3] y en *Refining Search*[14], la búsqueda puede efectuarse en tiempo real a medida que el usuario define los parámetros que condicionarán los resultados. A bajo nivel, a medida que las restricciones son modificadas, nuevas peticiones de búsqueda que incluyen los parámetros

ingresados en ese momento particular son efectuadas al servidor. Cada respuesta recibida por parte del servidor provocará una actualización de la UI, la cual mostrará los nuevos resultados en tiempo real. Adicionalmente, utilizando la técnica de *Submission Throttling*, como ya se mencionó, se reduce el tráfico innecesario y una alta cantidad de peticiones que potencialmente pueden llegar a ser descartadas.

Suggestion

Con el mismo fundamento que el patrón *Live Search* mencionado en la sub-sección anterior, la misma técnica de interacción puede ser utilizada a fin de asistir al usuario que está ingresando texto en un determinado campo con opciones comúnmente utilizadas. La idea de esta funcionalidad es el facilitar al usuario el ingresar valores usuales que potencialmente pueda estar deseando utilizar, o bien asistir en el ingreso de un determinado valor dentro de un conjunto de valores válidos posibles. Este patrón es catalogado en [3] como *Suggestion* y en [14] como *Auto Complete* o *Live Search*, estando este último asociado a sugerencias en motores de búsqueda. Un ejemplo de aplicación del mismo se describe en la Figura 10.

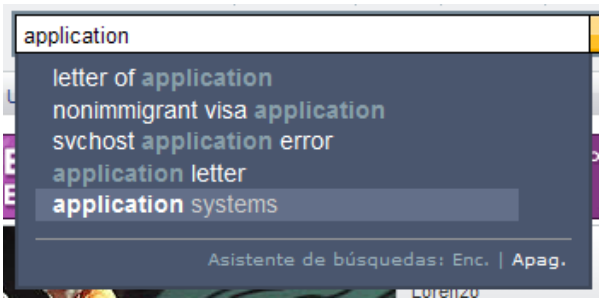


Figura 10. Sugerencia de opciones de búsqueda de acuerdo a los términos más buscados provista por el buscador Yahoo!

Selección

La posibilidad de seleccionar objetos de una lista se hace presente en aplicaciones web tradicionales a través del uso de *checkboxes* o *radio buttons* en formularios. Sin embargo, en las Aplicaciones Ricas, la capacidad de selección de uno o más objetos de una lista no se remite en términos de UI solamente al uso de campos de formularios, sino que cualquier lista o colección de elementos presentes en la interfaz de usuario puede ser objeto de selección. Esta capacidad se hace factible gracias a la posibilidad de detección y manejo de eventos de UI en el client-side y a la aptitud de modificar la estructura de la interfaz de usuario en tiempo real. El pattern *Selection* [14] abstrae precisamente este comportamiento.

El objetivo final de la selección de un subconjunto de objetos de una colección es usualmente el de aplicar cierta acción común sobre los mismos, la cual será disparada por un elemento o evento de interfaz de usuario particular. Una alternativa a la utilización de este patrón es el adjuntar, por cada elemento de la colección, un conjunto de componentes de UI que permitan ejecutar una acción particular sobre su elemento asociado, patrón de diseño el cual es definido como *In-Context Tool* en [14]. Este último esquema es aplicable

también a aplicaciones web tradicionales y, por ende, es igualmente posible de implementar con interfaces ricas. El método corre con la principal desventaja de requerir una nueva petición al servidor por cada acción que se ejecute sobre un elemento particular, si no se cuenta con las capacidades de interacción provistas por las RIAs. Contando con una UI rica al implementar *In-Context Tools*, es posible mantener en una cola las acciones particulares que se desean ejecutar y llevarlas a cabo en un solo request. Este comportamiento puede verse como una variación del pattern *Submission Throttling*[3] anteriormente mencionado, en el cual se agrupan acciones a ejecutar en lugar de caracteres, las cuales serán agrupadas y enviadas en una sola petición. Adicionalmente, la interfaz de usuario, tratándose de una interfaz rica, no necesita recargarse en su totalidad por cada acción o conjunto de acciones ejecutadas, sino que basta con que se hagan las transformaciones pertinentes en su estructura a fin de mostrar el nuevo estado de la lista.

Figura 11. Selección de correos en GMail sobre los cuales se efectuará una acción determinada (por ejemplo, archivamiento o borrado). En la misma imagen, puede apreciarse un ejemplo del patrón *In-Context Tool*: la posibilidad de marcar correos como *starred*, a través del *widget* provisto en cada ítem de la lista.

Drag and drop

La funcionalidad de *drag and drop* consiste en la posibilidad de reacomodar elementos de interfaz de usuario o ejecutar acciones asociadas a los datos representados por los mismos haciendo utilización del cursor, sin necesidad de ningún otro mecanismo o componente de UI adicional. En las aplicaciones web convencionales, este tipo de reordenamiento suele ser una tarea relativamente tediosa, debido a que por cada reacomodo es necesario una nueva petición al servidor y una nueva recarga completa de la UI .

Como se observa en [3] y en [14], la funcionalidad asociada a la interacción de tipo *drag and drop* no es única, sino que puede estar asociado a diferentes tareas. El primer uso en el que este tipo de interacción puede ser empleado naturalmente es en el reordenamiento de elementos de datos mostrados en forma de lista (Figura 12). En segundo lugar, la técnica de *drag and drop* suele ser útil para conformar colecciones de objetos arrastrando los mismos desde una lista o colección a un área especial destinada a tal fin; este comportamiento es capturado en el patrón *Remembered Collections* [14]. Finalmente, la ejecución de una acción particular sobre el objeto siendo arrastrado representa otro uso cotidiano de la técnica. En este último caso, el elemento en cuestión es llevado y depositado en un área especial, teniendo esto como semántica subyacente el ejecutar alguna acción o

tarea asociada al mismo como, por ejemplo, su eliminación.

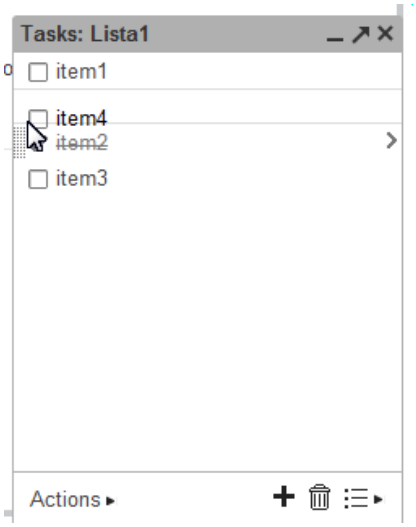


Figura 12. Reordenamiento de tareas a través de interacción de tipo *drag and drop* en el manejador de tareas recientemente integrado a GMail.

Dentro de los usos no asociados directamente a los objetos de datos, el reacomodamiento de módulos de UI se presenta con frecuencia, como se ejemplifica en la Figura 13 y se menciona en [14] con el pattern D*rag and Drop Modules*.

Figura 13. Módulos de aplicación que pueden ser reacomodados con *drag and drop* (http://igoogle.com)

Edición interactiva

En las aplicaciones web tradicionales, la edición de las propiedades concernientes a un objeto de datos particular implica tres pasos básicos en términos de interacción con la interfaz de usuario que permite llevar a cabo esta tarea. Los primeros dos corresponden a la selección del objeto en cuestión y navegación hacia una segunda UI compuesta por un formulario con sus valores actuales, los cuales son suceptibles a ser modificados. La

modificación se hace concreta al ser enviado este formulario con los nuevos valores. Una tercer UI será cargada en última instancia, haciendo visible el elemento modificado con su contenido actualizado.

La capacidad de modificar la interfaz de usuario en el client-side sin intervención del servidor y la posibilidad de realizar invocaciones asincrónicas al mismo desde la UI abren la posibilidad de simplificar este esquema, a fin de mejorar la interactividad y reducir la carga en el servidor. La edición interactiva, mencionada en [3] como el patrón *Malleable Content* y en [14] como *Inline Edit*, brinda precisamente esta posibilidad. En el esquema de interacción que estos patterns describen, los objetos de datos son mostrados frente al usuario y un evento particular como, por ejemplo, un click en el campo adecuado o la presión de un componente de interfaz de usuario, habilita la edición de una o más de sus propiedades. Un evento de UI de índole similar hace efectivos los cambios enviando directamente una notificación al servidor o bien registrando el hecho para que sea notificado en forma diferida. No se requiere para efectuar esta tarea llevar a cabo navegación alguna, lo cual ahorra la consecuente carga completa de otras interfaces de usuario a fin de implementar la misma funcionalidad. La Figura 14 describe un ejemplo de edición interactiva en GMail Chat. Los cambios efectuados son comunicados al servidor al producirse un evento similar al que inicia la modificación, por ejemplo, un click en un componente de UI particular o fuera del área de edición.

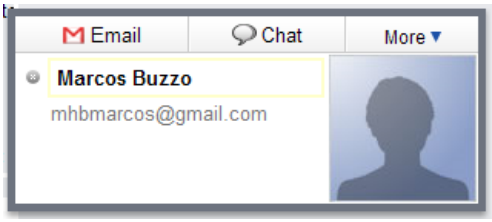


Figura 14. Capacidad de edición interactiva del nombre de un usuario en la interfaz de chat de GMail. La UI descripta en esta figura es la misma que en la Figura 9.

En lo respectivo a la interacción y al evento que dispara la edición, pueden definirse distintas variantes [14]. La edición puede ser llevada a cabo en el mismo espacio ocupado por el componente que se encontraba en modo lectura (cómo en el ejemplo de la Figura 14) o bien por una interfaz aparte, la cual se hace visible en el mismo nodo navegacional como, por ejemplo, un *popup*. El evento disparador puede ser un simple click en el área propensa a ser editada, como en el ejemplo de la Figura 14, o bien pueden proveerse como alternativa componentes especiales de UI que denoten esta posibilidad como, por ejemplo, un botón o link. Lo mismo se aplica a la característica de UI utilizada para indicar que se desea finalizar la edición y hacer efectivos los cambios.

La principal consecuencia del uso de este patrón de interacción es una reducción considerable en el tráfico entre cliente y servidor. En primer lugar, no es necesaria navegación alguna con el objeto de modificar un elemento de datos, lo cual también representa una mejora en cuanto a tiempo de respuesta de la UI, adicionando a esto la posibilidad de notificar al servidor de los cambios asincrónicamente o incluso en forma

diferida. En segundo lugar, explotando la capacidad de ejecución en el client-side relativas a las RIAs, puede detectarse en el mismo cliente qué valores han sido modificados y, por ende, puede evitarse el enviar atributos que no hayan sufrido cambios. Esto último, bajo el esquema básico planteado, es imposible en las aplicaciones web tradicionales.

5.3.7 Presentación

Como ya se dijo con anterioridad, las capacidades de interacción que las RIAs brindan en el client-side, en contraposición con las aplicaciones web actuales, representan mejoras en términos de interactividad y de comunicación entre cliente y servidor, provenientes de la migración de la arquitectura thin-client a la variante C-S. Asimismo, estas mismas características abren paso a nuevas posibilidades en términos de presentación de la UI. De esta manera, en la actualidad, una gran cantidad de aplicaciones han explotado estas ventajas, enriqueciendo su interfaz de usuario con efectos gráficos y componentes de UI altamente interactivos.

Un ejemplo común de este tipo de mejoras en cuanto a presentación es el uso de *Progress indicators* [14][3], componentes de UI que muestran el grado de completitud de cierta tarea a medida que la misma se va llevando a cabo (por ejemplo, el envío de un archivo). Otro ejemplo tradicional es el uso de indicadores pasivos a fin de denotar cierto resultado de una operación, mostrados en alguna parte de la UI (pattern *Inline Status* [14])

En términos de animaciones, existen distintas variantes destinadas a embellecer cambios en componentes de la UI como, por ejemplo, modificaciones en color, opacidad, y resaltado. Varios patterns capturan estas características: *One-Second Spotlight*[3], *One-Second Mutation*[3], *Brighten*[14], *Cross Fade*[14], entre otros.

Figura 15. Patrón *Inline Status* presente en GMail: se muestra el resultado de la última operación efectuada sobre items de la lista de correos electrónicos.

5.3.8 Validación

La validación de los datos ingresados por el usuario y de la consistencia del estado de la UI en relación a los mismos es una necesidad presente en la mayor parte de las aplicaciones. En las Aplicaciones Ricas existe la posibilidad de mejorar la interacción, proveyendo validación llevada a cabo en el client-side y, a su vez, de incrementar la seguridad efectuando la misma también en la faceta servidor. Esta necesidad de duplicar las validaciones cuyo origen subyace en fortalecer la seguridad en el server-side conlleva a redundancia de código, implementando la misma acción de validación tanto en el cliente como en el servidor, posiblemente bajo distintos lenguajes y plataformas. Varios frameworks web actuales han considerado aspectos de validación entre sus facilidades para desarrollar aplicaciones, proveyendo distintos tipos de elementos denominados *validadores*,

los cuales llevan a cabo las tareas de validación más comunes de manera automática tanto en la faceta cliente (de ser posible) como en la faceta servidor. Un ejemplo de un framework orientado a la construcción de aplicaciones web tradicionales que implementa esta característica es Tapestry[24], para la plataforma Java. En el caso de aquellos que implementan características de interacción ricas, la validación es efectuada automáticamente en el cliente, aunque no siempre es llevada a cabo también en el server-side. Esta técnica es descripta en el pattern *Inline Validation*[14] y un ejemplo real de su aplicación puede observarse en la Figura 16.

El estado de la UI también es usualmente objeto de validación, por ejemplo, en situaciones donde para realizar una tarea es necesario previamente la selección de un elemento con la que se llevará a cabo. Estos aspectos son inherentes al uso y especificación interactiva de la UI, así como también a la semántica y configuraciones posibles de los datos que en ella se representan. En estos casos, en lugar de mostrarse un mensaje de advertencia al usuario, la UI puede adecuarse a fin de no permitir el ingreso de información semánticamente incorrecta. Este comportamiento es capturado en el pattern *Live Form*[3].

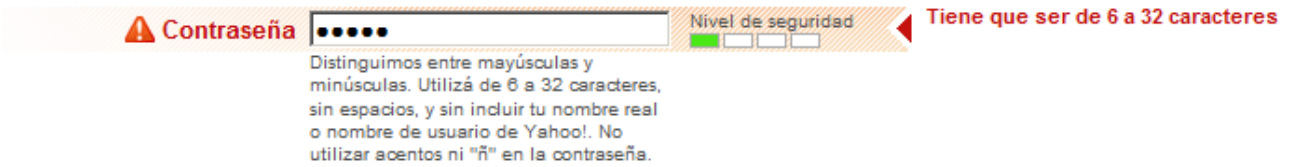


Figura 16. Aplicación del patrón *Inline Validation* presente en la pantalla de registro de un nuevo usuario en el portal Yahoo. De modo interactivo se valida que la longitud de la contraseña ingresada sea la correcta.

5.3.9 Login

El login y la organización de permisos de acceso a funcionalidades es un aspecto omnipresente en gran cantidad de aplicaciones y sistemas modernos, sobre todo en los relacionados a la web. En las plataformas web tradicionales, un error en los datos de autenticación involucra una recarga completa de la interfaz de login. Para apaliar este problema, la capacidad de interacción mejorada presente en las RIAs permite implementar interfaces de login como las mencionadas en el pattern *Direct Login*[3], el cual describe la posibilidad de efectuar el envío de los datos de autenticación bajo algún esquema seguro que tiene origen en el client-side (por ejemplo, hashing o alguna variante del mismo). Con esta técnica, no existe la necesidad de efectuar navegaciones extras para mostrar el resultado de la autenticación, pudiendo el mismo informarse a través de transformaciones en la UI. Técnicamente, este login puede ser implementado mostrando una vista a modo de *popup*, en la cual el usuario ingresa sus credenciales, recibiendo luego la respuesta correspondiente, sin involucrar en el proceso acción de navegación alguna.

El esquema de *Direct Login* no sólo evita el efectuar acciones de navegación para llevar a cabo la tarea de login, sino que también mejora la interactividad en caso de que ocurriese un *time out* de sesión. En este último caso, si la sesión se perdiera o caducara, se podría requerir nuevamente la autenticación en una interfaz tipo *popup*, sin necesidad de abandonar el nodo navegacional actual ni de perder el estado actual de la UI rica (por

24 http://tapestry.apache.org/

ejemplo, datos ingresados que aún no fueron notificados al servidor). Finalmente, el optar por una arquitectura cliente-servidor como la correspondiente a las RIAs en lugar de la arquitectura de clientes livianos de las aplicaciones web tradicionales tienen como ventaja adicional que, al poseer el cliente más capacidades de interacción y comportamiento, gran parte del estado de ejecución que antes era almacenado en el server-side puede ahora ser mantenido y modificado directamente en el cliente. Esto conlleva a una mejor interacción en el client-side, y a una reducción considerable de la carga en el server-side.

Consideraciones para una arquitectura de capa de servicios RESTful

La técnica utilizada comúnmente para la posterior autenticación una vez que el usuario ingresó sus correspondientes credenciales, consiste en que el servidor genere un código de identificación único que el cliente almacena (por ejemplo, a través de una cookie en el browser) y envía al server con cada petición. Esto requiere que el servidor sea el encargado de mantener el estado de la sesión, preservando como mínimo la asociación entre el código de identificación y el usuario correspondiente al mismo, de manera de poder ejercer los permisos de acceso adecuadamente. Dado que una característica importante de los servicios implementados según la arquitectura REST es que el servidor no mantenga estado de ejecución alguno, el esquema planteado atenta contra la misma. Una variante que puede aplicarse sin alterar los principios de la arquitectura REST y que además permite al servidor controlar el acceso a recursos es utilizar el mecanismo de autenticación ya incluido en el protocolo HTTP. A fin de preservar las credenciales de usuario, la información puede ser transferida a través de conexiones seguras provistas por el protocolo HTTPS.

5.3.10 Estado del arte en lenguajes y metodologías de modelado de aplicaciones web tradicionales

Dada la existencia de varias metodologías y lenguajes de modelado web en la actualidad y, debido a que las Aplicaciones Ricas y las aplicaciones web tradicionales comparten aspectos en común, el análisis de los mismos puede resultar de utilidad para definir características del meta-modelo a especificar. Más allá de los artefactos particulares que cada lenguaje de modelado define, es de interés tener en cuenta qué aspectos o conceptos considera representar cada uno en su meta-modelo, a fin de que un extracto de los mismos puedan influir o directamente ser incluido en el meta-modelo que en este documento se propone.

En [18] se busca unificar los lenguajes de modelado web WebML[11], OO-H y UWE[9]. Si bien en el mismo documento se fundamenta el porqué no se pueden integrar directamente los artefactos correspondientes a las distintas metodologías para formar un lenguaje único, se propone la idea de definir un meta-modelo general que resuma las características de los tres. Este último servirá también como *pivot* para efectuar conversiones de modelos entre los distintos lenguajes. En el paper no se define un meta-modelo formal particular, sino que se utiliza una estrategia top-down para definir los aspectos primordiales que cualquier lenguaje orientado al modelado de aplicaciones web debe considerar. En concreto, se definen una serie de *layers* o capas las cuales son ordenadas de acuerdo al grado de abstracción con respecto a las características de aplicación que permiten especificar, a saber:

- Capa 0: Modelado de contenido. Consiste en definir los objetos del dominio y sus propiedades, los cuales serán utilizados luego por la aplicación.
- Capa 1: Modelado de hipertexto. Considera los requerimientos de navegabilidad y obtención de datos para ser mostrados en nodos navegacionales. Esta capa considera:
  - *Navegación Global*: La especificación de la navegación entre distintas páginas que componen la aplicación, partiendo de un punto inicial (home page).
  - *Publicación de contenido*: Se define una página la cual se listan una serie de objeto de dominio y se muestra por cada uno de ellos cierto conjunto de propiedades.
  - *Publicación de contenido paramétrica*: Se define una página con similares características y objetivos que en el caso anterior, brindando para cada objeto listado un mecanismo de navegación. Este mecanismo permite que, por cada objeto, el usuario pueda navegar a otra página la cual contenga información asociada al mismo.
  - *Publicación de contenido con asociación explícita de parámetros*: Se define una página que contiene uno o más campos de entrada de datos cuyo contenido se utiliza para listar cierto conjunto de objetos de dominio. Los objetos listados deberán satisfacer cierto predicado cuyos términos incluyen los valores ingresados por el usuario en los mencionados campos de entrada.
- Capa 2: Modelado de manejo de contenido. Esta capa define métodos para especificar operaciones que permitan actualizar los objetos de datos del dominio y sus relaciones.

En el caso de la capa 0, dado que las Aplicaciones Ricas se caracterizan principalmente por mejorar la capacidad de comportamiento en el client-side, no hay en lo respectivo a definir el modelo de datos mayores diferencias entre los aspectos del meta-modelo a considerar y los ya existentes para aplicaciones web tradicionales.

La capa 1 y 2 consideran aspectos de interactividad que en las aplicaciones web tradicionales están mayormente relacionadas con la navegación y la publicación de datos a través de formularios. En este contexto, los aspectos relativos a los meta-modelos existentes y el definido en este documento pueden variar sustancialmente debido a que las Aplicaciones Ricas, como ya se mencionó en la sección *Multi-page vs. single-page web applications*, la navegación puede ser reemplazada por otras actividades. Este reemplazo es factible gracias a la capacidad de modificar la UI desde el client-side y sin intervención del servidor. Estas diferencias no modifican las actividades que el meta-modelo deberá permitir especificar en cuanto a trabajo y procesamiento de datos se refiere, aunque sí establece distintas maneras de representar, en términos de interactividad, cómo las mismas serán llevados a cabo.

5.3.11 Estado del arte en el modelado de Rich Internet Applications

En [19] se mencionan distintas problemáticas y posibles soluciones relativas al diseño y especificación de RIAs que deben tenerse en cuenta en los lenguajes que las especifiquen.

En lo respectivo a los datos, como ya fue comentado, las Aplicaciones Ricas permiten el almacenamiento en el cliente de los mismos, los cuales a bajo nivel son mantenidos por el *runtime* sobre el que corre a la aplicación. Si bien esto proporciona ventajas en términos de tráfico cliente-servidor y en cuanto a hiperactividad, da origen a los problemas de distribución y consistencia de datos entre ambas facetas y al almacenamiento sensible de datos en el client-side, los cuales pueden llegar a ser extraídos desde el *runtime*. La publicación considera distintos niveles de duración y persistencia de datos de acuerdo a su volatilidad en el client y server-side, así como también establece ciertos niveles de granularidad para mantener consistencia entre los datos distribuidos entre ambas facetas. Como conclusión, el modelador debe ser capaz de elegir los datos que serán almacenados en el client-side y el tiempo que durarán en el mismo, así como también cómo y con qué frecuencia serán sincronizados con su réplica en el server-side.

En cuanto a la lógica de negocios, la arquitectura cliente-servidor que subyace a las RIAs permite que en el client-side puedan efectuarse tareas de complejidad similar a las que ocurre en el server-side. Esto implica que la lógica de negocios puede dividirse entre las facetas cliente y servidor de acuerdo a las necesidades particulares de la aplicación. En este contexto, es posible el definir la lógica de negocios de manera de que abstraiga de dónde es ejecutada, y luego llevar a cabo un refinamiento en la especificación que detalle este aspecto particular.

El términos de presentación, las RIAs difieren de las aplicaciones web tradicionales en las que la UI es conformada por un documento. Las interfaces de usuario relativas a las Aplicaciones Ricas son similares a las aplicaciones desktop y, como tales, pueden proveer mayor detalle acerca de cómo los elementos que las componen son organizados y mostrados. Aunque representa una ventaja en términos de la personalización del aspecto visual de la aplicación, una principal desventaja de esta característica está relacionada con los problemas de adaptabilidad de la UI a distintos dispositivos cuyas características físicas y funcionales como, por ejemplo, tamaño y resolución de pantalla, son muy particulares. Una solución posible en términos de modelado es permitir en el mismo especificar distintas vistas o componentes a utilizar según el tipo de dispositivo en el cual la aplicación corra, de manera que se adapte la misma a las características físicas y funcionales del mismo.

Finalmente, el último aspecto a considerar es la comunicación. La misma representa distintos aspectos de las aplicaciones: transferencia y sincronización de datos, sincronización con tareas distribuidas concernientes a la lógica de negocios particular de la aplicación y transferencia de fragmentos de UI. Este último aspecto es muy característico de las Rich Intenet Applications, dada su migración al paradigma *single-page web application*. Estos tres aspectos que conforman *cross-cutting concerns* deben diferenciarse en el meta-modelo, ya sea explícita o implícitamente.

6 DESCRIPCIÓN INFORMAL DEL META-MODELO

6.1 Introducción

En esta sección se describirá la composición del meta-modelo utilizado para definir Aplicaciones Ricas. La elaboración del mismo se basa en los principios y aspectos a considerar en el desarrollo de RIAs, de acuerdo a la investigación efectuada y documentada en secciones anteriores en términos de patterns y tecnologías orientadas a su diseño e implementación.

Para acotar el dominio y, de esta manera, dotar de más semántica al meta-modelo, el mismo tendrá como destino la confección de Aplicaciones Ricas centradas en datos. Los conceptos del lenguaje serán introducidos a medida que se vaya avanzando en la descripción de las funcionalidades que el meta-modelo permitirá definir. A su vez, la descripción de los mismos no será completamente formal, sino que tendrá un carácter introductorio. En secciones posteriores, se describirán formalmente los componentes, reglas y semántica del meta-modelo a través de un lenguaje de meta-modelado, así como también aspectos de su implementación y representación en la herramienta elegida.

El lenguaje de dominio específico será referido como RIADL (Rich Internet Application Definition Language).

6.2 Datos

Los datos que la aplicación manejará son especificados en el meta-modelo con la introducción del concepto de *entidad*. De manera similar a las clases en el lenguaje UML, una *entidad* tiene un nombre y una serie de *atributos*, teniendo cada uno de estos un tipo de dato particular asociado. Los tipos de datos considerados incluyen aquellos tipos primitivos y tradicionales como `Integer` y `String`, así como también son considerados tipos de mayor abstracción o relativos a multimedia como `Date`, `Image`, etc.

De manera similar al meta-modelo Entidad-Relación, dos *entidades* se relacionan entre sí a través de una *relación*, la cual posee un nombre que la distingue. Por simplicidad, una *relación* en RIADL puede estar asociada a exactamente dos *entidades*. Cada conexión entre una *entidad* y una *relación* tiene una cardinalidad mínima y máxima, las cuales representan la cantidad de instancias mínima y máxima de esa *entidad* que pueden participar en una *relación* con un elemento de la otra respectivamente. A su vez, dichas conexiones también poseen un *nombre de rol*, el cual expresa cómo es interpretada la *relación* desde el punto de vista de la *entidad* vinculada por la conexión.

La definición en RIADL del concepto de *relación* necesario para vincular *entidades*, permite identificar las asociaciones entre éstas y facilitar el especificar navegaciones de datos cuando fuera necesario. Los *nombres de rol* tienen, en primera instancia, un fundamento expresivo, denotando cómo se aprecia la relación desde una *entidad* hacia la otra. Adicionalmente, los mismos permiten que el código derivado incluya esta información directamente en la implementación, por ejemplo, utilizándolos para definir identificadores.

La Figura 17 muestra un modelo de ejemplo extremadamente simple para modelar

libros en una biblioteca. Un autor está representado por la *entidad* `Autor` y posee tres *atributos*: un nombre (`nombre` de tipo `String`), una fecha de nacimiento (`fechaDeNacimiento` de tipo `Date`) y una biografía (`biografia` de tipo `String`). Un libro es modelado a través de la *entidad* `Libro` y posee 4 *atributos*: un título (`titulo`, de tipo `String`), un ISBN (`ISBN`, de tipo `String`), una fecha de publicación (`fechaDePublicacion`, de tipo `Date`) y una imagen de portada (`imagenDePortada` de tipo `Image`). A su vez, la *relación* `autorLibro` y sus cardinalidades correspondientes establecen que un autor puede tener asociado múltiples libros (lo cual representa sus libros publicados) y un libro debe poseer indefectiblemente un autor asociado.

Figura 17. Esquema de datos representando a dos *entidades* (`Autor` y `Libro`), sus propiedades y tipos correspondientes y una *relación* (`autorLibro`) entre las mismas

Como fue planteada, esta fracción del meta-modelo orientada a definir la estructura de los datos que la aplicación manejará no se diferencia en gran medida de otras pertenecientes a distintos lenguajes o meta-modelos existentes de especificación de aplicaciones web tradicionales como, por ejemplo, WebML[11]. Eso se debe, como ya se ha mencionado con anterioridad, al hecho de que en términos de la especificación del dominio de datos subyacente, las aplicaciones web tradicionales y las Aplicaciones Ricas no poseen grandes diferencias.

6.2.1 Capa de servicios: definición y características que debe satisfacer

Dado los beneficios provistos ya mencionados, el acceso a los datos descriptos a través de *entidades* y sus *relaciones* serán efectuados a través de una capa de servicios. En particular, debido a las simplicidad y definición de los datos en lugar de las operaciones como centro de la arquitectura, la capa de servicios será implementada siguiendo la arquitectura REST. Por cada *entidad* se generará un recurso particular, el cual permitirá efectuar lecturas, altas, bajas y modificaciones, así como también crear o destruir *relaciones* entre instancias de datos. A modo de ejemplo, la generación de la capa de servicios correspondiente al modelo de la Figura 17 resultará en la definición de dos recursos principales distinguidos por URLs particulares, uno relativo a la *entidad* `Libro` y otro a la *entidad* `Autor`. En términos técnicos, siguiendo los preceptos de la arquitectura REST, el acceso a funcionalidades básicas relativas al manejo de instancias de *entidades* se llevará a cabo a través de peticiones HTTP al servidor, de acuerdo a la semántica de la acción que se quiera realizar, acarreándose en dichas peticiones o bien en sus respuestas representaciones de las susodichas instancias en formatos particulares (por ejemplo, XML).

En lo relativo al acceso a las instancias de datos correspondientes a cada *entidad*, por cuestiones de eficiencia y requerimientos, es necesario proveer en la capa de servicios funcionalidades que permitan definir restricciones al conjunto de instancias que se desea obtener. Dos ejemplos de esta funcionalidad pueden ser la capacidad para acceder al

contenido de a páginas o bien para seleccionar instancias de datos que satisfagan determinado predicado lógico.

Asimismo, las instancias de datos relacionadas a datos requeridos correspondientes a un recurso pueden ser ignoradas, referenciadas, o directamente incluidas dentro del contenido retornado al efectuar una petición a la capa de servicios. De optarse directamente por la inclusión, puede que se transfieran gran cantidad de datos desde el servidor hacia el cliente sin que los mismos sean necesarios, lo cual se traducirá en una gran cantidad de tráfico generada en vano. De igual manera, cuando se requiera acceder a las instancias de datos relacionadas a los datos que se solicitaron inicialmente, el no incluirlos en la respuesta a la petición solicitada puede llevar a efectuar un número importante de peticiones al server-side para obtener la información faltante. Una opción razonable es que el client-side sea capaz de especificar qué instancias de datos relacionadas a la solicitud principal desea obtener de acuerdo al uso que requiera de la información. Siendo esta funcionalidad importante en términos de eficiencia por lo dicho anteriormente, se hace necesario considerarla a la hora de definir la capa de servicios que será derivada.

Todas estas características, si bien son de vital importancia en la capa de servicios, no requieren ser consideradas directamente en RIADL como conceptos particulares que el modelador debe especificar. Esto se debe a la relación directa existente entre una *entidad* a nivel de meta-modelo y un *recurso* en la capa de servicios a generar. Sin embargo, como se verá luego, al definir los aspectos del meta-modelo relativos a la UI, será indispensable el uso de las características que en esta sub-sección se mencionan, aunque no se vean de manera explícita como un aspecto del meta-modelo.

6.3 Validación

La validación de datos es una tarea frecuente a llevar a cabo, más aún en aplicaciones centradas en su manejo y procesamiento, como ya se ha mencionado. Dado que la validación de datos tiene como objeto que la información que será almacenada en instancias de *entidades* sea consistente, el meta-modelo propuesto la considera en relación a *entidades*, más allá de que comúnmente esta verificación se especifique e implemente a nivel de la UI. Luego, el código necesario para efectuarlas puede ser derivado y trasladado directamente a la interfaz de usuario, buscando realizar la mayor cantidad posible de las mismas en el client-side, en pos de alcanzar una mejor interacción. Bajo este mismo esquema, el código de validación puede ser incluido en el server-side, con el objeto de proteger el backend de la aplicación.

En RIADL, la obligatoriedad o no de proveer un valor para un *atributo* de *entidad* particular, siendo una restricción que usualmente es necesaria explicitar, es definida directamente como una propiedad inherente a los *atributos*, y es representada por un valor booleano. Para campos textuales, la validación de contenido en base a expresiones regulares permite amplia libertad para validar la forma que el input deberá respetar, por lo que se permite asociar este tipo de expresiones para *atributos* de tipo `String`, relacionando los mismos a componentes de validación denominados `RegExp`. Este tipo de componentes se referirán en el meta-modelo como *validadores*.

Para *atributos* con tipo de dato ordinal, como los numéricos, fechas, etc., se definen *validadores de rangos* los cuales son denominados `Range`. Los mismos son capaces de

corroborar que los datos ingresados estén dentro de determinado intervalo de valores.

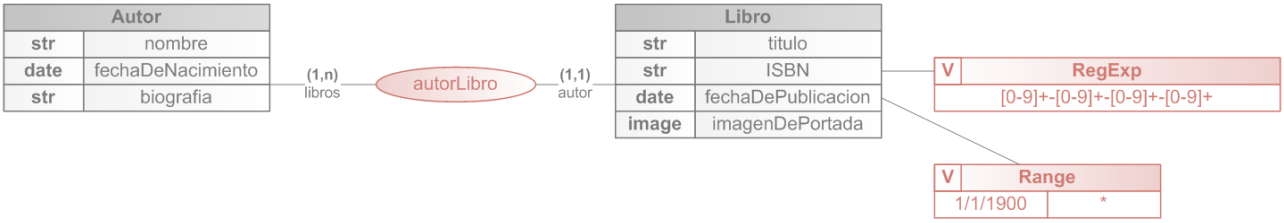


Figura 18. Uso de validadores de tipo `Range` y `RegExp` en el modelo de datos planteado.

El ejemplo de la Figura 18 muestra cómo especificar un validador de expresiones regulares para restringir el formato del campo `ISBN`, y un validador de rango a fin de no permitir registrar libros anteriores a el año `1900`.

6.4 Interfaz de usuario: navegación y vistas

Dada la diferencia ya planteada entre aplicaciones web *single-* y *multi-page*, y pudiendo las Aplicaciones Ricas implementar y combinar características concernientes a ambas, el meta-modelo define, con el objeto de representar una UI de manera independiente en lo relativo a este aspecto, un elemento del lenguaje denominado *vista*. Una *vista* puede contener dentro de sí un conjunto de componentes de interfaz de usuario simples o compuestos, los cuales además pueden ser estáticos como, por ejemplo, texto o imágenes, o dinámicos o que permitan cierto grado de interacción como, por ejemplo, *textboxes*, *botones* o *date pickers*. A su vez, una *vista* puede conformar, o bien un nodo navegacional en términos de una aplicación *single-page*, o bien un conjunto de componentes que serán mostrados en un momento determinado en una aplicación *multi-page*. La representación gráfica de este tipo de componente en RIADL puede observarse en la Figura 19.

Figura 19. Representación gráfica de una *vista*

Para diferenciar entre un nodo navegacional y un conjunto de componentes que serán mostrados en el mismo nodo de navegación actual, las vistas tienen como propiedad el *estilo* (propiedad `Style`) en el que se harán visibles, el cual tiene dos valores posibles: `page` o `popup`. El estilo `page` hace corresponder a la vista con un nodo navegacional, mientras que el estilo `popup` establece que el conjunto de componentes que la misma incluye serán mostrados en el mismo nodo de navegación actual, en modo de *popup*. Adicionalmente, una vista dispone de una propiedad `Title`, la cual define el título que la página o ventana mostrará.

Los elementos compuestos del modelo, es decir, aquellos que pueden contener otros sub-elementos como por ejemplo las *vistas*, pueden establecer *acciones* que se llevarán a cabo dentro de su contexto y que serán ejecutadas frente a la ocurrencia de determinado evento de interfaz de usuario. La *acción* de navegación o *muestra* de una *vista* es denominada `Show` y tiene como parámetro otra *vista*, la cual es presentada de acuerdo a su propiedad `Style` (Figura 20).

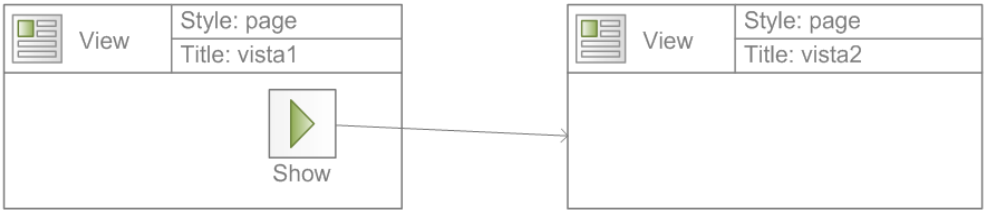


Figura 20. Representación gráfica de una *acción* de navegación desde una *vista* hacia otra.

Este esquema permite establecer la navegación básica entre páginas, disparadas a partir de eventos de interfaz de usuario en Aplicaciones Ricas y, en algunos casos, también presentes en aplicaciones web tradicionales como, por ejemplo, un click en un link (Figura 21). Los elementos de UI y las acciones se asocian a partir de un tipo de evento particular, el cual establece que frente a su ocurrencia, la acción se ejecutará correspondientemente. Cada elemento de UI compuesto establece una serie de eventos válidos a los cuales pueden asociarse acciones.

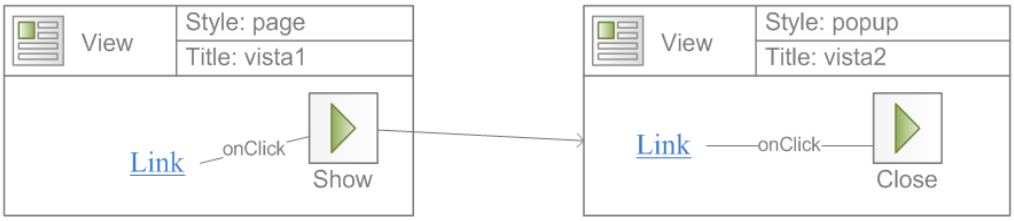


Figura 21: Navegación efectuada a partir de un evento de click en un *link*. Dado el valor de la propiedad `Style` en `vista2`, la misma se muestra sobre el contenido de `vista1`, en lugar de navegarse hacia ella.

En una primera instancia, la navegación puede definirse a través de acciones `Show` incluidas dentro de cada *vista* particular, sin necesidad de considerar los componentes de UI que las dispararán. Esto permite que la especificación de los aspectos de navegación puedan llevarse a cabo en una etapa preliminar y se abstraigan de aspectos posteriores de la interfaz de usuario.

Resta aclarar que, a diferencia de las *vistas* en modo `page`, aquellas en modo `popup` no tienen permitida la navegación a otras *vistas*, dado que se encuentran contenidas dentro de una en particular. Debido a esto, la acción `Show` no puede incluirse en el contexto de una *vista* de este tipo. La acción permitida en su lugar es llamada `Close`, y produce el ocultamiento de la *vista* que la contiene y el retorno de control de la interfaz de usuario a la *vista* invocadora (Figura 21).

6.4.1 Tipificación de usuarios y permisos de acceso

Siendo la autenticación y control de acceso una tarea frecuente en las aplicaciones web, el meta-modelo considera la posibilidad de clasificar distintos tipos de usuario y restringir su permiso a diferentes áreas de la aplicación, automatizando también la implementación de una interfaz de login y administración de privilegios de acceso. De no especificarse estas características, se asume que todo usuario tiene acceso a la totalidad del sistema y no se incluye interfaz de manejo alguna.

La autenticación frente al sistema es llevada a cabo automáticamente de ser necesario, siendo el código que implementa esta funcionalidad derivado automáticamente si la misma se requiere. De igual manera, la implementación del manejo y administración de usuarios también es generada directamente. La interfaz de login seguirá el pattern *Direct Login*[3] descripto en detalle en la sección anterior, buscando promover la mayor interactividad posible en la aplicación en lo relativo a autenticación.

El acceso es restringido a nivel de *vistas*. RIADL permite definir un conjunto de *roles de usuario*, pudiendo los mismos pertenecer a una de tres clases posibles (*meta-roles*):

- `<<guest>>`, el cual incluye a todos los usuarios no autenticados frente al sistema.
- `<<user>>`, el cual identifica a cualquier usuario autenticado en el sistema que no tiene permisos para utilizar la interfaz de administración de usuarios.
- `<<administrator>>`, el cual define aquellos *roles de usuario* que tienen acceso a la interfaz de administración de usuarios y privilegios de acceso.

Los *roles de usuario* se relacionan con una o más *vistas* para indicar permiso de acceso a través de una relación de tipo `access`, y deben tener obligatoriamente una *vista* asociada que conforme su página de inicio a través de una relación denominada `homePage`.

A su vez, los *roles de usuario* pueden heredar los permisos de acceso de otros usuarios a través de la relación `inherits`. Implícitamente, todos los *roles de usuario* definidos heredan naturalmente de `<<guest>>`.

En la Figura 22 se define un ejemplo de *modelo de roles y privilegios de acceso*. Los usuarios que no efectúen su login correspondiente (*meta-rol* `<<guest>>`), sólo podrán acceder a la página principal del sitio. Los usuarios registrados con *rol* de `Lector` podrán acceder a las *vistas* `Listar libros` y `Editar preferencias`, y no podrán acceder a la interfaz de manejo de usuarios. Por otro lado, los usuarios con *rol* de `Bibliotecario` podrán acceder a las mismas *vistas* que los usuarios con *rol* de `Lector`, dada la relación de herencia existente de los primeros hacia los segundos, en conjunción con otras tres *vistas* más. Adicionalmente, dado que el *rol de usuario* `Bibliotecario` tiene como *meta-rol* `<<administrator>>`, los mismos podrán acceder a la interfaz de manejo de usuarios.

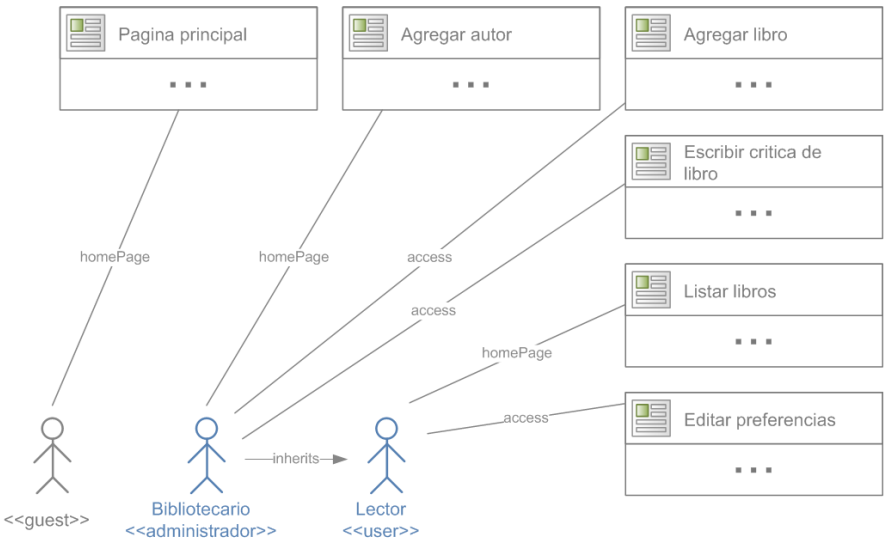


Figura 22: Ejemplo de modelo de *roles de usuario* y acceso a *vistas*.

Finalmente, dado que muchas veces es necesario asociar información personal a usuarios, los mismos pueden ser tratados como *entidades* y, por lo tanto, existe la posibilidad de incluirlos en el mismo modelo de datos. Asimismo, puede brindarse acceso en el meta-modelo a sus atributos característicos como, por ejemplo, su *username*.

6.5 Asociación entre elementos de UI y datos

Dado el dominio particular de aplicaciones orientadas a datos que RIADL se propone especificar, la asociación entre componentes de interfaz de usuario que conformarán las *vistas* y los datos es una característica importante a considerar en el lenguaje gráfico.

6.5.1 Contextos

Siguiendo una idea similar a la del framework OpenLaszlo, el cual permite definir para elementos visuales compuestos un origen de datos (*data source*) asociado que sus componentes internos pueden referenciar a fin de mostrar información obtenida desde el mismo, el meta-modelo considera lo que se denominan *contextos* o *contexts*. Un *contexto* es un elemento del lenguaje que contiene un grupo de sub-componentes y que hace referencia a una instancia de una *entidad* de datos particular. Cada subcomponente incluido en el *contexto* puede referenciar y mostrar una propiedad particular de la instancia de datos asociada al mismo.

Figura 23. Sintaxis concreta de una *vista* que contiene un *contexto* de información correspondiente a una instancia de `Libro`, el cual incluye ciertos componentes que muestran sus propiedades.

En la Figura 23 se muestra un ejemplo de una *vista* que contiene un *contexto*, el cual referencia a una instancia de la *entidad* `Libro`. Esta referencia se especifica a través del parámetro `Entity` definido en el mismo. Dentro del *contexto*, se definen tres componentes visuales: una imagen y dos elementos textuales. Mientras que la imagen referencia a la propiedad `imagenDePortada` de la instancia de `Libro` y está etiquetada como `Portada`, los otros dos elementos textuales se etiquetan como `Titulo` y `Fecha de Publicación` y muestran el contenido de los *atributos* `titulo` y `fechaDePublicacion` de la instancia de `Libro` respectivamente. La instancia concreta de `Libro` a la cual el *contexto* refiere se verá más adelante en esta descripción informal del meta-modelo.

Como efecto colateral del ejemplo orientado al uso de *contextos* mostrado en la Figura 23, se aprecia la introducción de dos nuevos tipos de componentes de UI presentes en RIADL:

- Los elementos textuales o simplemente *texts,* los cuales representan texto a mostrarse en la interfaz de usuario. El meta-modelo define dos tipos:
  - *Paramétricos* o *dinámicos*, como el mostrado en la figura del ejemplo, los cuales deben tener asociado un *atributo* correspondiente a la *entidad* referenciada por el *contexto* en el que están definidos. El *atributo* asociado a elementos textuales de este tipo deben tener como tipo de dato una cadena de caracteres (`String`) o bien cualquier otro tipo que pueda ser representado naturalmente en forma textual como por ejemplo, fechas (`Date`).
  - *Estáticos*, en los cuales el modelador define su contenido, el cual no cambiará en tiempo de ejecución.
- Imágenes (*Images*), las cuales representan cierto contenido gráfico a ser mostrado en la UI. Dicho contenido es obtenido desde *atributos* de tipo

`Image` declarados en la *entidad* referenciada por el *contexto* en el que están definidas, o bien es fijado estáticamente por el modelador.

Ambos componentes de UI tienen como propiedad común una *etiqueta* (`Label`), la cual describe su contenido y permite identificarlos en la interfaz de usuario.

6.5.2 Listas

Según lo planteado anteriormente en la sección *Estado del arte en lenguajes y metodologías de modelado de aplicaciones web tradicionales*, un requisito indispensable de un lenguaje de modelado de aplicaciones web es el permitir especificar la visualización de instancias de datos existentes en la aplicación y definir cierta capacidad para interactuar con las mismas. Este requerimiento se aplica de igual manera para las Aplicaciones Ricas y es capturado por componentes que en el meta-modelo propuesto se denominan *listas*.

Una *lista* es un elemento del lenguaje que, en principio, tiene una *entidad* de datos asociada y es capaz de mostrar en la interfaz de usuario todas las instancias de datos existentes relativas a la misma[25]. Al igual que ocurre en un *contexto*, los componentes de UI que se definen dentro de una *lista* pueden hacer referencia a los *atributos* de la *entidad* que se asocia a la misma. Sin embargo, a diferencia de un *contexto*, una *lista* replica la estructura de interfaz de usuario definida por sus componentes internos por cada elemento de datos particular, utilizando el contenido del mismo para brindarle a los subcomponentes los datos que requieren.

Asimismo, una *lista* define 2 propiedades las cuales permiten habilitar y configurar características de interacción ricas:

- `Paging`, la cual puede ser `none` o bien un número natural. La propiedad indica si la *lista* mostrará todos sus resultados en una sola unidad o bien dividirá el contenido de los datos en páginas y proveerá algún mecanismo de scrolling para navegar por las mismas. Si el valor de la propiedad es `none`, no se utilizará paginación alguna, mientras que de ser un número entero, el mismo indicará cuántos elementos por página se mostrarán. Como es de esperar, el avanzar entre páginas no implicará una recarga completa de la interfaz de usuario, sino que sólo será modificado el contenido interno de la *lista*, pudiéndose inclusive reusar los mismos componentes de UI ya alocados, cambiándose sólo su contenido o referencia a datos.
- `Selectable`, la cual puede tener 3 valores posibles: `none`, `one` o `many`. Esta propiedad determina si los elementos en la *lista* podrán ser seleccionados y, en caso de que sí, si la selección involucra sólo un elemento o bien muchos (selección múltiple). El valor `none` indica que no es posible selección alguna en la *lista*, mientras que los valores `one` y `many` indican que es posible la selección de sólo un elemento o bien de muchos respectivamente. En términos de UI, la selección puede ser efectuada por algún widget pertinente generado en la derivación como, por ejemplo, un *checkbox*, o bien utilizando eventos de interfaz de usuario como un click o la presión de una tecla

25 Formas adicionales para especificar la fuente de datos que la *listas* (y otros componentes orientados a datos) mostrarán se verán en secciones posteriores.

determinada.

La Figura 24 muestra un ejemplo de *lista* que referencia a la *entidad* `Libro` ya definida y, por ende, muestra todos las instancias de `Libro` que se hallan en la aplicación. La *lista* divide los elementos en páginas de a 10 y permite seleccionar un elemento particular. La Figura 25 muestra la estructura genérica de una *lista* a través de su sintaxis concreta, incluyendo la composición de sus propiedades.

El uso de los elementos seleccionados de una *lista* y otros flujos de datos en la interfaz de usuario, así como también optimizaciones relativas al tráfico entre client y server-side relativas a este tipo de componente serán mencionados posteriormente en este documento.

Figura 24: Ejemplo de *lista* que referencia a la *entidad* `Libro`.

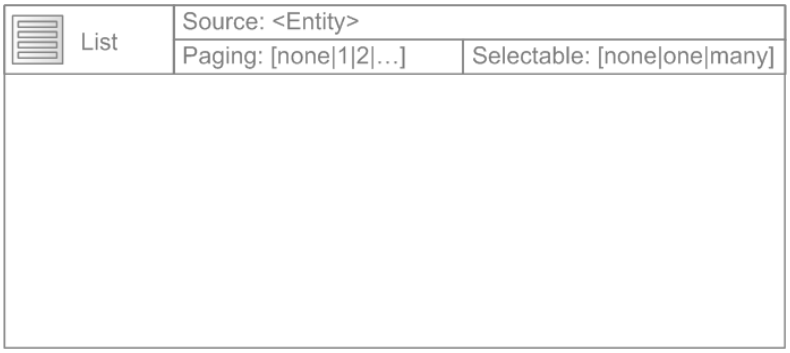


Figura 25. Estructura genérica de una *lista*

6.5.3 Filters

Las *listas* consideradas hasta el momento son capaces de mostrar todas las instancias de datos pertenecientes a una *entidad* particular. Sin embargo, muchas veces es indispensable acotar el conjunto de información que quiere observarse, a partir de parámetros especificados por el usuario o que la misma aplicación define, como se comenta

en [18]. Para poder efectuar esta selección, se define en RIADL un nuevo tipo de componente: los *filtros* o *filters*. Un *filter* se asocia a una *lista* y define ciertas restricciones lógicas utilizadas para seleccionar qué instancias de datos provenientes de la *entidad* que la *lista* referencia serán mostradas.

Los *filters* pueden contener componentes de interfaz de usuario orientados a datos a través de los cuales pueden obtener información para efectuar el filtrado de información, así como también pueden definirse valores constantes dentro de los mismos con un objetivo similar. En ambos casos, estos deben ser relacionados con *atributos* de *entidades* a través de una operación de comparación o similar, a fin de formar elementos básicos del predicado lógico que el *filter* define y, por lo tanto, se requiere un componente de lenguaje que permita referenciarlos. Los elementos de lenguaje que llevan a cabo esta tarea son los `Attributes`, los cuales representan a un *atributo* de la *entidad* referenciada por el contexto de datos de la *lista.*

Los *filters* a su vez dan la posibilidad de habilitar la sugerencia automática para componentes que permitan entrada de texto, siguiendo el pattern *Suggestion*[3] mencionado con antelación. La habilitación de esta funcionalidad se define asociando el componente de UI que debe implementarla con un componente del lenguaje denominado `Suggest`. Adicionalmente, los *filters* disponen de una propiedad booleana denominada `Live search`, la cual habilita la posibilidad de búsqueda dinámica a medida que el usuario ingresa la información de consulta, siguiendo el patrón *Live Search*[3].

En el ejemplo de la Figura 26, se establece un *filter* para la *lista* ya definida en la Figura 24, en la cual se seleccionan para ser mostrados aquellos Libros que hayan sido publicados luego del `1/1/1956`, y cuyo nombre de autor incluya el texto ingresado por el usuario en el *textbox* rotulado `Titulo`. A su vez, se determina que dicho campo de texto habilite su propiedad de sugerencia automática de datos, de manera de orientar al usuario en el ingreso de información relativa al título del libro en cuestión.

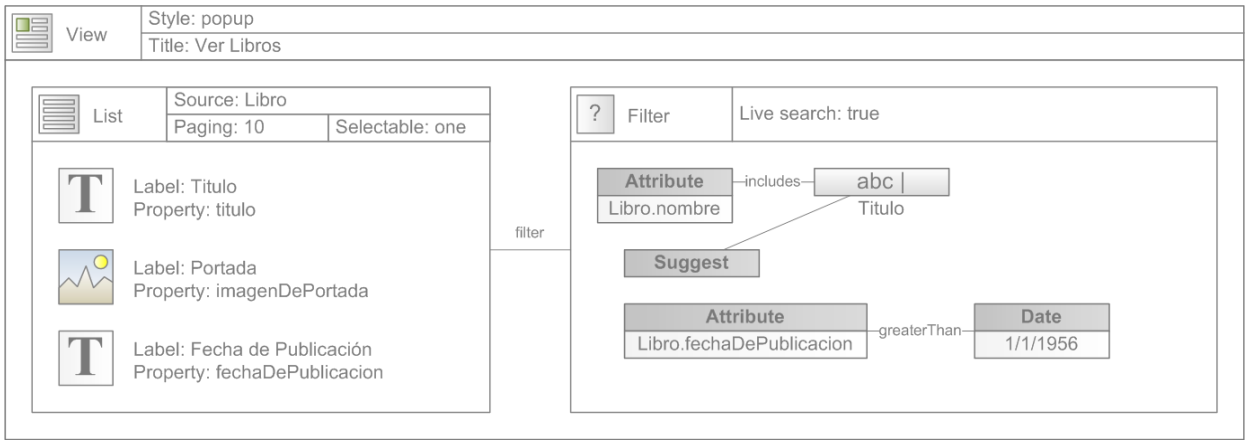


Figura 26. Componente *filter* asociado a la *lista* de `Libros` definida en la Figura 24.

Los *filters* pueden ser visibles o no en la interfaz de usuario, dependiendo de si incluyen o no componentes de UI con los que el usuario pueda interactuar. El valor posible de la propiedad `Live search` depende también de la existencia de componentes de UI que permitan ingreso de datos de consulta por parte del usuario, no pudiendo ser verdadera si no

se declara al menos uno de ellos en el *filter*.

A continuación, se detallarán y resumirán los nuevos conceptos de RIADL introducidos vagamente o de manera implícita en la Figura 26:

- Los campos de texto o *textboxes*, como el que se observa rotulado como `Titulo` en la figura del ejemplo, son componentes de UI que le permiten al usuario ingresar contenidos textuales. Su única propiedad a destacar por el momento es su etiqueta, similar a la ya vista para contenidos textuales e imágenes.
- El componente rotulado como `Date` representa un *valor* concreto de tipo `Date`, el cual puede ser usado en conjunción con un *attribute* a fin de realizar una operación de comparación o similar. El meta-modelo define este tipo de componentes para la mayoría de los tipos de datos básicos.
- Los elementos de tipo `Attribute` son utilizados para representar *atributos* de la *entidad* referenciada por la *lista* asociada al *filter*. Pueden relacionarse a través de un operador lógico a un componente de UI capaz de proveer datos de tipo compatible (como el *textbox* `Titulo` en la figura del ejemplo) o bien con *valores* constantes (como el valor `1/1/1959` de tipo `Date` especificado en la misma figura). Si bien en el ejemplo se utiliza el operador lógico *mayor a*, formalizado como `greaterThan`, el meta-modelo define operadores de comparación más comunes aplicados a tipos de datos conocidos.

6.5.4 Details

Como se mencionó en la sección *Detalles*, las características de interacción mejoradas presentes en la RIAs pueden explotarse a fin de permitirle al usuario obtener interactivamente y sin necesidad de navegación, información y comportamiento extra relativos al objeto de datos que está observando actualmente. Con este propósito, RIADL define una clase de componentes de UI denominado *detalles* o *detail*. Este tipo de componentes pueden contener un conjunto sub-elementos orientados a proveer datos o funcionalidad extra a la ya presente en la *vista* actual, los cuales serán mostrados por petición del usuario o a partir de la ocurrencia de un evento de UI particular. En [18] se resalta la necesidad de que un lenguaje de modelado de aplicaciones web provea esta funcionalidad, aunque en este caso, al no disponerse de riqueza en la UI, la misma está asociada obligatoriamente a una acción de navegación.

Dada su naturaleza, en principio, los componentes *detail* pueden referenciar y mostrar información en el contexto en el que están incluidos. Utilizados de manera aislada deben proveer, en términos de UI, un mecanismo para que el usuario pueda mostrarlos u ocultarlos. Otros posibles usos de este tipo de componentes, persiguiendo el mismo fin para el que son definidos, se verá a posteriori en conjunto con una generalización de cómo los componentes de UI especifican el origen de sus datos y el momento de su carga.

Teniendo en cuenta las optimizaciones mencionadas en cuanto a tráfico y performance en la comunicación entre cliente y servidor descriptas en la sección *Interactividad y performance en comunicación*, los *detalles* permiten personalizar el momento de la carga de su contenido de acuerdo a la interacción que el diseñador requiera a

través de la propiedad booleana `Prefetching`. De ser `true`, la misma implica que los datos que requieran ser cargados en el *detail* serán obtenidos antes de que el usuario desee observarlos, evitando su carga bajo demanda a futuro y la consecuente demora que esta conllevaría asociada. Como ya se ha comentado, esta ventaja se ve contrastada con la necesidad de una mayor transferencia de datos inicial al cargar la UI.

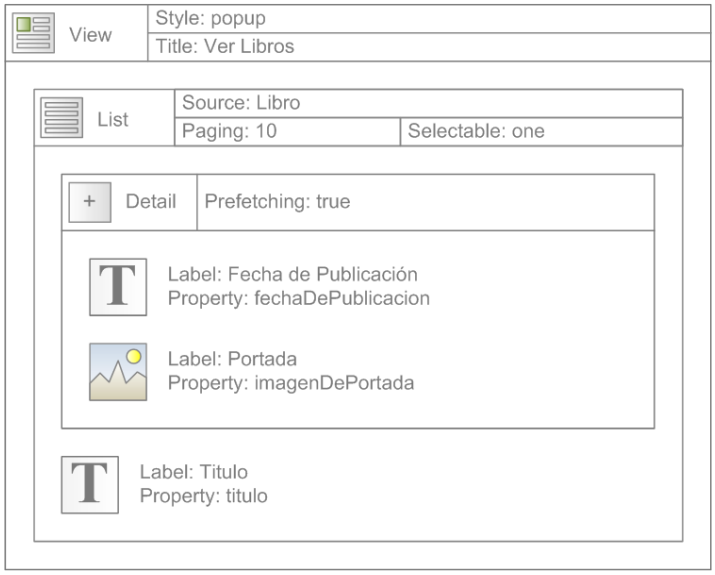


Figura 27. Uso de un componente *detail* para mostrar información extra de un libro particular en una *lista* de libros

En la Figura 27 se muestra un ejemplo de la utilización de *detail* en relación a una *lista* de elementos `Libro`. Visualmente, se mostrará el título del `Libro` y un enlace, botón o artefacto de UI similar que permita mostrar la información contenida en el *detail*, la cual en este caso consiste en la fecha de publicación y la portada de la instancia particular del `Libro` listado. Siendo verdadera la propiedad de `Prefetching`, el único dato extra a cargarse (la imagen de portada) será obtenido al inicializarse la UI, lo cual permitirá que, una vez cargada por completo la *vista*, el usuario no enfrente demoras extra relativas a la descarga de datos desde el server-side.

Si bien cumple con la funcionalidad buscada, el tipo de *detail* mencionado en la Figura 27 asume que los detalles correspondientes al objeto de datos a mostrar consisten en extra concernientes a la misma instancia particular. Sin embargo, puede que los datos que conforman la información detallada que se desea hacer visible se obtengan a partir de otros objetos relacionados con cardinalidad máxima `1`. Por ejemplo, para un `Libro` determinado, parte de la información de detalle puede ser el `Autor` asociado al mismo. Estos casos serán descriptos más adelante, cuando se describan aspectos del meta-modelo orientados a la obtención de instancias de datos asociadas a través de *relaciones*.

6.5.5 Details a modo de tooltips

Hasta el momento sólo se ha considerado el hacer visible la información detallada a partir de un evento de click en un componente de UI que el mismo *detail* generará al momento de la derivación. Sin embargo, es usual que se desee mostrar el contenido

detallado frente a la ocurrencia otros eventos de UI como, por ejemplo, el dejar estático el cursor unos segundos sobre un objeto determinado (observar el ejemplo retratado en la Figura 9). Este último caso sugiere la posibilidad de que los componentes *detail* se relacionan con cualquier otro componente en modo similar a los *tooltips* usualmente presentes en las aplicaciones de escritorio. Este comportamiento es capturado por el meta-modelo que aquí se propone y, en términos sintácticos, se define a través de una relación de tipo `tooltip` entre un componente de UI que posea contexto de datos (un *contexto* o una *lista*) y un tipo especial de *detail* denominado *tooltip detail*:

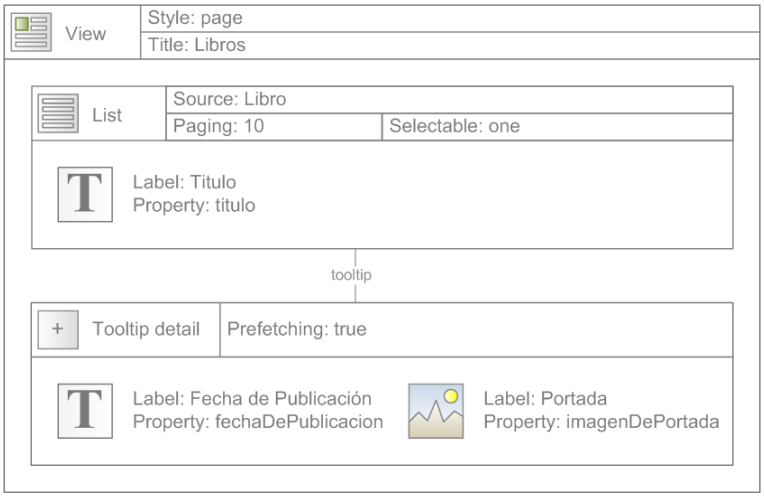


Figura 28. Utilización de un componente *detail* a modo de *tooltip*.

En la Figura 28 se muestra gráficamente la asociación entre una *lista* y un *detail* a modo de *tooltip*, para el ejemplo ya introducido en la Figura 27. La relación de tipo `tooltip` entre un *detail* y una *lista* tiene como consecuencia, en términos de comportamiento, que al dejar estático el cursor bajo un elemento de la esta última por unos segundos, se mostrará el grupo de componentes correspondiente al *detalle*. El contexto de datos que el *detail* puede referenciar en este tipo de relaciones es el mismo que el del componente compuesto al cual se asocia, y la semántica de la cláusula `Prefetching` es idéntica a la ya mencionada para *details* convencionales.

Información y fundamento del patrón de UI representado por los *tooltips* puede hallarse en la sección *Tooltips*.

6.6 Constructores

Hasta ahora se han definido los componentes de RIADL orientados a la obtención y visualización de los datos de la aplicación. En esta sub-sección se definirá un nuevo tipo de elementos del lenguaje denominados *constructors* o *constructores*, los cuales permitirán generar nuevas instancias de datos.

Al igual que los componentes orientados a datos ya mencionados, este tipo de componente posee un contexto de datos el cual, en este caso, referencia a una *entidad* para la cual se generará una nueva instancia, la cual es referenciada por una propiedad denominada `Entity`. Por cada *atributo* obligatorio de la *entidad* referenciada, deberá existir

un componente de UI compatible en términos de tipo de dato que genere su valor correspondiente.

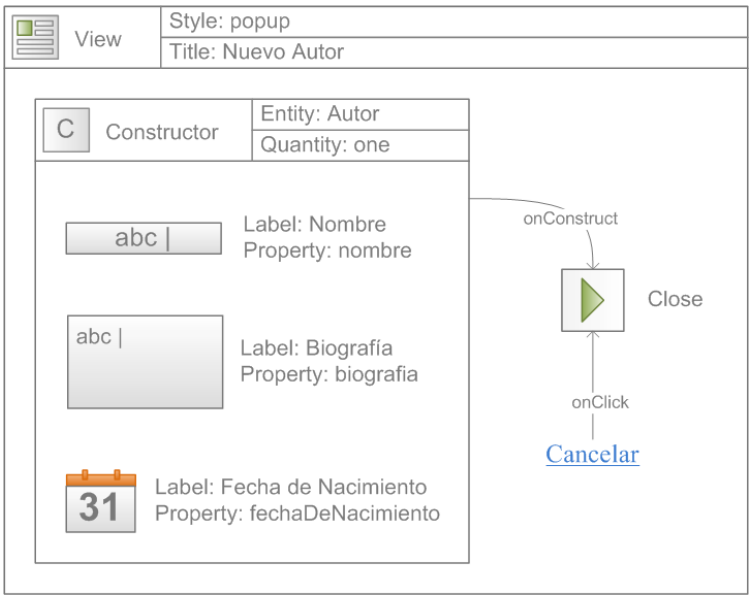


Figura 29. Ejemplo de uso de un *constructor*.

En la Figura 29 se ejemplifica el uso de un *constructor*. En particular, el *constructor* tiene como contexto de datos a la *entidad* `Autor`, por lo que su objetivo es generar una nueva instancia de datos para la misma. La propiedad `Quantity` determina cuántas instancias de datos pueden construirse con el mismo, y sus valores posibles son `one` (sólo se genera una instancia) o `many` (se pueden generar una o más instancias). En este último caso, las múltiples instancias son construidas en el client-side y es deseable que sean dadas de alta en una única petición al server-side, siguiendo la misma lógica que el patrón *Submission Throttling*[3] propone. En el caso particular del ejemplo de la Figura 29, como puede observarse, se creará una sola instancia de datos. En lo respectivo a interfaz de usuario, se utiliza un *textbox*[26] etiquetado `Nombre` para definir el valor del *atributo* `nombre` del `Autor`, un *date picker*[27] etiquetado `Fecha de nacimiento` para especificar el *atributo* `fechaDeNacimiento`, y un *textarea*[28] con label `Biografía` para ingresar el texto correspondiente al *atributo* `biografia`.

El modelo de ejemplo mostrado en la Figura 29, sin embargo, no es válido semánticamente, dado que existe una *relación* obligatoria (con cardinalidad mínima `1`) entre instancias de `Autor` y de `Libro`, la cual no es tratada en el *constructor*. Esta asociación será vista en secciones posteriores.

Implícitamente en la figura anterior se define un evento particular que un *constructor* puede invocar el cual es denominado `onConstruct`. El mismo ejecuta la acción que tiene asociada al generarse y darse de alta exitosamente la(s) instancia(s) de datos en cuestión. En el caso particular de la Figura 29, el *constructor* se muestra en una *vista* en modo *popup* la cual es cerrada al construirse efectivamente la instancia de `Autor`.

26 Componente de UI que permite el ingreso de texto.
27 Componente de UI que permite el ingreso de fechas asistido, por ejemplo, a través de calendarios interactivos.
28 Componente de UI que permite el ingreso de contenido textual que puede tener múltiples líneas.

6.6.1 Aspectos de implementación

En términos de comportamiento, cualquier validación especificada en relación a las *entidades* y sus *atributos* es llevada a cabo automáticamente al ingresarse los datos en los campos definidos dentro del *constructor* o intentar hacer efectiva el alta. El código que lleva a cabo estas validaciones es derivado de acuerdo a las restricciones de validez impuestas en términos de *entidades*, las cuales fueron introducidas al meta-modelo de datos en la sección Validación.

De ser `many` el valor de la propiedad `Quantity`, el *constructor* deberá permitir crear múltiples instancias de datos de la *entidad* que referencia. Esto puede lograrse sin llevar a cabo mayores modificaciones en la UI, enviando una petición al server-side la cual haga el alta y reusando luego los mismos componentes para efectuar una segunda instanciación, repitiéndose este comportamiento por cada instancia que se quiera generar. Sin embargo, explotando la capacidad de modificar la UI en tiempo real y sin intervención del servidor, puede permitírsele al usuario simplemente replicar los componentes de interfaz de usuario pertenecientes al *constructor* para que pueda especificar cuantas instancias desee, y luego efectuar el alta de las mismas de una sola vez.

El patrón de comportamiento descripto en el párrafo anterior, el cual fue identificado, descripto y catalogado anteriormente en este documento como *Merged Uploads* en la sección *Interactividad y performance en comunicación*, ofrece ventajas en términos de interactividad y de reducción de carga en el server-side. No obstante, el mismo requiere la generación de componentes de UI que permitan llevar a cabo las tareas que plantea. En primer lugar, deben generarse dos elementos de interfaz de usuario los cuales permitan generar réplicas de la interfaz de construcción de instancias y efectuar la petición que llevará a cabo las altas respectivamente. Luego, también es deseable que el usuario pueda eliminar las réplicas de la interfaz de construcción de instancias generadas si por arrepentimiento o error no quisiera efectuar un alta particular. Esto usualmente puede lograrse ofreciendo en cada interfaz de construcción particular un componente de UI que permita eliminarla.

6.7 Actualización de información

Habiendo definido conceptos del meta-modelo que permitan el observar y generar nuevas instancias de objetos de datos, aún no se han especificado en el mismo mecanismos que permitan la edición de *atributos* en aquellos ya existentes o su eliminación. Estos aspectos serán introducidos en esta sección, resaltándose antes las principales características de interactividad que las Rich Internet Applications proveen ante la necesidad de efectuar cambios en objetos de datos existentes.

6.7.1 Modificación y eliminación de datos en Aplicaciones Ricas

Como se mencionó con anterioridad en la sección *Edición interactiva*, la riqueza de interfaz de usuario que caracteriza a las RIAs permite llevar a cabo modificaciones en los objetos de datos que se observa sin necesidad de navegación o recarga de una nueva UI a este fin. Este comportamiento se logra, en primer lugar, explotando la capacidad de modificación de la UI directamente en el client-side, permitiéndole al usuario tanto observar

como modificar *atributos* de los objetos de datos mostrados, existiendo la posibilidad de cancelar dicha modificación sin que esto involucre interacción con el server-side. En segundo lugar, las facilidades para comunicación asincrónica con el servidor presente en las Aplicaciones Ricas permiten hacer efectivos los cambios transfiriendo sólo la información requerida y sin necesidad de navegación o carga completa de otra UI.

En términos de eliminación, las ventajas de interacción son similares. Los objetos a eliminar pueden seleccionarse directamente en el client-side, y eliminarse efectuando una sola petición al servidor. Asimismo, en la faceta cliente puede pedirse la confirmación al usuario antes de efectuar la baja de datos, así como la misma puede ser cancelada sin necesidad de interactuar con el server-side.

6.7.2 Modificaciones

Dada la asociación ya existente entre *atributos* de *entidades* y componentes de UI, puede utilizarse la misma con el objeto de especificar el modo en el cual puede modificarse la información relativa a instancias de datos particulares existentes en la aplicación. En términos concretos, se establece que cada componente orientado a la visualización de datos como, por ejemplo, un *text*, siempre que sea posible, poseerá una propiedad booleana asociada denominada `editable`, la cual de ser verdadera determinará que el mismo brinda posibilidades de edición de la propiedad la cual tiene asociada. Al momento de la derivación de código, de activarse esta propiedad, se generará en términos de implementación algún mecanismo que le permita al usuario activar la edición de contenido para ese componente de UI particular y, a su vez, que le brinde la posibilidad de aceptar o cancelar los cambios efectuados. Los cambios serán almacenados en el client-side y comunicados al servidor cuando sea necesario o el usuario lo requiera.

Resta aclarar que, como ya se mencionó, a fin de activar la edición de un elemento de información que hasta el momento era representado en un componente de UI de sólo lectura, es necesario modificar la interfaz de usuario con el objeto de reemplazar dicho componente por uno que permita el ingreso de datos. El componente de UI reemplazante puede no ser único; por ejemplo, una fecha puede ser editada tanto a través de un campo de texto como por un componente de tipo calendario o *date picker*. En los casos en que pueda elegirse uno entre varios componente de UI que hagan posible la edición, el mismo será especificado a través de un atributo particular en el widget de sólo lectura.

En lo respectivo a modificaciones en el modelo de datos que pueden llevarse a cabo a través de la UI, lo mencionado hasta el momento cubre aquellas relativas a los *atributos* particulares de un objeto de datos determinado. Las modificaciones que impliquen el crear o destruir *relaciones* entre instancias de datos serán vistas en la sección posterior.

6.7.3 Eliminación

La eliminación de elementos de datos en términos de UI, a diferencia de la modificación de *atributos* particulares, es una acción que es llevada a cabo sobre objetos de datos en sí y no sobre alguna de sus propiedades. Por este motivo, y dado que las instancias de datos son referenciadas por *contextos* y *listas*, son estos componentes los que proveerán la capacidad de eliminación.

Los *contextos* conllevarán una propiedad booleana asociada denominada `allowDeleting`, la cual establecerá la posibilidad de eliminar el elemento actual al cual hacen referencia. De igual manera, una *lista* permitirá la eliminación de elementos a través de una propiedad definida como `deleteMode`, la cual podrá cobrar tres valores posibles:

- `none`, caso en el cual el borrado de elementos no se habilitará,
- `selected`, la cual indicará que se proveerá un método para eliminar aquellos elementos de la *lista* que hayan sido seleccionados, o
- `perElement`, la cual establecerá que por cada elemento de datos se creará un componente de UI el cual permitirá su eliminación, siguiendo el pattern *In-Context Tool*[14].

En términos de derivación, tanto en los *contexts* como en las *listas* deberán generarse los componentes de UI necesarios que permitan llevar a cabo la eliminación.

6.8 Navegación entre instancias de datos

Los componentes de UI orientados a datos que han sido definidos en RIADL hasta el momento son capaces de mostrar información relativa a una instancia de datos de una *entidad* particular, pero no de referenciar datos relacionados a la misma. Un ejemplo particular de lo problemático de esta limitación es la incapacidad para referenciar al `Autor` de un `Libro` particular en el *contexto* mostrado en la Figura 23. En esta sección se introducirán aspectos del meta-modelo que permitirán la navegación entre instancias de datos[29] a través de *relaciones* existentes entre las mismas. Estos aspectos permitirán que los componentes de UI orientados a datos puedan obtener su contenido a mostrar no sólo desde la instancia de la *entidad* particular al que su contexto de datos referencia, sino también desde instancias relacionadas a ésta.

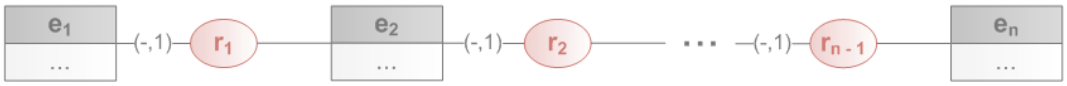
6.8.1 Navegación entre instancias de datos relacionadas por cardinalidad máxima 1

Actualmente, los componentes de UI simples orientados a datos (aquellos que son capaces de mostrar una propiedad particular de una instancia de datos como, por ejemplo, *texts* o *images*) son asociados a un *atributo* específico de la *entidad* que su contexto de datos referencia, lo cual los limita a mostrar datos relativos a propiedades de instancias de la misma. Sin embargo, de existir una *relación* con cardinalidad máxima 1 entre una instancia de la *entidad* referenciada por el contexto de datos del componente y otra instancia particular, podría navegarse desde la primera hacia la segunda y utilizar uno de sus *atributos* como origen de datos del susodicho componente. Por ejemplo, en el *contexto* definido en la Figura 23, podría navegarse la *relación* `autorLibro` (ver Figura 17) a fin de mostrar el *atributo* `nombre` de la instancia de `Autor` relacionada con el `Libro` particular que en el mismo se referencia.

El mismo razonamiento planteado en el párrafo anterior puede aplicarse inductivamente para una sucesión de instancias de *entidades* conectadas a través de

29 El concepto de navegación que se utiliza en esta sección no está relacionado con el uso tradicional que se le da en este documento, el cual está orientado a interfaces de usuario e hipertexto, sino que hace alusión a la obtención de instancias de datos recorriendo *relaciones* establecidas en el modelo de datos ya comentado en secciones previas.

relaciones cuya cardinalidad máxima sea 1 en el sentido en que son navegadas. En general, de existir una sucesión de *entidades* y *relaciones* de la forma



, siendo 1 la cardinalidad máxima de las *relaciones* en el sentido de navegación desde e_1 hasta e_n, es posible que un elemento de UI simple cuyo contexto de datos referencia a la *entidad* e_1 sea capaz de navegar hacia la *entidad* e_n y mostrar el contenido de alguno de sus *atributos*.

En el meta-modelo que aquí se propone, la navegación entre instancias de datos estará dada por una nueva propiedad que componentes de UI orientados a datos poseerán, la cual será denominada `Source`. La misma tendrá contenido nulo cuando se muestren datos de instancias de la *entidad* que el contexto de datos del elemento de UI simple directamente referencia. De especificarse cierta navegación entre instancias de datos a través de *relaciones* a fin de alcanzar la instancia deseada, la propiedad `Source` contendrá la secuencia de *relaciones* de cardinalidad 1 en el sentido de navegación que deberán atravesarse para alcanzar la instancia particular. La sintaxis que se utilizará aquí, la cual será expresada luego con elementos del lenguaje de meta-modelado que se elija en secciones posteriores, se corresponde con el patrón

$$(r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow ... \rightarrow r_n)\, e \quad ,$$

donde r_1 a r_n representan las *relaciones* sucesivas que se van atravesando a partir de la instancia del contexto de datos del componente a fin de alcanzar la instancia final, cuya *entidad* asociada es *e*.

Según lo establecido en el párrafo anterior, si en el ejemplo de la Figura 23 se quisiera hacer referencia al único `Autor` del libro (relacionado a través de la *relación* `autorLibro`, cuya cardinalidad máxima navegando desde `Autor` hacia `Libro` es 1), la misma se podría alcanzar con el modelo ejemplificado en la Figura 30 (a).

Los *contextos*, aunque a diferencia de los anteriores son elementos de UI compuestos orientados a datos, tienen como origen de datos a una única instancia de una *entidad*. Por ende, puede utilizarse en ellos el mismo razonamiento planteado hasta el momento y definir una propiedad `Source` en los mismos que permita especificar navegación entre instancias asociadas por *relaciones* de cardinalidad máxima 1 en el sentido en que son navegadas. Esto último es ejemplificado en la Figura 30 (b).

Es conveniente aclarar que la semántica del origen de datos de los dos *contexts* utilizados en la Figura 30 (b) es diferente en uno que en otro. Mientras que el primero tomará los datos desde un origen que aún no ha sido definido en este documento, el segundo tiene como origen de datos la instancia a la cual referencia su *contexto* padre, a partir de la cual realiza una navegación a través de una *relación* con cardinalidad máxima 1 en el sentido en el que es navegada alcanzando a una instancia de la *entidad* `Autor`.

Finalmente, aún no se ha mencionado qué ocurre si una o más de las *relaciones* que unen la secuencia de instancias a través de las cuales se navega es 0 y, por ende, la navegación se ve interrumpida al no existir una instancia relacionada en algún punto de la

secuencia. En este caso, en términos de UI, puede simplemente optarse por mostrar un mensaje aclarando esta cuestión en lugar de componente de UI que lleva a cabo infrucructuosamente la navegación a fin de obtener los datos a mostrar.

Figura 30. Uso de navegación de datos para mostrar información de instancias relacionadas a la referenciada por un *contexto* particular.

6.8.2 Navegación entre instancias de datos relacionadas por cardinalidad máxima *n*

Entre instancias de datos asociadas por *relaciones* de cardinalidad máxima 1, la navegación es *determinística*: solo puede navegarse desde una instancia particular hacia su asociada. Al tenerse *relaciones* de cardinalidad máxima mayor a 1, este *determinismo* deja de existir, dado que existe una opción de navegación particular por cada instancia de datos asociada a la instancia de origen. Desde un punto de vista funcional, mientras que la navegación a través de una *relación* con cardinalidad máxima 1 tiene como dominio e imagen instancias particulares, la navegación a través de *relaciones* con cardinalidad máxima n tiene como dominio una instancia particular y como imagen un conjunto de instancias asociadas a la primera. Teniendo en cuenta esto último, no es posible que componentes cuyo contexto de datos se refiera a una única instancia efectúen este tipo de navegaciones a fin de obtener la información que requieren. Sin embargo, las *listas*, componentes del lenguaje que ya fueron introducidos con anterioridad, se adaptan perfectamente a esta situación, dado que están orientadas a mostrar una colección de instancias de datos pertenecientes a una *entidad* particular. Por ende, es posible que componentes de tipo *lista* efectúen navegaciones a través de *relaciones* con cardinalidades mayores a 1 para obtener la información que requieren.

Un ejemplo de lo dicho anteriormente aplicado al modelo de datos de la Figura 17 es la posibilidad de navegar desde una instancia de `Autor` particular a su conjunto de instancias de `Libro`s escritos. La obtención de estos especificando la navegación necesaria a través de la propiedad `Source` ya fue introducida con anterioridad. En la misma, se especifica un *contexto* el cual hace referencia a un `Autor` particular y, dentro del mismo, una *lista* que muestra instancias de `Libro` las cuales obtiene navegando la *relación* `autorLibro` a partir de la instancia de `Autor` referenciada por el *contexto* que la incluye.

Se hace notar que la semántica de obtención de datos de la *lista* en este caso es diferente a la vista hasta el momento. Mientras que en los ejemplos anteriores las *listas* han tenido como origen de datos a todas las instancias de una *entidad* particular, eventualmente realizando algún filtrado a través de *filters*, en este caso se obtiene sólo aquellas que estén asociadas a través de la *relación* o secuencia de *relaciones* que en la propiedad `Source` se especifique.

Figura 31. Ejemplo de navegación a través de *relaciones* con cardinalidad máxima *n*

En general, la propiedad `Source` para *listas* o cualquier otro componente de UI capaz de mostrar un conjunto de instancias de una *entidad* particular puede tener la forma:

$$(r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow ... \rightarrow r_n)\, e$$

donde r_i para i entre *1* y *n-1* es una *relación* cuya cardinalidad máxima en el sentido de navegación es 1, y r_n es una *relación* cuya cardinalidad máxima hacia la *entidad e* es mayor a 1.

6.9 Origen de datos

En la sección anterior se menciona que, para el caso particular de las *listas*, existen dos maneras distintas en que las mismas pueden especificar su origen de datos: declarando una *entidad* de la cual obtendrán todas sus instancias, o bien describiendo una secuencia de una o más *relaciones* que deberán atravesarse partiendo de la instancia que el contexto de datos del componente en el que están incluidas referencia. Más allá de este caso particular, aún no se ha descripto de manera general los métodos a través de los cuales los

componentes de UI pueden obtener las instancias de datos que requieren y relacionarse entre sí en términos de información. En esta sección, se definirán aquellos aspectos de RIADL relativos a la especificación de *orígenes de datos* para los componentes de interfaz de usuario.

En el lenguaje propuesto, se definen 3 modos a través de los cuales los componentes de UI pueden obtener los datos que requieren:

- *Independiente*: el componente obtiene los datos que requiere por sí mismo, especificando una *entidad* de la cual obtendrá todas sus instancias. Este método es utilizado por las *listas*, como se las ha introducido hasta el momento.
- *Contextual*: el componente de UI utiliza el contexto de datos definido por el componente compuesto que lo contiene para obtener la información que requiere. Opcionalmente, puede utilizar la instancia de datos que en dicho contexto se referencia como punto de partida de una acción de navegación que atraviese una o más *relaciones*, hasta alcanzar el elemento de datos deseado, haciendo uso de la cláusula `Source`. Este modo de obtención fue utilizado implícitamente en la mayoría de los ejemplos hasta el momento: lo utilizan los componentes de UI simples como *texts* e *images*, así como también aquellos compuestos como los *detalles*.
- *Asociativo*: el componente obtiene los datos que requiere desde otro componente definido en la UI, pudiendo opcionalmente efectuar navegación entre instancias, como en el caso de la obtención *contextual*. Los elementos que utilizan este método definen, en términos de sintaxis concreta, un elemento interno denominado `dataSource`, el cual funciona como receptor de datos provenientes desde otros elementos de la UI.

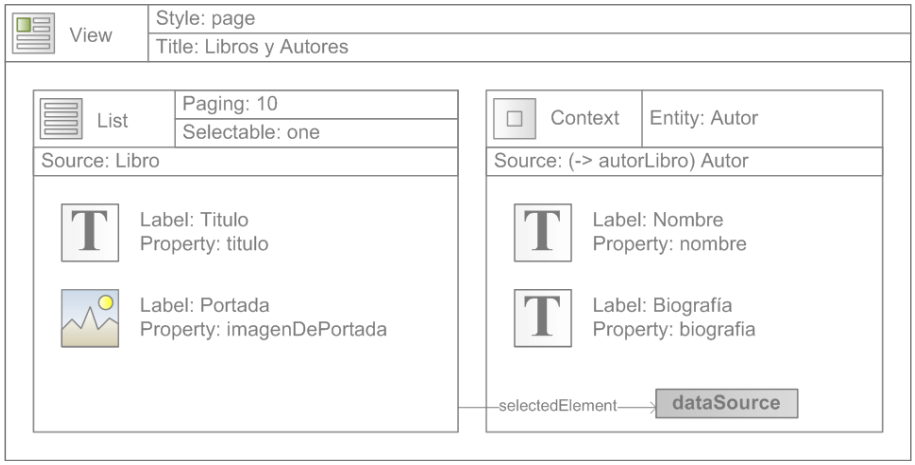


Figura 32. Uso de la propiedad `dataSource` para indicar la obtención de datos desde un componente particular de la UI.

La Figura 32 muestra un modelo de ejemplo que utiliza modo de obtención de datos *asociativo*: el elemento seleccionado en una *lista* (una instancia de `Libro`) es aportado como origen de datos a un *contexto*, el cual obtiene la instancia de `Autor` que requiere navegando

a través de la *relación* `autorLibro`, a partir de la instancia de `Libro` que la *lista* le provee. Semánticamente, en términos de comportamiento de la UI derivada, el seleccionar un elemento de la *lista* provocará la consecuente actualización del *contexto*, dado el cambio que se produce en su origen de datos. Esta actualización se trasladará a todos sus componentes internos, debido a que los mismos obtienen sus datos contextualmente.

A continuación, se resumen en la Tabla 1 los modos de obtención de datos que pueden utilizar los componentes del meta-modelo introducidos hasta el momento.

	Contextual	Asociativo	Independiente
Lista	X	X	X
Contexto, Detalle	X	X	
Componentes simples	X		

Tabla 1. Modos de obtención soportados por los distintos componentes de UI definidos en el meta-modelo

Resta aclarar que la posibilidad de utilizar navegación entre datos a través de la propiedad `Source` en un componente sólo puede efectuarse si los mismos son obtenidos contextual o asociativamente. En caso de que el componente especifique su origen de información de manera independiente se hace imposible la navegación, dado que no se dispone de una única instancia como punto de partida de la misma.

En la Figura 33 se muestra un ejemplo real de la semántica implicada por el modelo de la Figura 32, el cual se halla implementado en la interfaz de administración de contactos de GMail.

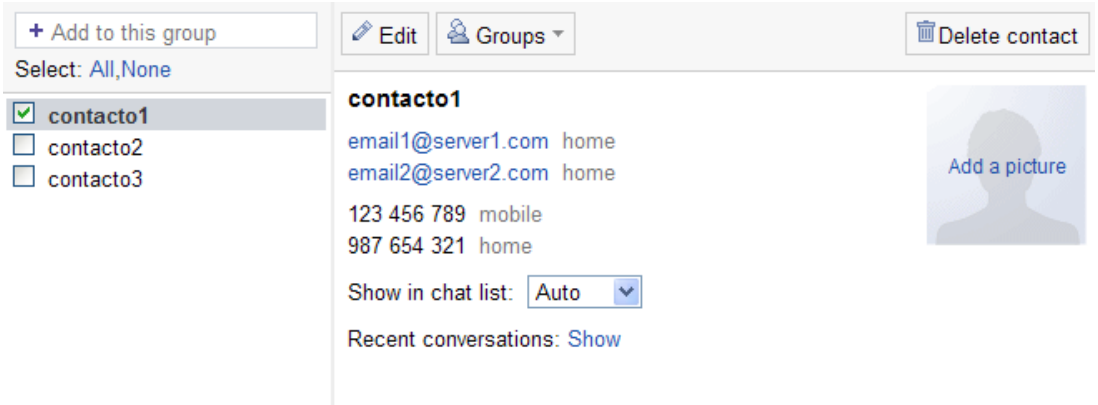


Figura 33. Acceso a datos detallados de un contacto particular en la interfaz de administración de contactos de GMail. El hacer clic en un elemento en la lista de la izquierda actualiza los datos presentes en la fracción derecha de la UI.

6.9.1 Consideraciones de performance en orígenes de datos

Cuando se realizan peticiones a la capa de servicios, existe un *tradeoff* entre la

magnitud de datos transferidos al momento de la carga inicial de la Aplicación Rica y la necesidad de peticiones futuras para obtener datos que se requieren y no están disponibles en el client-side. En el caso planteado en la Figura 32, al obtener las instancias de `Libro` que la UI requiere, puede indicarse a la capa de servicios, como ya fue planteado con anterioridad, que en la misma petición se incluya o no la información relativa a los `Autor`es asociados a través de la *relación* `autorLibro`. Si se eligiera incluir esta información, el volumen inicial de datos transferido podría llegar a tener una magnitud considerable, dado que acarrearía los datos respectivos a `Libro`s y sus `Autore`s relacionados. Sin embargo, esta inclusión tendría también una consecuencia favorable: si se seleccionara un `Libro` distinto en la *lista* correspondiente, el *contexto* no tendría que efectuar una petición al server-side para obtener los datos del `Autor` asociado al mismo, dado que esa información ya estaría cacheada en el client-side. Por otro lado, el no incluir la información relativa a `Autor`es en la petición inicial efectuada al inicializar la *vista* ahorraría ancho de banda y memoria en el client-side, pero tendría como consecuencia que un cambio en el `Libro` seleccionado en la *lista* correspondiente conllevaría una nueva petición al server-side para obtener la información de `Autor` requerida.

La elección del tipo de carga de datos que se efectuará según lo descripto en el párrafo anterior dependerá del tiempo de respuesta e interacción que se busque alcanzar, la diferencia en términos de magnitud de datos transferidos entre ambas variantes, cuán probable es que el usuario consulte los datos asociados, etcétera. Dado que estas características dependen del dominio particular de la aplicación, en el meta-modelo que aquí se propone, el momento de la carga de datos asociados puede ser personalizado por el modelador cuando esto sea semántica y tecnológicamente posible.

En términos semánticos, la pre-carga es posible siempre y cuando exista cierta división en los datos que permita que los mismos puedan ser transferidos en dos o más etapas. Un caso particular de esta división es la que existe entre instancias de datos relacionadas, las cuales pueden ser cargadas en distintos momentos, como se menciona anteriormente. Esto sugiere que, en los casos en los cuales se aplique navegación entre datos a fin de determinar el origen de la información de algún componente y el contenido resultante de la misma no sea indispensable para la inicialización de la UI, puede optarse entre efectuar o no la pre-carga de los datos dependiendo del comportamiento deseado. Esto será plasmado en RIADL a través de una propiedad booleana denominada `Prefetching`, la cual está asociada a la navegación entre datos efectuada a través de la cláusula `Source`. De ser verdadera, la misma indica que se llevará a cabo la pre-carga de los datos asociados a navegar, lo cual tendrá las consecuentes ventajas y desventajas anteriormente mencionadas. Por otro lado, si la susodicha propiedad fuera falsa, la carga de los datos por navegar será llevada a cabo bajo demanda cuando los mismos se requieran.

Como ya se dijo, aunque se utilice navegación entre instancias de datos para determinar el origen de la información de algún componente, la pre-carga de la información a navegar tiene sentido sólo cuando la misma no es indispensable para la construcción inicial de la UI. El no utilizar pre-carga de datos en esta circunstancia obliga a llevar a cabo más de una petición al server-side para transferir datos que son requeridos para construir la interfaz y comenzar la interacción con el usuario, los cuales podrían ser cargados en la misma petición inicial, indicando las cláusulas de inclusión pertinentes al server-side. Por ejemplo, cuando se tiene un *contexto* que se asocia a una instancia de datos particular y,

dentro del mismo, una *lista* que la referencia contextualmente y navega hacia otras asociadas para determinar su contenido, la pre-carga establece que se obtendrán en una misma petición las instancias que tanto el *contexto* como la *lista* requieren y, de no llevarse a cabo, dos o más peticiones serían necesarias a la faceta servidor a fin de transferir la misma información. Sin embargo, el origen de datos de modo contextual no siempre implica la imposibilidad de habilitar la pre-carga. En *detalles*, por ejemplo, aunque se dé la obtención de datos contextualmente, dado que la construcción de su contenido puede hacerse bajo demanda y puede ser posterior a la construcción inicial de la interfaz de usuario, la posibilidad de especificar o no la carga anticipada es semánticamente correcta. Un ejemplo de esto puede observar en la Figura 34.

Es pertinente destacar a partir del mencionado ejemplo de la Figura 34, que la semántica de la propiedad `Prefetching`, antes destinada únicamente a los *detalles*, es ahora generalizada a todo componente compuesto del meta-modelo capaz de efectuar acciones de navegación.

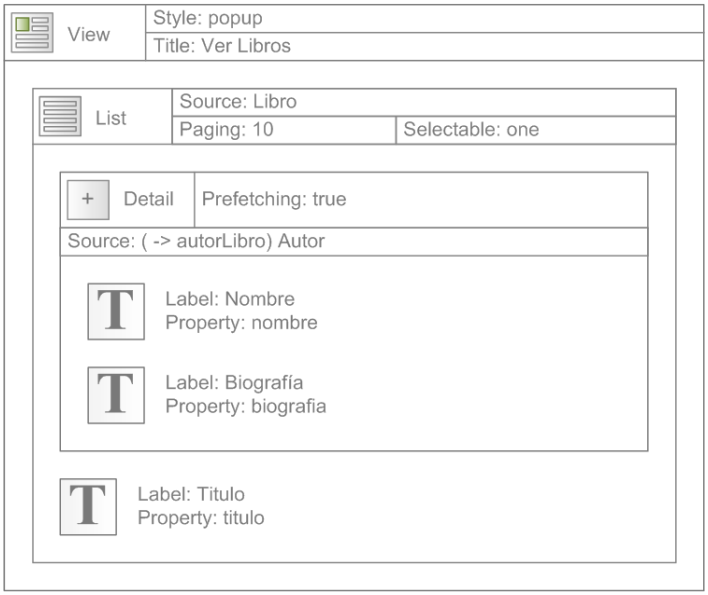


Figura 34. Uso de la cláusula `Prefetching` en un *detail*. A pesar de que el *detail* obtiene su información contextualmente (y luego efectúa una navegación de datos a partir de la instancia obtenida), el personalizar la pre-carga tiene sentido, dado que por la semántica particular del componente, los datos son requeridos cuando el usuario desee observarlos.

La cláusula `Prefetching` unifica y generaliza la manera en la cual se puede especificar la pre-carga de datos en componentes de UI compuestos. En la Figura 35 puede observarse un ejemplo del uso de la misma, en relación a una *lista* de `Libro`s que obtiene sus elementos partiendo de un `Autor` seleccionado en una *lista* asociada y efectuando una navegación de datos a través de la *relación* `autorLibro`. Al estar habilitada la propiedad de `Prefetching`, la carga de `Autor`es implica también implícitamente la carga de sus `Libro`s asociados, información la cual se obtendrá en una misma petición al server-side y se mantendrá almacenada en la faceta cliente. Como consecuencia, un cambio en el elemento actualmente seleccionado en la *lista* de `Autor`es no requerirá una petición al server-side con el objeto de cargar sus `Libro`s relacionados.

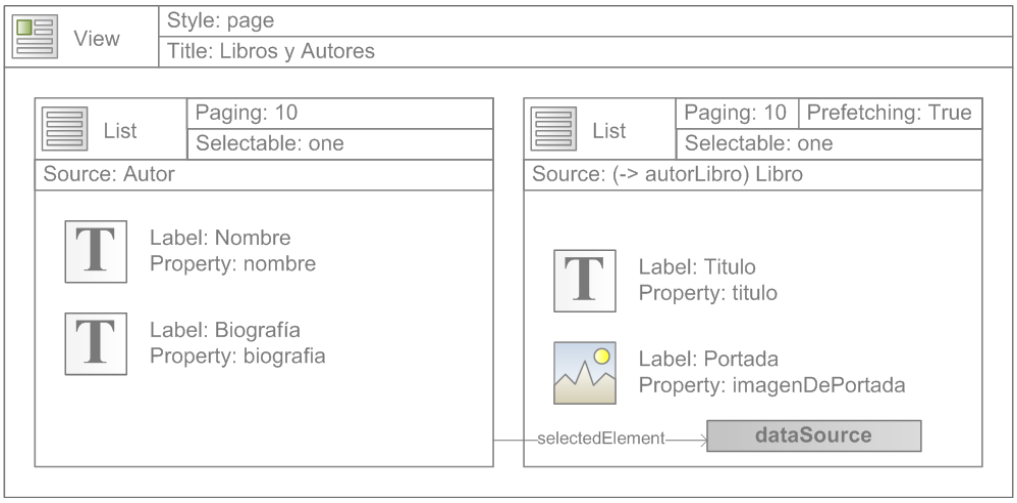


Figura 35. Uso de la cláusula `Prefetching` para habilitar la pre-carga de datos relacionados

La Figura 36 muestra un ejemplo de la interfaz de manejo de grupo de contactos de GMail, el cual podría ser especificado y luego derivado a partir de un modelo estructuralmente similar al descripto en la Figura 35.

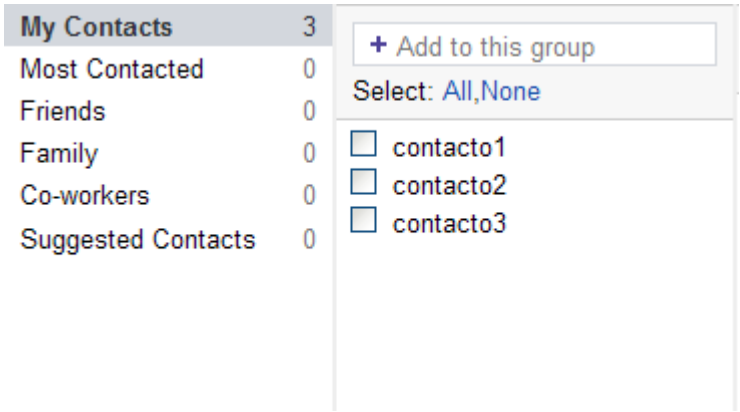


Figura 36. Acceso a contactos relativos a un grupo particular utilizando dos listas. El contenido de la lista de la derecha depende del elemento particular seleccionado en la lista de la izquierda (GMail)

6.9.2 Prefetching sin navegación

A pesar de que la cláusula `Prefetching` es introducida en el contexto de la obtención de datos relacionados a instancias particulares, en determinadas situaciones la cláusula puede ser igualmente aplicada sin estar relacionada necesariamente con el concepto de navegación de datos, aunque con diferente semántica.

En el caso particular de componentes cuyo origen de datos puede ser independiente (por ejemplo, las *listas*), la semántica de la propiedad `Prefetching` establecida a `true`

indicará que se pre-cargarán todos los elementos que serán obtenidos, de manera de no tener que efectuarse peticiones futuras la server-side para obtenerlos luego de inicializada por completo la UI. La Figura 37 muestra un ejemplo de este tipo de uso particular de la cláusula en un componente *lista*.

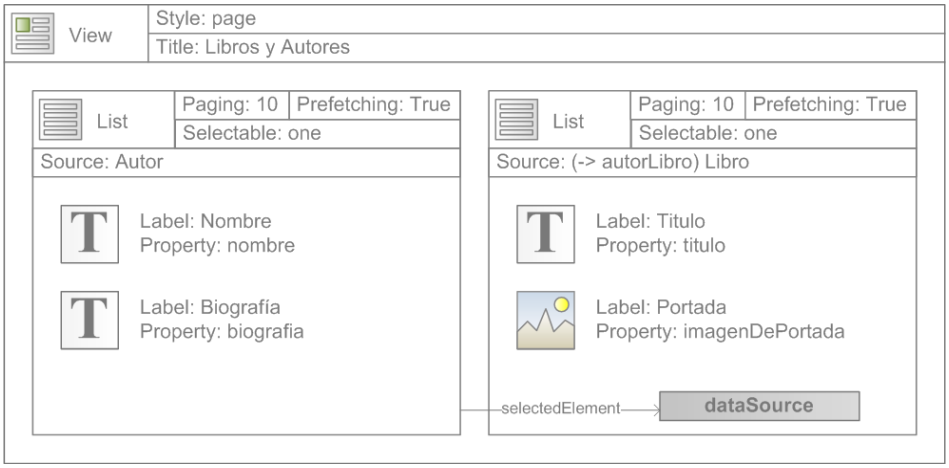


Figura 37. Uso de la cláusula `Prefetching` en un componente de UI orientado a datos capaz de obtener los mismos independientemente

6.10 Creación y destrucción de relaciones entre datos

De la misma manera en la que las *relaciones* entre objetos de datos pueden establecerse al momento de su instanciación, es indispensable que exista la posibilidad de crearlas y destruirlas sobre instancias de información ya existentes de acuerdo a las necesidades del dominio de datos y negocios de la aplicación.

Habiéndose introducido mecanismos a través de los cuales un componente puede aportar datos a otro elemento del lenguaje con un fin determinado, una técnica similar puede utilizarse para asociar o desasociar instancias de datos en el contexto de una *relación* particular. En términos concretos, RIADL introduce con este objeto dos tipos de *acciones* particulares:

- *Asociación*, la cual debe especificar una *entidad* y una *relación*, y debe recibir una instancia de la *entidad* en cuestión y otra de la *entidad* que se quiere asociar a la primera. El resultado de la ejecución de la misma consiste en construir una asociación entre ambas instancias de datos en el contexto de la *relación* particular
- *Disociación*, la cual recibe los mismos parámetros que la anterior y tiene como resultado de su ejecución la disociación de las instancias particulares en términos de la *relación* especificada.

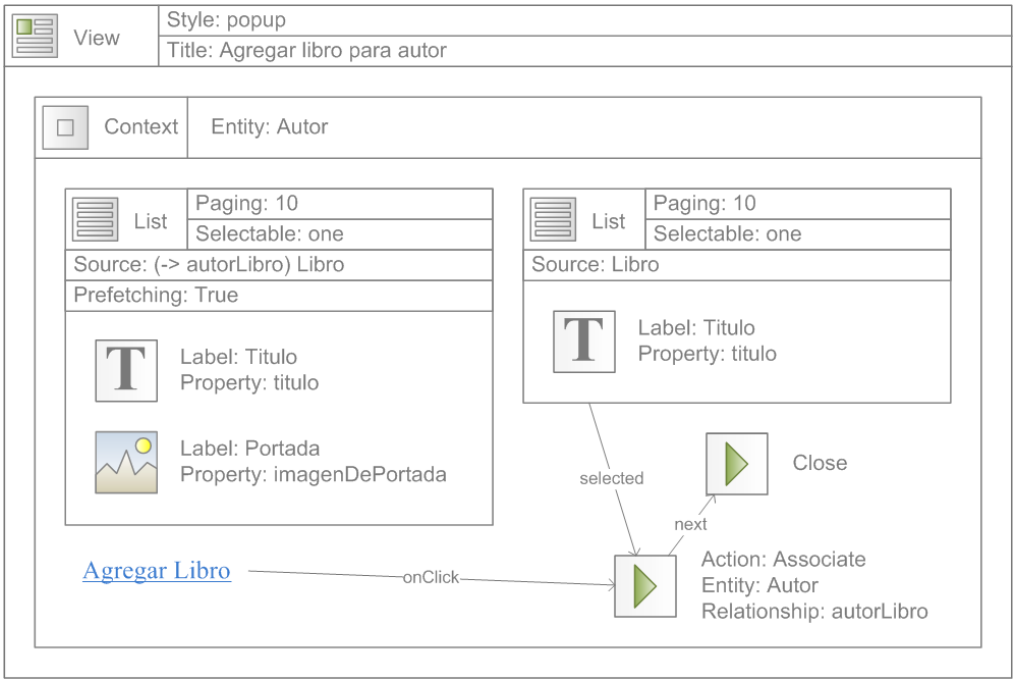


Figura 38. Uso de una *acción de asociación* para relacionar instancias de `Autor` con instancias de `Libro`.

La Figura 38 muestra un ejemplo del uso de una acción de asociación que tiene como objeto crear un vínculo entre una instancia de `Autor` y otra de `Libro`, en el contexto de la *relación* `autorLibro`. La instancia de `Autor` se obtiene desde el *contexto* que engloba a los otros componentes de la UI, mientras que el `Libro` particular es determinado por el elemento seleccionado de una *lista* aparte. Luego de construir y dar de alta la *relación*, puede ser deseable ejecutar acciones adicionales como, por ejemplo, ocultar la *vista* actual, navegar hacia otra página, entre otras. Con este objeto, las acciones de asociación y disociación son definidas como *no terminales*, en el sentido de que pueden especificar opcionalmente una cadena de una o más acciones a ejecutarse luego de la tarea que tienen asignada. Esta cadena es formada a partir de asociaciones de tipo `next`, cuya sintaxis concreta es es introducida en la Figura 38. Esta asociación puede partir desde cualquier acción que sea definida como *no terminal* y puede tener como destino a cualquier otra acción sea la misma *terminal* o no.

6.10.1 Constructores

El ejemplo de uso de *constructors* mencionado en la Figura 29, como se dijo en la sección correspondiente, no es semánticamente válido, dado que la creación de un nuevo `Autor` implica no sólo la definición de sus *atributos* obligatorios, sino también la especificación de uno o más instancias de `Libro` asociadas al mismo. Al igual que como se observó en las sub-secciones anteriores, esas instancias pueden ser seleccionadas desde una *lista* o componente similar que sea capaz de proveer una o más. Una vez seleccionadas, dicha(s) instancia(s) puede(n) proveerse como parámetro a un *constructor* que la(s) requiera (en este caso, un *constructor* de instancias de `Autor`), indicando la *relación* a través de la cual se asociarán a la instancia de datos a construir.

Para llevar a cabo lo mencionado en el párrafo anterior, el meta-modelo introduce

una nueva cláusula denominada `associatedBy`, la cual especifica la *relación* a través del cual las instancias de datos obtenidas serán asociadas con el objeto siendo creado por el *constructor*.

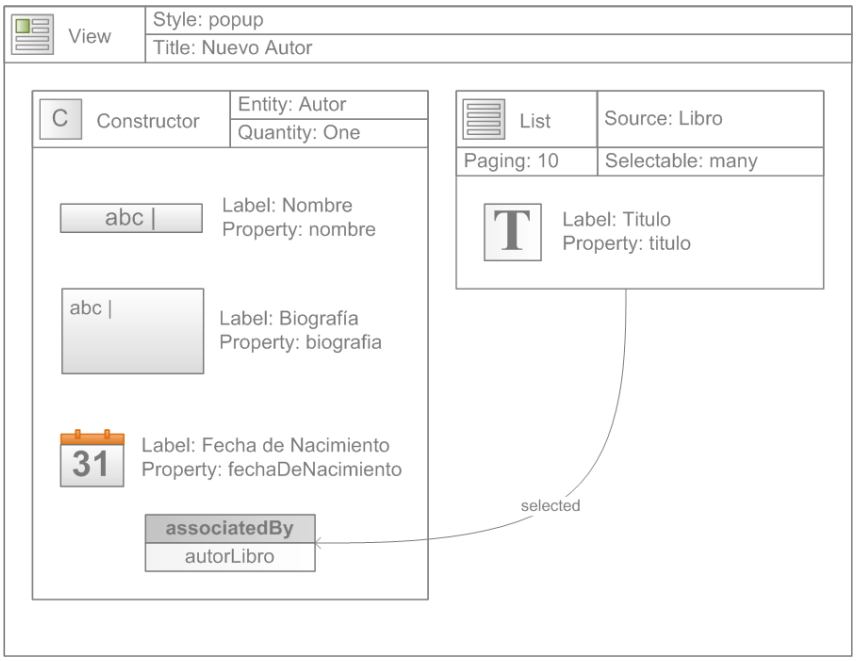


Figura 39. Uso de la cláusula `associatedBy` para especificar objetos relacionados a la instancia a construir.

En la Figura 39 se ejemplifica el uso de la cláusula `associatedBy`, utilizándola para especificar las instancias de `Libro` que serán asociadas al `Autor` en construcción. Siendo la cardinalidad mínima `1`, antes de efectuar el alta invocando el servicio correspondiente, la aplicación controlará que al menos una instancia de `Libro` esté seleccionada para ser relacionada.

6.11 Comunicación entre vistas

Hasta el momento, todos los elementos de RIADL que han sido introducidos se encuentran en el contexto de una única *vista*. Si bien, dado que los componentes definidos están pensados para dotar de riqueza a la aplicación, los mismos han permitido abstraer ciertos casos o patrones de interacción usuales, muchas veces se desea aislar o modularizar la misma en dos o más *vistas*, lo cual requiere la definición de un método de comunicación entre éstas.

Un primer enfoque para comunicar *vistas* entre sí, de igual manera que ocurre entre componentes pertenecientes a una misma *vista*, es asociar directamente dichos componentes como ya fue mencionado con anterioridad, por ejemplo, a través de la relación `selected`. Sin embargo, en un contexto en el cual los componentes relacionados están separados en distintas *vistas*, puede darse el caso de que más de una *vista* necesite navegar o mostrar otra particular la cual requiere información extra. Bajo este enfoque, deberá existir una conexión de datos explícita entre el componente de la *vista* origen y el correspondiente en la *vista* destino. En consecuencia, se corre con la desventaja de que las *vistas* están acopladas

fuertemente, dado que necesitan conocer su estructura interna para interactuar.

Para desacoplar las *vistas* interactuantes y de este modo obtener mejor modularización, se definen *parámetros* que las *vistas* pueden especificar como requeridos para poder ser mostradas o navegadas correctamente. Los *parámetros*, representados en el lenguaje por componentes denominados `Parameter` que pueden ser incluidos sólo dentro de componentes de tipo *vista*, representan una *puerta de acceso* a instancias de datos las cuales toda *vista* externa deberá proveer si desea hacer visible o navegar hacia aquella que los declara. Esto permite aislar y modularizar la definición de las *vistas*, abstrayéndose de la estructura interna que cada una define. Un *parámetro* debe establecer tres atributos básicos:

- `Entity`, el cual explicita la *entidad* de la cual el/los objeto(s) provistos debe(n) ser instancia.
- `Mandatory`, la cual es una propiedad booleana que indica si el parámetro es obligatorio u opcional.
- `Quantity`, cuyo valor indica si el parámetro requiere que se provea una sola instancia (`1`) de un objeto de datos o una colección (`many`).

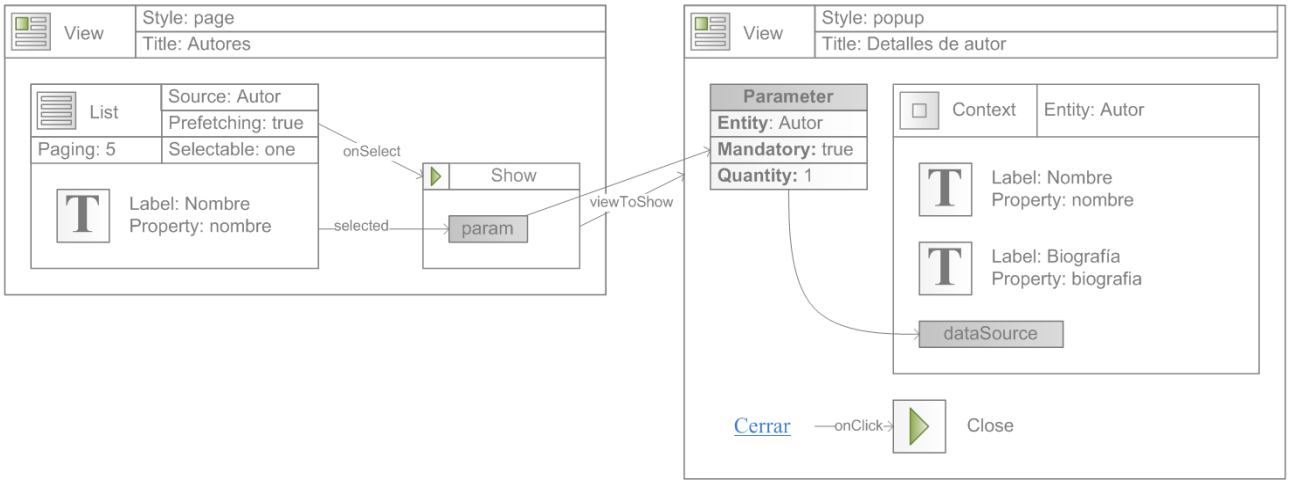


Figura 40. Uso de *parámetros* para especificar los objetos de datos externos que una *vista* requiere y el modo en que los mismos son asignados desde *vistas* externas

La Figura 40 describe un ejemplo de la utilización de *parámetros* en *vistas*. En la misma se modela una *vista* la cual contiene una *lista* de `Autor`es. Al ser seleccionado un `Autor` particular (evento `onSelect`) en esta última, se muestra una segunda *vista* en modo `popup`, detallando los datos del mismo en un elemento *context*. La *vista* `Detalle de Autor` podrá ser reusada en la especificación y un número ilimitado de *vistas* podrán mostrarla, siempre y cuando provean el parámetro de `Autor` correspondiente.

De manera implícita en la Figura 40, se enriquece la acción `Show` ya mencionada, volviéndola compuesta. Dentro de la misma ahora pueden definirse uno o más elementos `param`, los cuales tienen como fin relacionar un *parámetro* definido en una *vista* particular con el objeto de datos que se proveerá para el mismo. Si bien podría asociarse directamente el origen del objeto de datos que se provee con el elemento `Parameter`, esto posee una limitación: si se tuvieran dos o más acciones que muestran la misma *vista*, no podría en cada

una especificarse una fuente de datos particular, sino que la fuente debería ser obligatoriamente la misma. Esto ocurre debido a que la relación entre el elemento `Parameter` y el objeto de datos que se le provee no involucra a la acción `Show` particular. En la Figura 41, se describe un modelo de ejemplo esquemático que grafica este problema. La versión enriquecida de `Show` mostrada en la Figura 40 permite, a través de la definición de elementos `param`, que la acción explicite directamente el origen y parámetro destino de las instancia de datos requeridas para navegar o hacer visible la *vista* requerida.

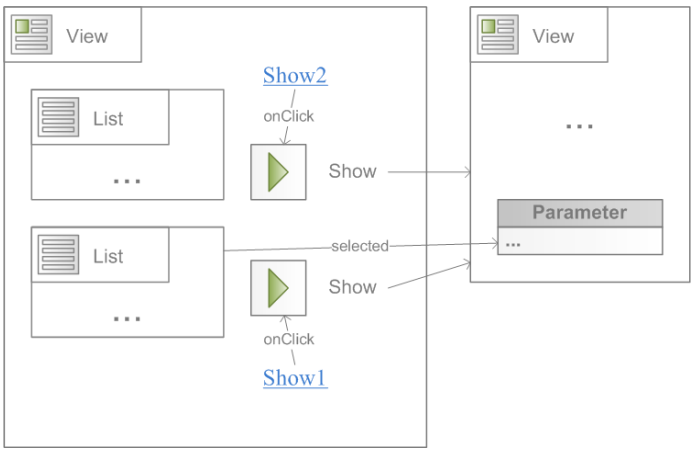


Figura 41. Ejemplo de la limitación inherente a relacionar directamente el origen del objeto de datos con el elemento `Parameter`: ambas acciones `Show` están obligadas a proveer el mismo objeto proveniente de la *lista* inferior, sin poder elegir independientemente cada una el *origen de datos* a aportar al *parámetro* obligatorio.

6.12 Consecuencias

En esta sub-sección se analizarán las consecuencias y ventajas que provee la especificación de Aplicaciones Ricas orientadas a datos bajo el meta-modelo propuesto.

6.12.1 Portabilidad de UI

El abstraer la interfaz de usuario en componentes de uso común y no ligados a ninguna plataforma de ejecución particular, hace posible el portar la UI especificada a distintas tecnologías actuales a través de la modificación o creación de nuevos generadores de código, sin que deban cambiarse los modelos que especifican el software a derivar. Bajo la plataforma de ejecución DHTML/Ajax, es particularmente difícil proveer esta portabilidad, debido a que distintos *runtime* (browsers, en este caso) implementan de diferente manera ciertas características, lo cual provoca que la aplicación o alguno de sus aspectos se comporte distinto dependiendo del browser particular. La portabilidad entre browsers se logra generando código que sea capaz de identificar al navegador particular y actuar de acuerdo al mismo. La posibilidad de generar automáticamente código a partir de especificaciones de mayor abstracción permite incluir fácilmente en la aplicación final este comportamiento de identificación y adaptación, ya sea directamente o a través de librerías específicas. Como se dijo anteriormente, el definir componentes de UI compatibles entre

browsers suele ser tan importante que es identificado como un patrón de diseño particular en lo que a Aplicaciones Ricas se refiere[3].

El mantener las especificaciones de UI de manera suficientemente abstracta también permite que las mismas puedan ser derivadas a tecnologías futuras. Esto es en realidad, una consecuencia de la proximidad del meta-modelo al dominio en cuestión y es precisamente una de las ventajas que la metodología DSM proporciona.

6.12.2 Abstracción en la especificación de la UI

Los componentes de UI que se plantearon en RIADL están orientados a detallar la interacción que tendrá el usuario con la información del modelo de datos subyacente, dentro del marco de las capacidades brindadas por las RIAs. Esto implica que los desarrolladores no necesitan invertir tiempo en trabajar con aspectos de bajo nivel al especificar la aplicación, sino que directamente consideran su definición en términos semánticos y estructurales.

Un ejemplo de esta abstracción son los componentes de tipo *lista*, cuya funcionalidad está orientada a interactuar con una multiplicidad de instancias de datos de determinada *entidad*. Su declaración no se asocia implícitamente con ningún componente de UI de bajo nivel, mientras que en la práctica se definirán uno o más de estos componentes concretos para que las implementen como, por ejemplo, *listboxes*, *carousels*[29] o *tablas*.

Los componentes de UI concretos que implementarán a los abstractos pueden fijarse en los generadores de código o bien pueden ser especificados a través de extensiones efectuadas al meta-modelo. Esta última posibilidad resulta en mayores beneficios si se cuenta con un entorno de meta-modelado capaz de fraccionar el meta-modelo en distintos partes inherentes a diferentes aspectos del dominio. Siendo este el caso, mientras que un grupo de desarrolladores puede llevar a cabo la especificación de la UI en términos estructurales y semánticos, otro equipo con mayor conocimiento de requerimientos y de tecnologías RIA podría decorar los modelos confeccionados por los primeros a fin de especificar características de más bajo nivel relativas a la UI como, por ejemplo, los componentes concretos que implementarán a aquellos abstractos como las *listas*.

6.12.3 Derivación automática de aspectos estructurales o *estáticos*

La semántica de los componentes y sus relaciones planteadas en RIADL permite derivar una gran cantidad de aspectos estructurales.

En primer lugar, el modelo de datos planteado en la sección *Datos*, puede derivarse a un modelo relacional e implementarse en cualquier RDBMS moderno. Los scripts de generación automática de tablas, identificadores, claves foráneas, etc., pueden obtenerse directamente desde los modelos de datos, teniendo en cuenta la composición de las *entidades* y la cardinalidad de sus *relaciones*. De igual manera, puede plantearse la derivación de una capa intermedia entre la base de datos y el frontend que la capa de servicios provee, generando un modelo de clases y el mapeo O-R correspondiente. Esto permite que el modelo físico de la base de datos y el modelo de la aplicación se mantengan separados. La capa de servicios que permitirá la comunicación entre el client- y server-side podrá de igual manera ser derivada en funciones que sean atendidas por un servidor o

contenedor web.

La composición estructural de la interfaz de usuario es otro aspecto que puede ser derivado desde el meta-modelo, de acuerdo a la relación de contención entre componentes. Este aspecto no varía en gran medida de los lenguajes que distintas tecnologías RIAs brindan a fin de describir la UI de las aplicaciones.

6.12.4 Derivación automática de comportamiento o aspectos *dinámicos*

Gran parte de los aspectos mencionados del lenguaje están orientados a definir comportamiento. A continuación, se enumerarán los aspectos más relevantes en cuanto a comportamiento se refiere, capaces a ser derivados desde el meta-modelo propuesto.

Validación

Las validaciones de datos especificadas de acuerdo a las reglas definidas en la declaración de *entidades* y *relaciones* son uno de los principales aspectos dinámicos que pueden generarse automáticamente. Las restricciones de validación implican aspectos de comportamiento en términos de la UI a generar como, por ejemplo, resaltar campos inválidos, mostrar mensajes de error y no permitir efectuar un alta, así como también el incluir otros componentes de UI como checkboxes a fin de indicar si un *atributo* opcional tendrá o no un valor asociado.

Finalmente, las mismas validaciones que son llevadas a cabo en el client-side pueden ser efectuadas en el servidor, a fin de dotar de mayor seguridad y robustez a la aplicación. Esto último habilita a la capa de servicios generada a ser reusada por aplicaciones externas a la implementación derivada desde el meta-modelo, las cuales desconozcan las reglas de validez de datos a priori.

Componentes de UI y su interrelación

El comportamiento e interacción a nivel de componentes de UI particulares como, por ejemplo, un *date picker*, son automáticamente generados desde las definiciones presentes en los modelos. En caso de que la plataforma de ejecución subyacente no brinde componentes con el grado de interactividad y abstracción buscado, pueden utilizarse librerías que faciliten esta tarea y permitan simplificar la derivación. Estas librerías, de ser necesarias, pasarán a ser parte del framework de dominio de los generadores. En este contexto, el código derivado consistirá en la instanciación y configuración del componente de UI particular de acuerdo a lo buscado.

Las dependencias entre componentes de datos como la planteada en el ejemplo de la Figura 32, en el cual una *lista* necesita de un dato provisto por otra para determinar su contenido, son inferidos automáticamente de acuerdo a las *relaciones* establecidas entre componentes, sus datos asociados y las consultas que permiten obtenerlos. Como consecuencia, las actualizaciones relativas al contenido de los elementos de UI son llevadas a cabo de manera automática, sin necesitarse la definición explícita de ningún mecanismo de dependencia y notificación como es usual.

La derivación del código fuente respectivo a la UI incluye también a la asociación de eventos de interfaz de usuario con las acciones que se llevan a cabo frente a su

ocurrencia. Estos eventos y acciones no sólo involucran a aspectos definidos explícitamente en modelos a través de conceptos especificados por RIADL, sino también a características inherentes a componentes particulares. Por ejemplo, para un componente *detail*, puede ser necesario derivar un componente de UI adicional que permita mostrar u ocultar el contenido detallado y asociar este componente a la acción correspondiente.

Interacción de la UI con la capa de servicios

Los llamados a la capa de servicios desde el frontend de la aplicación son derivados directamente desde la especificación provista en los modelos, de manera de efectuarse de acuerdo a los datos que se necesitan y al momento en que los mismos se requieren. Cada petición a la capa de servicios orientada a la obtención de información se configura de manera que cumpla con la calidad y cantidad de los datos buscados, a través del establecimiento de parámetros adecuados que personalicen la respuesta a obtener. Los parámetros de *prefetching* definidos en los modelos son tenidos en cuenta a fin de precaptar la información requerida al cargar la UI o al ocurrir determinados eventos de interfaz de usuario (por ejemplo, un avance de página en una *lista* paginada). De igual manera, el momento en que las peticiones son llevadas a cabo es influido por parámetros de configuración relativos a los elementos de UI orientados a datos. El *caching* y *submission throttling*[3], características dependientes de la implementación particular que se obtendrá desde los modelos, influirán en la cantidad de peticiones que se realicen, así como también lo harán las dependencias de información entre componentes de UI.

El código que lleva a cabo la adaptación de las respuestas obtenidas desde el servidor a los componentes de UI y la sincronización correspondiente es generado de manera automática.

6.12.5 Separación de aspectos

El meta-modelo propuesto separa varios aspectos distintos, muchas veces superpuestos, en la implementación de RIAs.

En primer lugar, de ser necesaria la autenticación de usuarios, las interfaces relativas a la misma y a actividades asociadas pueden ser derivadas automáticamente. El control de acceso a *vistas* de acuerdo al tipo de usuario es modelado directamente de la manera comentada en la sección *Tipificación de usuarios y permisos de acceso*. Como resultado, estando los generadores de código y modelos correctamente definidos, los desarrolladores no deben implementar manualmente los mecanismos de seguridad relativos al control de privilegios de acceso. Una vez probada la correcta generación de la lógica relativa a estos por parte de los derivadores, la misma es replicada a cuantas *vistas* sea necesario, evitando así su implementación manual, la cual es propensa a errores que pueden comprometer la seguridad de la aplicación.

Al estar centradas en el manejo de información, la definición de las *vistas* es dependiente del modelo de datos de la aplicación. Sin embargo, la asociación entre los componentes de UI que requieren estar relacionados a *entidades* y las *entidades* en cuestión puede postergarse a una segunda etapa del desarrollo a fin de elaborar prototipos de *vistas* y

su interacción. Dado el caso, aunque pueden utilizarse como un artefacto de diseño y de captura de requerimientos, es pertinente notar que dichos prototipos no permitirán derivar aplicaciones funcionales hasta que la asociación entre modelo de *vistas* y datos no sea formalizada. La definición interna de cada *vista* puede llevarse a cabo de manera independiente de la navegación, dado que la relación entre ambos aspectos consiste en enlazar elementos de UI con *acciones* de tipo `show`, asociación la cual puede especificarse a posteriori. Finalmente, cualquier dato externo que una *vista* requiera es expresado a través de parámetros de entrada, lo cual adiciona mayor modularidad en su definición.

La especificación detallada de la validación de datos puede efectuarse durante la construcción de las *vistas*, en conjunto con la definición del modelo de datos, o bien a posteriori, dado que esta definición es totalmente independiente de la declaración de los componentes de interfaz de usuario orientados al ingreso de datos.

Todo lo anteriormente mencionado en cuanto a separación de aspectos que RIADL brinda permite la división del trabajo en términos de la especificación de la aplicación, dado que distintos grupos de desarrolladores pueden focalizarse en definir diferentes aspectos de la misma. A su vez, si la herramienta DSM utilizada cuenta con la capacidad de referenciar elementos entre distintos modelos de un mismo tipo definidos de manera separada, un aspecto particular de la aplicación como, por ejemplo, el modelo de datos, puede ser definido por varios desarrolladores a la vez. Esto abre paso a la posibilidad de dividir la especificación de un aspecto concreto del software entre distintos integrantes del equipo de desarrollo y obtener, de esta manera, mayores ganancias en términos de productividad.

7 DEFINICIÓN FORMAL DEL META-MODELO

7.1 Introducción

En esta sección se llevará a cabo la definición formal del meta-modelo en un lenguaje y entorno de meta-modelado específicos. En primer lugar, se comentarán las características de distintas herramientas y lenguajes asociados a la definición de meta-modelos que fueron investigados. Luego, haciendo uso de una de estas herramientas y su lenguaje asociado, se especificará formalmente el meta-modelo de RIADL.

7.2 Herramientas investigadas

Previo a la definición del meta-modelo, se estudiaron distintas herramientas de meta-modelado y sus lenguajes asociados a fin de disponer de información relativa a sus características y optar por la que mejor se adecuen a las necesidades buscadas.

Las herramientas investigadas fueron Eclipse GMF[22], MetaEdit+[23] y GME[24]. A continuación se describirán brevemente las características de cada una, considerando sus cualidades principales, particularidades del meta-meta-modelo, lenguajes de especificación de restricciones y derivación. Asimismo, también se incluirán aspectos relativos a la asistencia que brindan al modelador y en lo respectivo a la confección del meta-modelo. Finalmente, se mencionará cual fue la herramienta elegida para describir formalmente el meta-modelo propuesto en esta Tesina y porqué.

7.2.1 MetaEdit+

MetaEdit+[23] es un entorno integrado para construir y utilizar soluciones DSM y se publicita como el más maduro y completo en la actualidad. El lenguaje que el entorno utiliza es denominado GOPPRR, y sus componentes básicos son: *Grafos* (conjuntos de objetos relacionados), *Objetos* (elementos primitivos que pueden ser definidos en un grafo), *Relaciones* (conexiones entre dos o más objetos), *Roles* (modo en el cual un objeto participa en una relación), *Puertos* (parte especial de un objeto a la cual pueden conectarse roles) y *Propiedades* (atributos atómicos asignados a elementos del lenguaje).

En adición al lenguaje de meta-modelado, MetaEdit+ define un lenguaje de scripting propio orientado a transformaciones de modelos a texto denominado MERL, el cual permite navegar entre los componentes de los modelos y sus relaciones, extraer información y generar contenido textual. La herramienta no provee un lenguaje particular para establecer predicados o restricciones lógicas que los modelos deban satisfacer para considerarse válidos.

Las pruebas efectuadas con MetaEdit+ mostraron que el entorno ofrece una manera sencilla y rápida para generar un meta-modelo y modelos a partir del mismo, incluyendo un editor gráfico interno para facilitar la definición de la sintaxis concreta de los lenguajes. Otras características que la herramienta ofrece son la posibilidad de trabajo colaborativo de múltiples equipos de desarrollo a través de repositorios de modelos y soporte para múltiples plataformas.

7.2.2 GME

GME[24] se define como un *toolkit configurable para crear modelos específicos de dominio y ambientes de síntesis de programas*. La herramienta permite definir meta-modelos a través de un lenguaje basado en UML y establecer restricciones de validez a los mismos utilizando una variante de OCL adaptada a su meta-meta-modelo particular.

El lenguaje que GME provee está compuesto por *Átomos* (objetos de modelado que sólo poseen atributos), *Conexiones* (relaciones entre dos partes de un modelo, las cuales pueden ser direccionales o no), *Links* (elementos que funcionan como *puerta de entrada* a modelos y admiten ser relacionados con otros conceptos del lenguaje a través de conexiones) y *Atributos* (propiedades que los objetos antes mencionados pueden poseer). Adicionalmente, el lenguaje define un concepto denominado *Aspecto* a fin de reducir la complejidad de los meta-modelos: un *Aspecto* agrupa una serie de elementos del meta-modelo los cuales serán observados de forma aislada en tiempo de modelado, ocultándose cualquier otro que no pertenezca al grupo que el mismo define. Esto permite modularizar el meta-modelo de acuerdo a las diferentes facetas del dominio que el mismo representa, facilitando la división de las especificaciones y promoviendo un número mayor de roles entre los modeladores.

GME, a diferencia de MetaEdit+, no brinda un lenguaje propio para especificar derivaciones de modelos a texto. En su lugar, implementa una interfaz denominada BON, a partir de la cual se puede acceder y modificar los modelos confeccionados directamente desde distintos lenguajes de programación como C++ o Java. De esta manera, además de consultar programáticamente los modelos, pueden efectuarse modificaciones en los mismos como si se trabajara directamente con el entorno de modelado que la herramienta provee.

7.2.3 Eclipse GMF

Eclipse GMF[22] (Graphical Modeling Framework) es una herramienta que permite desarrollar editores gráficos que corren sobre el IDE Eclipse. Estos editores brindan una manera gráfica, interactiva y asistida de construir modelos, los cuales son definidos a través de un entorno de meta-modelado subyacente. Este último es denominado Eclipse EMF (Eclipse Modeling Framework) y se define como un *framework de modelado y generación de código para construir herramientas y aplicaciones en base a modelos*. Por ende, ambas herramientas (Eclipse EMF y Eclipse GMF) funcionan en conjunto, siendo la última dependiente de la primera y cumpliendo el rol de asistente gráfico para ésta. Por simplicidad, a posteriori en este documento, cuando se haga referencia a Eclipse GMF se estará haciendo alusión a ambas tecnologías utilizadas en conjunto para definir meta-modelos y editores gráficos.

El lenguaje de modelado que utiliza Eclipse GMF es denominado Ecore, y es una implementación particular de MOF[25], un estándar de la OMG para Desarrollo Conducido por Modelos. El lenguaje Ecore será comentado en detalle en la sub-sección posterior.

A diferencia de las tecnologías mencionadas hasta el momento, Eclipse GMF no provee un único lenguaje de transformación de modelos a texto. En su lugar, existe un proyecto relacionado denominado Eclipse M2T, el cual se enfoca en la definición e implementación de distintos lenguajes de transformación. En particular, en las pruebas llevadas a cabo durante el desarrollo de esta Tesina, se investigó el uso del lenguaje de

derivación de modelos a texto MOFScript[26].

Finalmente, el entorno permite especificar reglas de validez a los modelos a través de predicados lógicos expresadas en lenguaje OCL.

El lenguaje Ecore

A continuación, se enumerarán los conceptos que conforman el núcleo del lenguaje Ecore [27].

- `EClass`: define *clases*, elementos del lenguaje identificados con un nombre y que contienen cierta cantidad de atributos y/o referencias. El concepto es similar al concepto de *clase* en el lenguaje UML.
- `EAttribute`: modela *atributos*, componentes de datos básicos de cualquier objeto en el modelo. Se identifican por un nombre y poseen un tipo de dato asociado. Se asimilan al concepto de *atributo* utilizado en el lenguaje UML.
- `EDataType`: modela tipos de datos asociados a los `EAttribute`s. Ecore adopta los tipos de datos definidos en el lenguaje Java y, por convención, los define anteponiendo una letra `E` detrás de su nombre original. Por ejemplo, los tipos de datos `String` e `Int` son definidos en Ecore como `EString` y `EInt`. Adicionalmente, el lenguaje define una clase especial de `EDataType` llamado `EEnum`, el cual permite especificar tipos de datos enumerativos.
- `EReference`: modela asociaciones entre clases. Una `EReference` es contenida en una clase y especifica una asociación direccional la cual parte desde la clase en la que es contenida y referencia a una clase destino. Por ende, de existir una relación bidireccional entre clases, la misma debe representarse como dos `EReference`s, una incluida en cada clase, las cuales referencian a la clase opuesta. Al igual que un `EAttribute`, una `EReference` tiene un nombre y un tipo de dato asociados. Este último representa, en este caso particular, a la clase destino a la cual la asociación referencia. Este tipo de componente del lenguaje contiene además límites mínimos y máximos de cardinalidad. Estos límites definen la cantidad de instancias mínima y máxima de la clase referenciada que se puede asociar a través de la `EReference` particular. Finalmente, las `EReference`s pueden utilizarse para definir relaciones de contención (llamadas *agregaciones por valor* en UML[27]), las cuales se denotan a través de la propiedad `containment`. Este tipo de asociaciones serán descriptas con mayor detalle debajo.

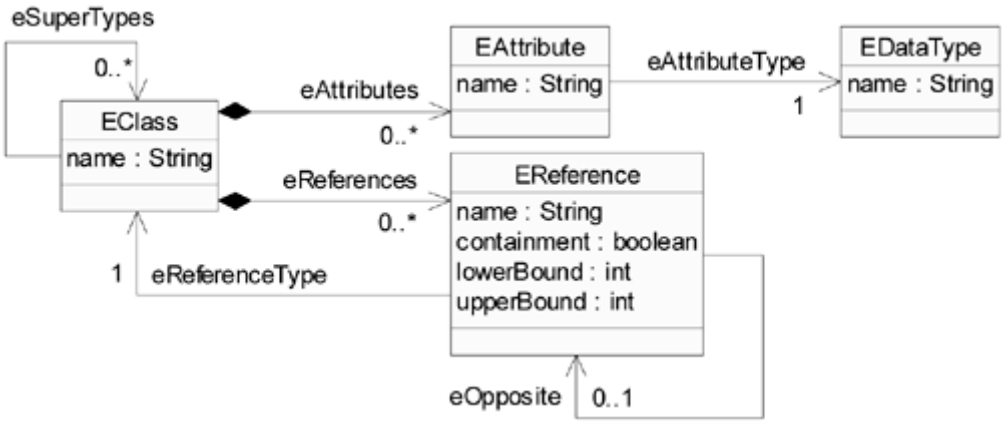


Figura 42. Relación existente entre los componentes que conforman el núcleo del lenguaje de meta-modelado Ecore.

La relación entre los conceptos anteriormente definidos es esquematizada en la Figura 42[30]. En la misma, pueden observarse algunos detalles que no han sido explicitados en lo dicho anteriormente. En primer lugar, una clase puede tener *supertipos* (asociación `eSuperTypes`), los cuales se definen como una o más `EClass`es de las cuales la clase particular hereda. La semántica de esta herencia implica que la clase que la defina incluirá todos los atributos y asociaciones que su(s) superclase(s) define(n). Adicionalmente, puede observarse cómo se modelan las relaciones bidireccionales a través de la asociación `eOpposite`.

La relación de contención comentada entre una `EClass` hacia otra a través de una `EReference` establece tres reglas semánticas a respetar entre las instancias de las `EClass`es utilizadas en los modelos[27]:

1. Un objeto del modelo no puede contener, directa o indirectamente, a otro que lo contiene.
2. Un objeto del modelo no puede ser contenido por más de una instancia contenedora.
3. El tiempo de vida del objeto contenido está acotado por el tiempo de vida del objeto que lo contiene

Estas reglas aseguran que se cumpla el concepto de *es parte de* que la semántica de la *agregación por valor* promueve.

Además de los componentes primarios ya mencionados, el lenguaje Ecore define varios aspectos extra como, por ejemplo, `BehavioralFeatures`. Se considera que los mismos están fuera del alcance de este documento, dado que no se busca describir en el mismo el lenguaje en detalle, sino más bien brindar una base formal que permita especificar de manera concreta el meta-modelo que se propone.

7.2.4 Eclipse GMF: Herramienta elegida para especificar el meta-modelo

De las herramientas mencionadas, se elige para definir el meta-modelo que en esta

30 Por ser similar a la del lenguaje UML, la sintaxis concreta del lenguaje Ecore (la utilizada en la Figura 42) no será detallada en este documento.

Tesina se propone a Eclipse GMF. Un primer motivo de esta elección es el uso del lenguaje Ecore que la misma utiliza como meta-meta-modelo, el cual está basado en el estándar MOF. De igual manera, los meta-modelos son serializados por el framework a través del formato de intercambio de meta-modelos estandarizado XMI. Estas dos características de la herramienta que promueven la adecuación a estándares favorecen la portabilidad de los meta-modelo implementados.

Asimismo, a partir de un meta-modelo particular, Eclipse GMF permite generar el software de modelado asistido en forma de plug-in para el IDE Eclipse. Esto independiza el entorno de meta-modelado del de modelado haciendo que los mismos sean dos artefactos de software distintos. Adicionalmente, la herramienta cuenta con una variedad de lenguajes de transformación entre modelos y de modelos a texto y, al ser de código abierto, permite mejoras o personalizaciones, tanto en el mismo framework de meta-modelado como en el código relativo a los entornos de modelado generados para meta-modelos particulares.

Finalmente, se comprobó mediante ejemplos la eficiencia del lenguaje MOFScript para especificar transformaciones de modelos a representaciones textuales, por lo cual es este lenguaje el que se ha escogido para implementar este tipo de derivaciones.

7.3 Definición del meta-modelo

En esta sección se definirá formalmente el meta-modelo en el lenguaje Ecore. La definición respetará la especificación informal y asumirá todo lo mencionado en la misma. La composición del lenguaje estará separada en distintos aspectos relacionados entre sí, los cuales serán tratados en distintas secciones.

7.3.1 Modelo de datos y de control de acceso

El elemento central de meta-modelo de datos planteado por RIADL son las *entidades*. Las mismas son representadas por una `EClass` llamada `Entity`, y tienen como atributo único un nombre (`name`) que las caracteriza. Como ya fue comentado, una *entidad* contiene uno o más *atributos*, los cuales en el meta-modelo son representados por la `EClass Attribute`. Cada uno de ellos deberá especificar un nombre (atributo `name`), un tipo de datos asociado (atributo `dataType`) y una condición de obligatorio u opcional, orientada a validación (atributo `mandatory`). El tipo de dato que un `Attribute` puede tener asociado es especificado a través del enumerativo (`EEnum`) `DataType`, el cual define los siguientes tipos: `STRING` (texto), `DATE` (fecha), `IMAGE` (imagen), `FLOAT` (número decimal) e `INTEGER` (número entero).

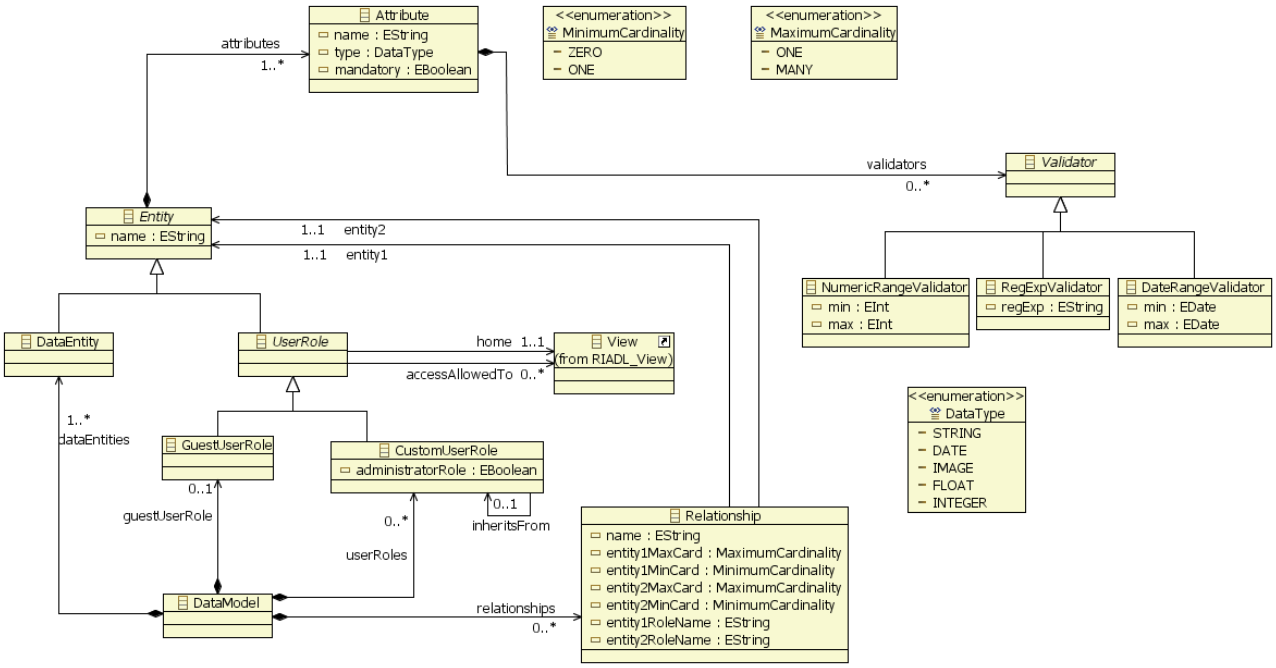


Figura 43. Meta-modelo de datos y control de acceso del lenguaje RIADL

El segundo elemento relevante en lo que al modelo de datos se refiere son las *relaciones*, las cuales son definidas a través de la `EClass Relationship`. En sintonía con su definición informal, las mismas tienen un nombre que las caracteriza (atributo `name`) y relacionan a dos *entidades*, las cuales son vinculadas entre sí a través de las asociaciones `entity1` y `entity2`. A su vez, una *relación* define un *nombre de rol* particular para cada una de estas *entidades* (`entity1RoleName` y `entity2RoleName`), así como también una cardinalidad mínima y máxima para las mismas (`entity1MinCard`, `entity1MaxCard`, `entity2MinCard` y `entity2MaxCard`). Las cardinalidades mínimas y máximas son representados con enumerativos a fin de brindar mayor expresividad.

La Figura 43 muestra la sintaxis concreta de la fracción del meta-modelo de RIADL orientada a la definición del modelo de datos y control de acceso. En la misma puede apreciarse que la `EClass Entity`, orientada a modelar *entidades*, es en realidad abstracta, y posee dos subclases: `DataEntity` y `UserRole`. Mientras que `DataEntity` representa el concepto de *entidad* como fue descripto hasta el momento en el lenguaje, `UserRole` es una `EClass` abstracta la cual define *roles de usuario*. Al igual que cualquier *entidad*, un *rol de usuario* se considera poseedor de un conjunto de *atributos* que caracterizan a sus usuarios asociados y puede participar en *relaciones* con otras *entidades*. Esta característica, implementada a partir de una relación de herencia entre `UserRole` y `Entity`, permite modelar información asociada a usuarios particulares de la aplicación, discriminando de acuerdo a su *rol* particular, lo cual suele ser un requerimiento usual en sistemas multi-usuario y orientados a datos que requieren distintos perfiles de acceso.

La jerarquía planteada a partir de `UserRole` es acorde a la definición informal del meta-modelo en lo respectivo a *roles de usuario*: la subclase `GuestUser` engloba a usuarios no registrados o autenticados en el sistema, mientras que `CustomUserRole` permite definir *roles* correspondientes a usuarios registrados y autenticados en la aplicación. El atributo

booleano `administratorRole` permite determinar si el *rol* es clasificado o no como *de administrador*, y la asociación `inheritsFrom` permite establecer relaciones de herencia entre *roles*, haciendo referencia a aquellos cuyas características son heredades por el *rol* particular. Finalmente, se incluye en la clase `UserRole` dos asociaciones hacia la `EClass View`, la cual representa en el lenguaje a una *vista* y que será descripta a posteriori. La asociación `home` determina la *vista* que será mostrada al usuario inmediatamente luego de su ingreso al sistema, mientras que `accessAllowedTo` especifica aquellas *vistas* a las cuales el usuario tiene acceso. Ambas características son dependientes del *rol de usuario* particular.

Como última característica del meta-modelo de datos y control de acceso se consideran aspectos de validación los cuales, como ya se ha comentado, son especificados a través de elementos de tipo *validador*, asociados a *atributos* particulares. En términos de implementación, se representarán relacionando elementos de clase `Attribute` con instancias de `Validator`, a través de la relación `validators`. Se definen tres tipos de *validadores*: `NumericRangeValidator` (valida que el valor numérico de un *atributo* particular esté dentro de determinado rango), `RegExpValidator` (valida que el contenido textual de un *atributo* particular cumpla con determinada expresión regular) y `DateRangeValidator` (controla que el valor de fecha de un *atributo* particular esté dentro de un rango fijo). Los *validadores* orientados a rangos también permiten controlar que el contenido de un *atributo* sólo sea mayor o menor a determinado valor constante, lográndose esto a través de la declaración de uno de los dos valores de límite que los mismos permiten especificar y dejando el otro como indefinido.

Finalmente, la `EClass DataModel` es la clase *raíz* de esta fracción del meta-modelo, dado que contiene a todas aquellas `EClass`es que pueden instanciarse en los modelos de este tipo. En el entorno de modelado generado por EMF, la instancia de esta clase es construida automáticamente al solicitar la creación de un nuevo modelo y todos los componentes que se utilicen en este último serán contenidos directa o indirectamente por la misma. En este caso particular, como puede observarse en la Figura 43, un modelo de datos y control de acceso (instancia de `DataModel`) deberá contener una o más `entidades` (instancias de `DataEntity`), cero o más *roles de usuario* concretos (instancias de `CustomUserRole`), podrá referenciar al de *rol de usuario* especial `GuestUserRole` para asociarlo a *atributos* y relacionarlo con *entidades*, y será capaz de definir cero o más *relaciones* entre *entidades* particulares (creando instancias de `Relationship`).

Resta aclarar que, a pesar de que a través de los conceptos que Ecore provee se definen de por sí gran parte de las restricciones de validez del meta-modelo, no pueden asegurarse el cumplimiento de algunas invariantes tales como, por ejemplo, el uso de *validadores* `DateRangeValidator` sólo asociados a *atributos* de tipo `DATE`. Con el objetivo de mantener el meta-modelo lo más sencillo posible en términos estructurales, este tipo de validaciones serán implementadas mediante el lenguaje de restricciones OCL.

7.3.2 Modelo de vistas

El núcleo del meta-modelo de *vistas* (Figura 44) está conformado por la `EClass` *raíz* `ViewModel`. La misma debe contener una o más *vistas* (instancias de la `EClass View`), las cuales son caracterizadas por un nombre particular (atributo `name`) y por el modo en que son mostradas, aspecto el cual es representado por el atributo booleano `popupMode`. De ser

falso, este último indica que la *vista* que lo define será navegada, mientras que de ser verdadero denota que la misma será mostrada sobre el nodo navegacional actual. Una *vista* define también un conjunto opcional de *parámetros* (instancias de `ViewParameter`) y debe contener uno o más componentes de interfaz de usuario (definidos por la `EClass` abstracta `UIComponent` y tratados más adelante).

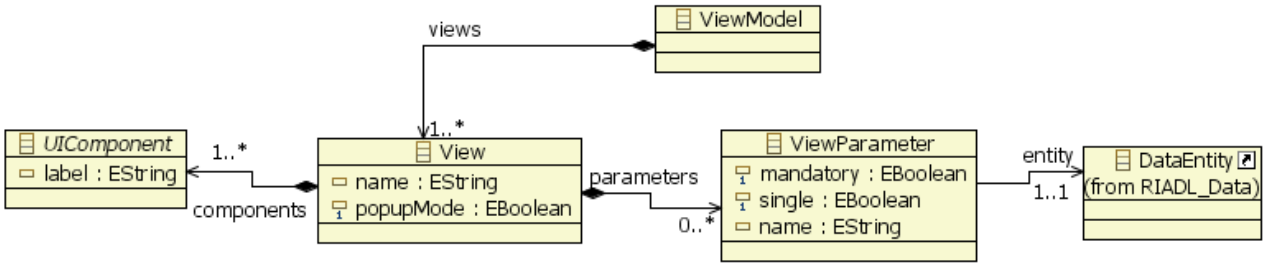


Figura 44. Núcleo del meta-modelo de *vistas* de RIADL.

Un *parámetro* de *vista* puede ser opcional (atributo `mandatory` falso) u obligatorio (atributo `mandatory` verdadero), puede acarrear una o más instancias de una *entidad* particular (atributo `single` verdadero o falso respectivamente), y puede ser caracterizado opcionalmente por un nombre (`EAttribute name`). La *entidad* de la cual deberán ser instancia los objetos acarreados en el parámetro al mostrarse la *vista* se especifica a través de la asociación `entity`. La flecha ubicada sobre el extremo superior derecho de la representación gráfica de la `EClass DataEntity` en la Figura 44 denota que la clase no es definida en el mismo meta-modelo de la figura, sino que representa en realidad a una `EClass` declarada en un meta-modelo distinto, en este caso particular, en el meta-modelo de datos y control de acceso definido en la sección anterior.

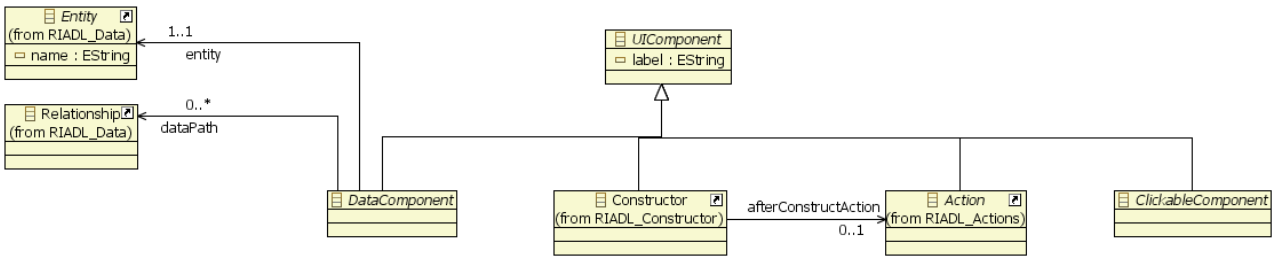


Figura 45. Clasificación de componentes de interfaz de usuario (jerarquía planteada por `UIComponent` y sus subclases)

Los componentes de interfaz de usuario (`UIComponent`) son una de las clases más importantes de esta parte del meta-modelo. La taxonomía de los mismos se define de la siguiente manera (ver Figura 45):

- `Action`s, elementos del lenguaje que representan cierta actividad o tarea (simple o compuesta) a ser ejecutada frente a eventos particulares de UI. Serán tratados más adelante.
- `DataComponent`s, cualquier componente fuertemente ligado a una o más

instancias de datos con el objeto de mostrar o modificar sus propiedades.

- `Constructor`s, componentes capaces de crear instancias de datos. Ofrecen la posibilidad de especificar acciones a ejecutar luego de la creación exitosa de las susodichas instancias mediante la asociación `afterConstructionAction`. Su especificación será vista en detalle en una sección particular.
- `ClickableComponent`s, cualquier componente que es capaz de capturar eventos de *click* y especificar una acción a ejecutar como respuesta a los mismos.

Como se ha mencionado en secciones anteriores, un componente de interfaz de usuario orientado a datos (representado por la `EClass DataComponent`) tiene como atributo asociado a una *entidad* particular de la cual poseerá una o más instancias con las que trabajará. Asimismo, este tipo de componentes también hará referencia a una lista ordenada de cero o más *relaciones* que deberán atravesarse para alcanzar dichas instancias, partiendo desde un origen de datos específico. Las asociaciones `entity` y `dataPath` permiten especificar estas características.

Entre los componentes de UI orientados a datos, surge una diferenciación natural entre compuestos (`CompoundDataComponent`) y simples (`SimpleDataComponent`) de acuerdo a si pueden o no contener componentes internos respectivamente. Esta clasificación puede apreciarse gráficamente en la Figura 46. En adición a esto, se observó que aquellos componentes compuestos como, por ejemplo, *contexts* o *lists*, son capaces de especificar su origen de datos de distintas maneras, mientras que los atómicos como *images* o *texts* sólo pueden obtenerlos contextualmente. Esto propone que aquellos componentes clasificados como *compuestos* no sólo se caracterizarán por la posibilidad de contención de otros componentes internos (aspecto modelado por la asociación `subcomponents`), sino también por cómo especifican su origen de datos, aspecto el cual es manifestado a través de la asociación `dataSource` con una instancia de `EntityInstanceDataSource`. Esta última es una `EClass` que engloba a elementos del lenguaje capaces de referenciar instancias de datos, las cuales pueden utilizarse como fuente de información para los componentes de UI que la requieran (en este caso, `CompoundDataComponent`s).

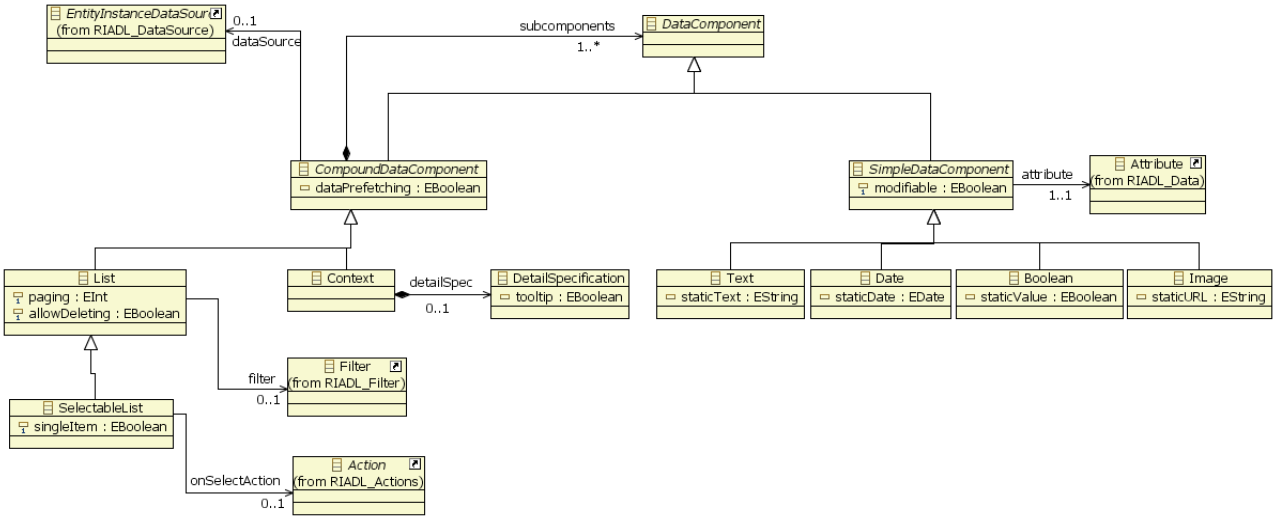


Figura 46: Fracción del meta-modelo relativa a componentes de UI orientados a datos

Dentro de los componentes de datos simples se hallan los *texts*, *dates*, *booleans* (muestran valores de verdad) e *images*, representados por las `EClasses` `Text`, `Date`, `Boolean` e `Image`, respectivamente. De acuerdo a lo ya estipulado, estos componentes pueden ser dinámicos, o estáticos. En el primer caso, el contenido de los mismos es definido por el valor de un *atributo* particular (especificado por la asociación `attribute`) de la instancia de datos aportada por el contexto de información de su componente contenedor. Tratándose de un componente estático, deberá definirse un valor concreto a través de la única propiedad particular que cada clase de componente define de manera acorde al tipo de dato que requiere. Finalmente, para `SimpleDataComponents` dinámicos, la propiedad booleana `modifiable` detalla si el componente permite o no modificar el valor del *atributo* particular de la instancia a la cual referencia. Estas modificaciones estarán asistidas en la UI final por componentes de interfaz concretos de acuerdo al tipo de `SimpleDataComponent` elegido.

Los componentes de datos compuestos comparten la propiedad `dataPrefetching` que, de ser verdadera, implica que pre-captarán los datos de acuerdo a lo ya comentado previamente. Este tipo de componentes está subdividido en dos clases:

- `List`: representa a una *lista*. Define dos propiedades. La primera, `paging`, debe ser un número entero el cual, de ser mayor a cero, determina que el componente hará paginación de datos, estando la longitud de sus páginas especificadas por el valor de la propiedad en cuestión. Por otro lado, el `EAttribute` booleano `allowDeleting` establece, de ser verdadero, la posibilidad de que el usuario pueda eliminar instancias de datos a través de este componente. Adicionalmente:
  - La subclase `SelectableList` caracteriza a una *lista* la cual brinda la posibilidad de selección de datos. Si su propiedad `singleItem` es verdadera, sólo permite la selección de un elemento, mientras que de ser falsa, habilita la selección de uno o más. Este tipo de *lista* puede especificar una acción a ejecutarse luego de que un elemento es seleccionado, la cual es definida a través de la asociación

onSelectAction. En el caso de que la propiedad allowDeleting fuera verdadera, se proveerá un elemento de UI que permita eliminar el elemento actualmente seleccionado. Si la misma estuviera habilitada y la *lista* fuera en su lugar una instancia de la superclase List, dada la imposibilidad de seleccionar elementos, se facilitará un componente de UI por cada ítem listado el cual permita llevar a cabo su eliminación, siguiendo el pattern *In-Context Tool*.

- Las *listas* pueden especificar un *filter*, el cual es plasmado en el meta-modelo en la EClass Filter, la cual será tratada en detalle posteriormente.

- Context: define a un *contexto*. Dado que, a excepción de algunos aspectos de comportamiento, los *contexts* comparten gran parte de sus características con *details*, estos últimos son definidos vinculando una instancia de Context con otra de DetailSpecification. Esta EClass *decora* al *context* con aspectos relativos a la semántica de un *detail*. En particular, el atributo booleano tooltip especifica si el *detalle* será tratado como un *tooltip* o será mostrado de manera convencional, como ya se ha relatado.

Al igual que en el caso anterior, esta fracción del meta-modelo requiere la especificación de reglas de validación tales como el corroborar que un componente de UI orientado a datos, simple y dinámico referencie a un *atributo* correspondiente a la *entidad* la cual se le vincula a través de la asociación entity. Otro ejemplo de validación requerida la cual implica mayor complejidad consiste en controlar que la lista de *relaciones* especificada por el atributo dataPath de un CompoundDataComponent sea correcta y exista en el modelo de datos.

7.3.3 Orígenes de datos

En esta sección se comentará cómo el meta-modelo permite especificar *orígenes de datos*. Un *origen de datos* o *data source* es un tipo de elemento del lenguaje el cual contiene un valor inherente que puede ser utilizado por otros componentes del meta-modelo para especificar determinado comportamiento o característica. Por ejemplo, los elementos seleccionados de una *lista* pueden ser utilizados en un *constructor* para determinar las instancias que se asociarán al objeto a construir a través de una *relación* concreta. En este caso, la *lista* que provee las instancias funciona como *data source*. Los conceptos aquí introducidos serán utilizados en la definición de distintos aspectos de la definición formal de RIADL que restan comentar.

Todo *origen de datos* hereda directa o indirectamente de la EClass DataSource (Figura 47). Los mismos son divididos en dos grupos: Por un lado se definen los ValueDataSources, capaces de proveer valores atómicos de determinado tipo como, por ejemplo, *strings* o fechas y, por otro, los EntityInstanceDataSources, orientados a referenciar instancias particulares de determinada de *entidad*.

Los ValueDataSources se subdividen en dos clases: InputComponents y StaticValues. Los primeros representan a componentes de UI a través de los cuales el usuario puede ingresar la información requerida. Cada uno de estos componentes es capaz

de proveer valores de un tipo determinado (por ejemplo, un `Textbox` puede proveer sólo contenidos textuales mientras que un `DatePicker` es capaz de suministrar valores de fecha solamente). Las instancias de `StaticValue` permiten especificar valores fijos de acuerdo al tipo de dato que se requiera.

Los `EntityInstanceDataSource`s, de acuerdo al meta-modelo definido hasta el momento, pueden ser 3:

- `SelectableList`: provee como valor a aquellas instancias que hayan sido seleccionadas en la *lista* particular.
- `ViewParameter`: provee como valor a aquellas instancias que hayan sido pasadas a través del *parámetro* correspondiente al mostrarse la *vista* que lo define.
- `Contextual`: obtiene la información desde el contexto de datos de su componente contenedor.

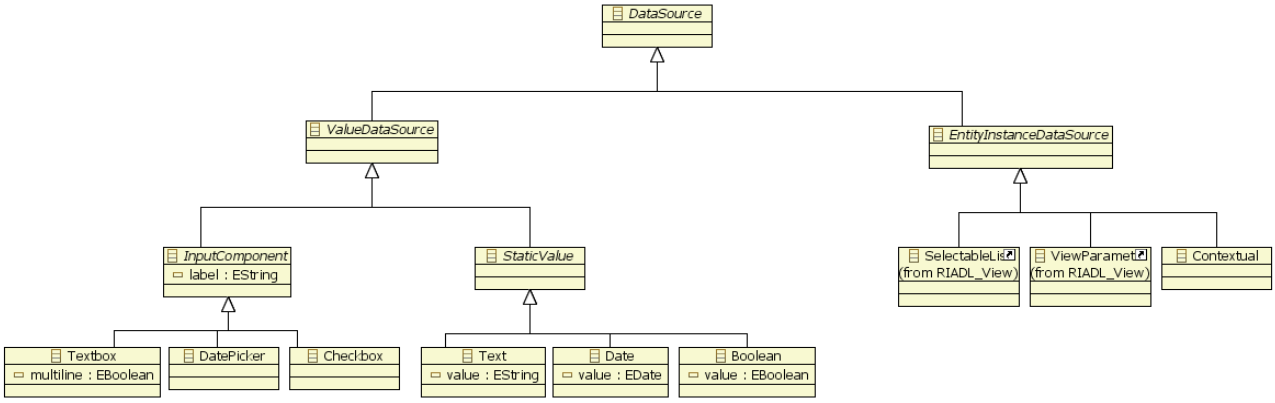


Figura 47. Esquema de la fracción del meta-modelo destinada a representar *orígenes de datos.*

7.3.4 Constructores

De acuerdo a lo comentado en la definición informal del lenguaje, los *constructores* son componentes de interfaz de usuario orientados a la creación de instancias de datos de una *entidad* particular. En el meta-modelo, los mismos serán definidos a través de la `EClass Constructor`, y su *entidad* asociada será especificada a través de la asociación `entity`. La cantidad de instancias que el *constructor* podrá crear será denotada por el valor de la propiedad booleana `multipleInstances` el cual, de ser falso, implicará que sólo permitirá la creación de una única instancia mientras que, de ser verdadero, se le dará la posibilidad al usuario de crear una o más.

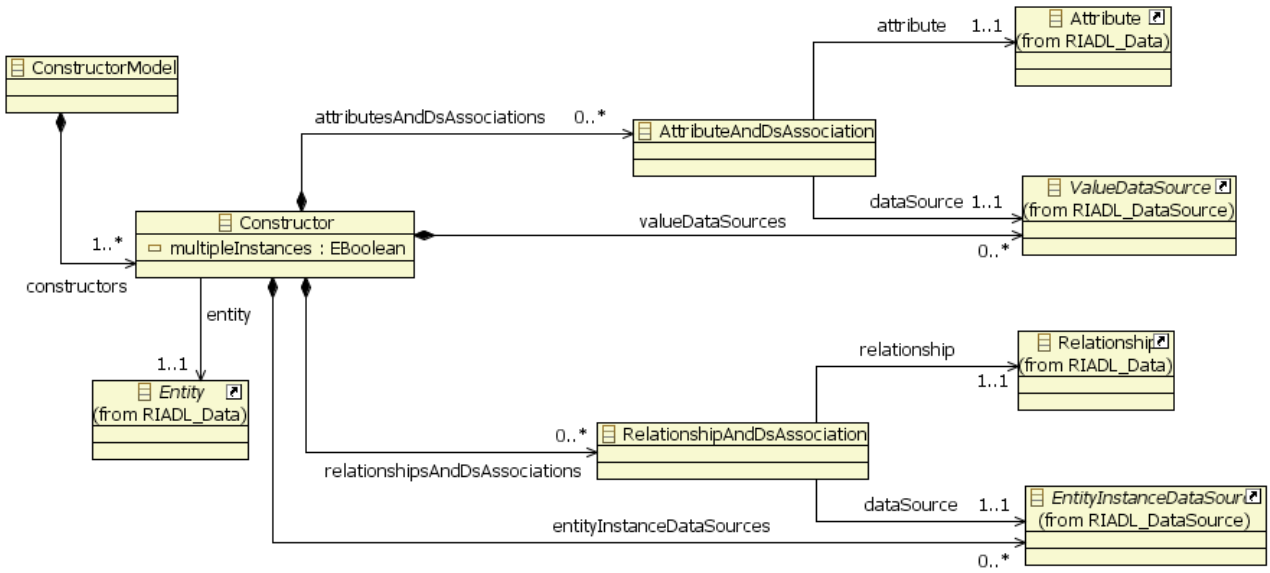


Figura 48. Fracción del meta-modelo de RIADL orientada a la definición de *constructores*

La manera en que los *constructores* permiten la creación de instancias de datos de determinada *entidad* es vinculando *atributos* y *relaciones* inherentes a ésta con valores tomados de alguna fuente particular. El vínculo entre valores y *atributos* es especificado formalmente en el meta-modelo a través de la asociación de una o más instancias de `AttributeAndDsAssociation` al `Constructor` en cuestión. Cada una de éstas permite relacionar un *origen de datos* orientado a la provisión de un valor (asociación `dataSource`) con un *atributo* concreto (asociación `attribute`). El vínculo entre *relaciones* de la instancia a crear y las instancias a relacionar se proveen de manera similar a través de la definición de una o más `RelationshipAndDsAssociation`s asociadas al `Constructor` particular. Tanto los *orígenes de datos* que proveen valores como los orientados a la provisión de instancias deben ser definidos dentro del `Constructor` a través de las asociaciones de contención `valuesDataSources` y `entityInstanceDataSources`. Con respecto a esto último, es pertinente destacar que no todo subtipo de `EntityInstanceDataSource` puede incluirse dentro de un componente `Constructor`. Un ejemplo de este tipo de excepciones son los *parámetros de vista* (`ViewParameter`), los cuales según su semántica sólo deberían ser contenidos dentro de instancias de `View`.

Al igual que en los casos anteriores, deberán considerarse como parte de la definición de esta fracción del meta-modelo validaciones que corroboren la corrección semántica de los modelos confeccionados. En este caso particular, por ejemplo, deberá ser necesario controlar que cada *relación* incluida en una `RelationshipAndDsAssociation` tenga asociada en uno de los extremos a la *entidad* referenciada por el *constructor* a través de la `eReference entity`.

7.3.5 Filter

De acuerdo a lo ya establecido, los *filters* son componentes del lenguaje que pueden asociarse a *listas*, los cuales contienen una o más condiciones lógicas que permiten filtrar las instancias de datos que en estas últimas serán mostradas.

En términos de implementación, un *filter* es representado por la `EClass Filter` (Figura 49). Su contenido consiste principalmente en una o más condiciones de filtrado, las cuales son especificadas a través de instancias de `FilterCondition`. Cada una de estas instancias agrupa a un *atributo* de una *entidad* particular con un *origen de datos* capaz de proveer un valor concreto (`ValueDataSource`), definiendo además un operador de comparación que los relaciona. En términos semánticos, el valor del *atributo* será comparado con el obtenido desde el *origen de datos* utilizando el operador especificado, el cual puede ser cualquiera de los definidos en el `EEnum` (`ComparisonOperator`).

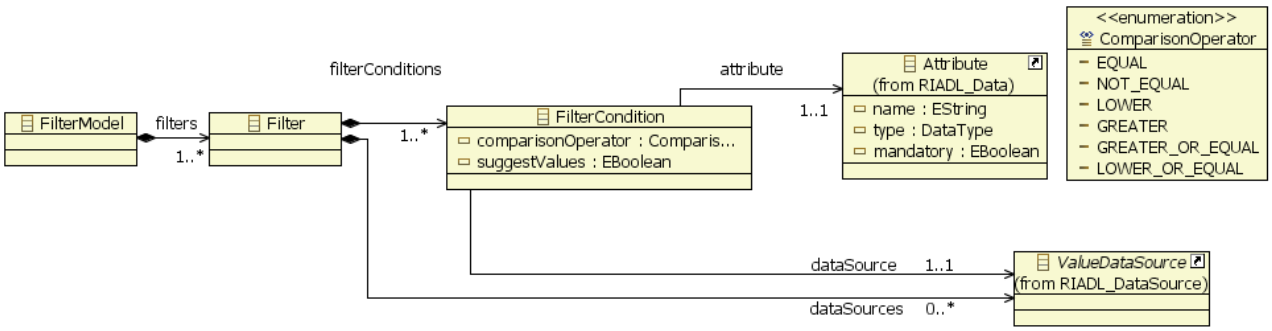


Figura 49. Fracción del meta-modelo orientada a la definición de *filtros*

Los *orígenes de datos* utilizados para confeccionar las condiciones pueden ser incluidos dentro de la misma instancia de `Filter` a través de la `eReference dataSources`. Adicionalmente, las `FilterConditions` pueden especificar la posibilidad de brindar sugerencias interactivas si los valores son obtenidos desde componentes de UI (instancias de `InputComponent`), estableciendo su propiedad `suggestValues` como verdadera.

Las validaciones concernientes a esta fracción del meta-modelo consistirán en evaluar que el operador de comparación sea válido de acuerdo al tipo del *atributo* y valor a comparar, que se use sugerencia interactiva sólo si se utiliza como *origen de datos* una subclase de `InputComponent`, entre otras.

7.3.6 Acciones

En términos semánticos, en RIADL las *acciones* representan una actividad o tarea a ser ejecutada frente a la ocurrencia de determinado evento de UI. En el meta-modelo, toda *acción* es definida como subclase de la `EClass` abstracta `Action` (Figura 50). De acuerdo a lo descripto en la definición informal del lenguaje, existen tres *acciones* principales:

- `Close`, lleva a cabo el ocultamiento de la *vista* en la que es declarada, si la misma es mostrada en modo *popup*. En caso contrario, su uso es inválido y debe considerarse como un error semántico.
- `Show` muestra una *vista*, ya sea efectuando una navegación o bien sobre el nodo navegacional actual. La *vista* particular a ser mostrada es definida a través de la asociación `viewToShow` y el modo en que la misma será presentada dependerá del valor de su propiedad `popup`, de manera acorde a lo comentado en la descripción informal del lenguaje. Dado que las *vistas*

pueden definir uno o más *parámetros* obligatorios, este tipo de *acción* debe permitir proveerle a los mismos instancias de datos tomadas de algún *origen de datos* particular. A este fin, se le asocia a `Show` un conjunto de `ParameterAndSourceAssociation`, `EClass` la cual tiene como único objetivo relacionar un parámetro de *vista* con un *origen de datos* orientado a proveer instancias (`EntityInstanceDataSource`).

- `ChangeObjectRelationship`, *acción* la cual crea o destruye *relaciones* entre instancias de datos particulares. Para esto, requiere que se especifiquen dos *orígenes de datos* los cuales definan las instancias a asociar o disociar (`eReferences source` y `destination`) y la *relación* a través de la cual la acción en cuestión será efectuada (`eReference relationship`). El tipo de dato enumerativo `RelationshipAction` permitirá determinar si la instancia de `ChangeObjectRelationship` implicará una asociación o una disociación entre las instancias de datos involucradas, en el contexto de la *relación* particular especificada.

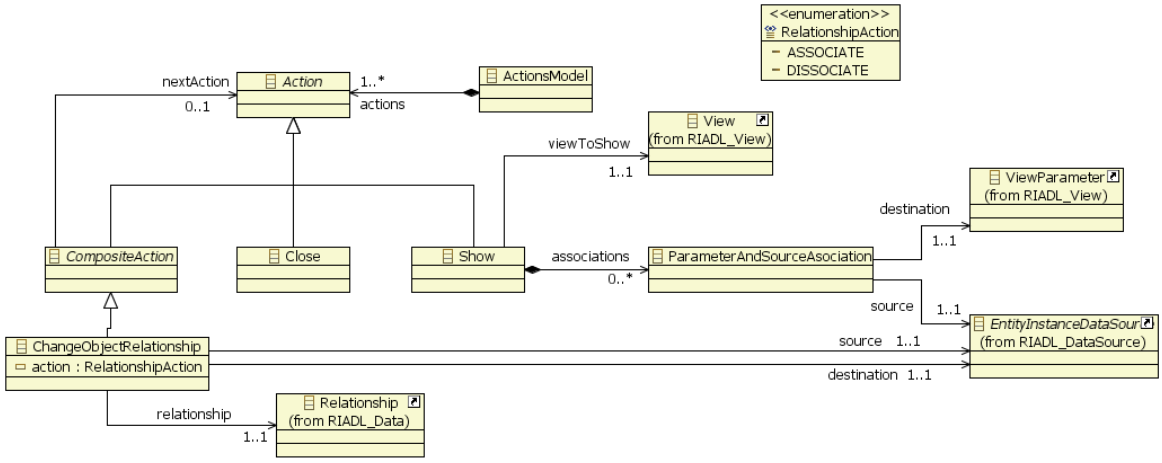


Figura 50. Meta-modelo de *acciones* de RIADL.

7.4 Conclusión

En esta sección se definió formalmente el lenguaje RIADL, el cual fue especificado de manera informal en la sección anterior. En primer lugar, se relataron brevemente las principales características de las herramientas de meta-modelado que fueron investigadas con antelación a la elaboración de cualquier especificación. Posteriormente, se fundamenta el porqué de la elección de Eclipse EMF/GMF como la herramienta utilizada para formalizar el meta-modelo.

Luego de tratados los aspectos relativos a herramientas y lenguajes de meta-modelado, se comienza la descripción formal de RIADL. La primer fracción del meta-modelo a introducir es el modelo de datos y control de acceso, el cual permite definir la estructura de la información que la RIA manipulará y qué perfiles de usuario existirán en la misma. Acto seguido, se continúa con la especificación del modelo de *vistas*, el más

complejo de todo el lenguaje y que permite definir los aspectos de interfaz de usuario que caracterizan principalmente a una Rich Internet Application. Finalmente, se detallan partes del meta-modelo que son utilizadas por las anteriormente definidas: *orígenes de datos*, *constructores*, *filtros* y *acciones*. A través de éstas se termina de especificar el *meta-modelo de vistas* y se completa la definición.

8 IMPLEMENTACIÓN

8.1 Introducción

En esta sección se describirán aspectos de implementación relativos a las RIAs generadas a partir del lenguaje propuesto, su derivación y el entorno de modelado que permite especificarlas. Se brindará primero un panorama general de la arquitectura de implementación de las facetas cliente y servidor, para luego describir aspectos relacionados con la derivación de cada una de éstas. Finalmente se comentarán características del entorno de modelado construido y de los generadores de código.

8.2 Esquema general

Para implementar las RIAs derivadas desde los modelos ya formalizados, se utilizaron diferentes conjuntos de tecnologías para el client- y el server-side. En la Figura 51 se denota un esquema de la arquitectura conformada con el uso de estas tecnologías, en el cual se resumen las distintas capas de abstracción y componentes que implementarán las Aplicaciones Ricas derivadas. Los elementos en color naranja conforman el framework de dominio que se construyó específicamente para facilitar el trabajo de los derivadores de código. En la Figura también se muestran, a través de líneas punteadas, las fracciones del meta-modelo intervinientes en la derivación de distintos aspectos de cada uno de estos niveles de abstracción.

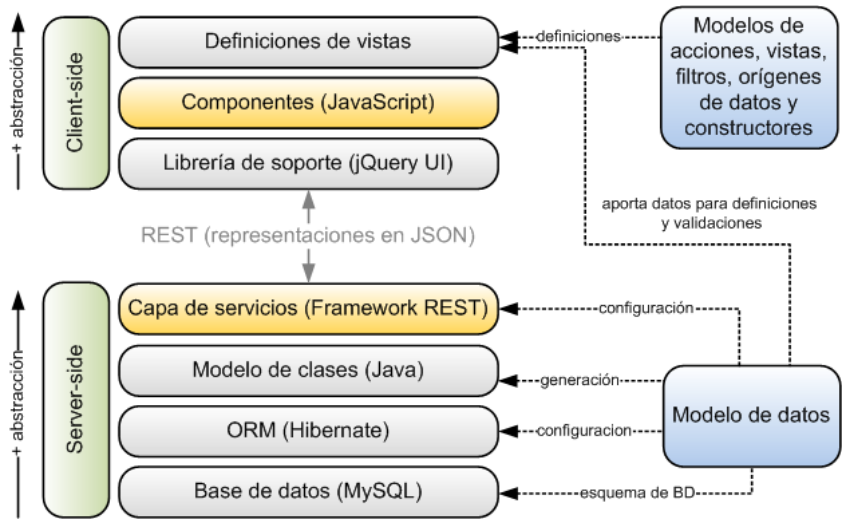


Figura 51: Esquematización de las capas de abstracción y componentes que conforman la arquitectura que se utilizará como base para implementar las RIAs generadas desde los distintos modelos. Las líneas punteadas indican en qué aspecto particular cada uno de los distintos tipos de modelos participa en la especificación de la Aplicación Rica final.

En lo respectivo a tecnología de base, para el server-side se utilizó MySQL como servidor de bases de datos y Java EE para la implementación de las capas superiores. Se deriva desde el modelo de datos un esquema de BD relacional y un conjunto de clases Java

las cuales se asocian a este esquema utilizando la tecnología de mapping objeto-relacional (ORM) Hibernate[31]. En un nivel superior, se implementa un framework que provee una capa de servicios RESTful el cual recibe las peticiones desde la faceta cliente y realiza la conversión entre objetos locales (Java) y remotos (enviados en una representación serial desde el server-side). Es esta capa de servicios quién se encargará de persistir y recuperar los objetos Java locales interactuando con la base de datos a través de la tecnología de ORM utilizada.

El client-side fue implementado en su totalidad en JavaScript, utilizando la librería jQuery y jQuery UI[32] para brindar la abstracción necesaria a nivel de navegador y para facilitar la programación de funcionalidades tales como detalles gráficos, animaciones y comunicación con el server-side. Se desarrollaron un conjunto de componentes cuyo comportamiento se ajusta a lo especificado en el meta-modelo, los cuales son instanciados usando una notación JSON[33] específica.

Los objetos de datos son transmitidos entre las facetas cliente y servidor también en formato JSON (en términos de REST, debería decirse que se utiliza JSON como representación por defecto para operar con los distintos recursos). El framework REST, de acuerdo a su configuración, convierte el contenido JSON en objetos Java y viceversa. Lo mismo realiza el client-side, llevando a cabo una conversión similar, tratando en este caso con objetos JavaScript.

8.3 Aspectos de implementación y derivación del server-side

En lo que a la faceta servidor respecta, el primer aspecto relevante a tratar en cuanto a derivación se refiere es la estructura de la base de datos. Como puede apreciarse en el esquema de la Figura 51, la misma es generada directamente desde el modelo de datos. En términos concretos, a partir de este último se deriva un script MySQL que construye las tablas y relaciones que representarán y estructurarán la información en el modelo relacional.

Cada *entidad* en el modelo de datos será representada en el esquema relacional a través de una tabla. Por cada *atributo* presente en la *entidad* en cuestión, se generará en la tabla correspondiente una columna de igual nombre y con tipo de dato compatible. Las *relaciones* entre *entidades*, de acuerdo a sus cardinalidades máximas, pueden o no requerir una tabla extra para ser representadas. Si se tratase de una *relación uno a uno* o *uno a muchos*, su representación puede lograrse a través de la inclusión de una clave foránea en una de las dos tablas asociadas a las *entidades* en relación. Por otro lado, si se tuviera una *relación muchos a muchos*, es necesaria la generación de una tercera tabla cuyo contenido consistirá en dos claves foráneas, las cuales indicarán en forma de pares ordenados las instancias asociadas en el contexto de la *relación* particular.

La generación de las clases Java que implementarán el modelo de objetos a través del cual serán representados en tiempo de ejecución los datos en el server-side se lleva a cabo de una manera similar. Por cada *entidad* se generará una clase y por cada *atributo* presente en la misma, una variable de instancia de tipo compatible y sus accessors correspondientes. Luego, por cada una de las *relaciones* en las que la *entidad* particular

31 https://www.hibernate.org/
32 http://jquery.com/
33 JavaScript Object Notation - http://www.json.org/

participe, se derivará en la clase que representa a ésta última una variable de instancia extra y sus accessors inherentes. Estos últimos permitirán obtener y establecer los objetos con los que la instancia se asocia a través de la *relación* específica. De acuerdo a la cardinalidad de la *entidad* en el contexto de la *relación* en cuestión, el tipo de la variable de instancia consistirá en una clase concreta o bien en una colección Java.

La configuración del ORM Hibernate es generada de manera acorde al esquema de base de datos y conjunto de clases Java derivado con anterioridad, de manera que la asociación entre el modelo de objetos y el modelo relacional sea correcta y consistente.

El framework REST requiere para su correcto funcionamiento información básica acerca de las clases y el modo de obtener, manipular y persistir sus instancias. El mismo fue adaptado para que funcione con Hibernate y en su configuración debe indicarse qué clases serán accesibles desde la capa de servicios y cómo se manejarán las relaciones recíprocas entre instancias de clases relacionadas. Esta configuración será provista programáticamente en un método particular de una clase Java, el cual funcionará además como programa principal para iniciar la ejecución del server-side. Resta aclarar que la capa de servicios RESTful fue implementada utilizando el servidor HTTP Jetty[34], el cual además es utilizado como repositorio de archivos que almacenará y permitirá el acceso a las distintas especificaciones de *vistas* por parte del client-side.

8.4 Aspectos de implementación y derivación del client-side

Como ya fue comentado, la implementación del framework de dominio en lo respectivo al client-side es desarrollada en el lenguaje JavaScript utilizando las librerías jQuery y jQuery UI, las cuales proveen abstracción a nivel de navegador y de aspectos relacionados con la presentación, interactividad y comunicación con el server-side. Haciendo uso de la facilidades de extensibilidad de estas librerías, se generan un conjunto de *widgets* de interfaz de usuario los cuales se adaptan a la semántica establecida por los componentes de UI descriptos en el meta-modelo.

Se definió adicionalmente un lenguaje de dominio específico basado en la notación JSON, a través del cual se declaran las *vistas*, sus componentes y su comportamiento inherente. La estructura de una *vista* y sus *widgets* especificados a partir de este lenguaje es interpretada en tiempo real en el client-side, generándose la interfaz de usuario bajo demanda. Esto facilita la transmisión de la UI a la faceta cliente y su creación dinámica en esta última.

Desde el punto de vista de la generación, por cada *vista* presente en el modelo de *vistas*, se derivará un archivo JavaScript conteniendo la especificación de la misma en el lenguaje de dominio específico planteado.

El correcto funcionamiento del framework de dominio client-side fue testeado con los siguientes browsers: Mozilla Firefox 3.5.1, Opera 9.63, Internet Explorer 7 y Google Chrome 2.0. Finalmente, se resumen las características más relevantes implementadas por el mismo, ya sea de manera prototípica o completa:

- Caching de datos.

34 http://www.mortbay.org/jetty/

- Almacenamiento de modificaciones locales en los objetos de datos y actualización diferida de los mismos en el server-side, a petición del usuario. Actualización automática de componentes que referencien información sometida a cambios locales en el client-side.
- Detección automática de dependencias de datos entre componentes.
- Carga y construcción de *vistas* bajo demanda.
- Ejecución de solicitudes de datos al server-side. Almacenamiento temporario de datos obtenidos desde el servidor en la faceta cliente de acuerdo a las semánticas de prefetching de cada componente.

8.5 Implementación del meta-modelo, entorno de modelado y derivadores

Cada una de las 6 fracciones en las que se divide el meta-modelo (datos, vistas, orígenes de datos, constructores, acciones y filtros) fue especificada haciendo uso de la herramienta de meta-modelado Eclipse GMF y serializada en un archivo Ecore separado.

A partir de estos archivos, haciendo uso del entorno de meta-modelado, se generan una serie de plug-ins para el IDE Eclipse los cuales implementan los editores que asisten en la confección de instancias de cada una de las distintas fracciones del lenguaje RIADL. Como ya se comentó en la sección correspondiente, Eclipse GMF enriquece a los editores que el framework EMF permite definir, brindándoles representaciones gráficas a los conceptos que a través de este último pueden especificarse. En este caso particular, dada la complejidad del meta-modelo implementado y el razonable trabajo de configuración que Eclipse GMF requiere, habiéndose definido además ya la sintaxis concreta del lenguaje, se implementaron sólo los editores asistidos no gráficos para RIADL.

Las derivaciones a texto son implementadas en el lenguaje MOFScript a través de un conjunto de especificaciones de transformación, las cuales son almacenadas en archivos de texto y pueden ser ejecutadas directamente sobre cualquier modelo RIADL válido. A pesar de la conveniencia de utilizar el lenguaje OCL para especificar restricciones de validez que no puedan ser expresadas en el mismo Ecore, se utilizó MOFScript para llevar a cabo estas especificaciones. Esto permitió integrar el proceso de derivación con el de validación en una misma tarea y, además, reusar funciones comunes entre estas dos etapas. No obstante, este esquema corre con la desventaja de que, de utilizarse otro lenguaje de derivación a futuro, sería necesario especificar nuevamente las condiciones de validación en el mismo. La derivación de las restricciones de acceso a *vistas* y *roles de usuario*, al tener poca incidencia en lo que a representación de RIAs en sí concierne, no fue considerada, dejándose planteada como trabajo futuro.

9 TRABAJOS RELACIONADOS

En esta sección se mencionarán trabajos relacionados a lo propuesto en este documento, resaltando sus principales similitudes y diferencias.

9.1 Extensiones a WebML

A pesar de que WebML, como ya se ha mencionado, es un lenguaje orientado a la especificación en un alto nivel de abstracción de aplicaciones web tradicionales, en [19] se han propuesto extensiones al mismo las cuales permiten la definición de características inherentes a Rich Internet Applications. Para poder comentar las extensiones que en la mencionada publicación se proponen, se hace indispensable detallar algunos de los componentes básicos que conforman al lenguaje en su versión básica, y sus diferencias con el lenguaje aquí propuesto.

9.1.1 Breve introducción a WebML

WebML separa la especificación de aplicaciones web tradicionales en dos etapas: el *diseño de datos* y el *diseño de hipertexto*. El primero consiste en la organización de los objetos principales de información capturados durante el análisis de requerimientos en un esquema formal y coherente, definido utilizando lenguajes ya existentes y de uso cotidiano como, por ejemplo, modelos Entidad-Relación o UML. La etapa posterior, el *diseño de hipertexto*, consiste en una serie de esquemas de vistas del sitio web, las cuales harán referencia al esquema de datos definido en la fase anterior. En este sentido, el lenguaje que en este documento se propone es estructuralmente similar, dado que permite llevar a cabo el diseño de datos principalmente a través de los conceptos de *entidades* y *relaciones*, haciendo posible luego la definición de de la interfaz de usuario del sitio a través de componentes de tipo *vista*, los cuales harán referencia al esquema de datos anteriormente definido.

Las páginas definidas en el modelo de hipertexto están compuestas por componentes denominados *content units*. Los *data units* y *multidata units* son dos tipos particulares de *content units*, los cuales referencian a algunos de los atributos de una o muchas instancias de datos de una entidad particular, respectivamente. Su concepto es similar al planteado por las *listas* y *contextos* definidos en el meta-modelo que aquí se propone. Sin embargo, dado que su uso no está orientado a definir interfaces de usuario ricas, no proveen inicialmente características que permitan determinar el momento en que obtendrán los datos desde el server-side ni el grado de permanencia de los mismos en el client-side. Adicionalmente, WebML permite la especificación de predicados lógicos llamados *selectores*, los cuales pueden adjuntarse a *multidata units* con el objeto de filtrar aquellas instancias que desean mostrarse de una entidad particular en este tipo de componentes. El ingreso de datos por parte del usuario, tanto para generar nuevas instancias de datos como para parametrizar búsquedas es también considerado por el lenguaje, y es llevado a cabo a través de componentes denominados *entry units*. Los *multidata* y *entry units* son conceptualizados en el lenguaje propuesto a través de *filters* y *constructores*.

WebML provee vínculos gráficos para relacionar los componentes definidos en los

modelos. En términos generales, estos vínculos representan, o bien la transferencia de información desde un componente hacia otro, o bien el flujo de navegación entre páginas. Nuevamente, esta es otra característica común con el lenguaje detallado en este documento, dado que en el mismo se definen, a través de relaciones entre componentes, navegación y transferencia explícita de datos. Sin embargo, dado que RIADL concibe además aspectos de interacción rica a nivel de interfaz de usuario, también permite asociar, a través de vínculos gráficos, componentes de UI con acciones particulares, las cuales se ejecutarán frente a la ocurrencia de determinados eventos capturados en el client-side.

Finalmente, el lenguaje WebML da la posibilidad de especificar operaciones de manipulación de datos clásicas, como lo son las altas, bajas, modificaciones, eliminaciones y creación y destrucción de relaciones entre instancias. Estos aspectos son considerados en RIADL, aunque diseminados entre distintos componentes de acuerdo a las posibilidades de interacción y contexto de datos que los mismos posean, en relación a los patrones que les dan origen.

9.1.2 Extensiones a WebML

Las modificaciones propuestas a WebML en [19] consisten, en términos generales, en decorar los conceptos ya definidos en dicho lenguaje a fin de definir si sus características inherentes serán incluidas como parte de la lógica de la faceta cliente o de la faceta servidor.

Las clases o entidades definidas en el modelo de datos, inicialmente tratadas en el server-side, pueden ser ahora asociadas al client-side, indicando que las instancias relativas a las mismas serán almacenadas y manipuladas en la faceta cliente, y luego sincronizadas con la faceta servidor, de acuerdo a una granularidad particular especificada por el modelador. Adicionalmente, las clases o entidades pueden indicar si su almacenamiento en el cliente o servidor es permanente o sólo temporal.

En lo respectivo a la funcionalidad relativa al modelo de hipertexto, la idea planteada en [19] para las extensiones a WebML consiste en refinar la especificación de las tareas a ejecutar por la aplicación, determinando si las mismas serán efectuadas en el client- o en el server-side. Como limitación es necesario mencionar que, dado que el modelo de hipertexto depende fuertemente del modelo de datos subyacente desde el cual obtiene la información a ser mostrada en la interfaz de usuario, este último limita en ciertas ocasiones la posibilidad de optar entre comportamiento orientado al cliente o al servidor.

Una de las principales diferencias entre el lenguaje descripto en [19] y el desarrollado aquí es que en este último no es necesario especificar en el modelo de datos aspectos relativos a la separación entre client- y server-side. Estas características son definidas en las *vistas* y sus componentes, los cuales podrán optar entre distintos modos para precaptar y actualizar sus datos desde el server-side, de acuerdo a las necesidades de la *vista* particular.

9.2 RUX-Method

El método RUX o RUX-Method, es un método guiado por modelos que se focaliza en el enriquecimiento de la interfaz de usuario de aplicaciones web tradicionales, extendiendo su funcionalidad existente. Su objetivo es permitir especificar UIs multi-

modales, multi-plataforma, multi-dispositivo y multimediales para RIAs.[31]

RUX-Method utiliza el modelo de datos, la lógica de negocios y los aspectos de presentación de la aplicación web a ser enriquecida, proveyendo una nueva interfaz de usuario rica para la misma. Para esto, divide la especificación de la nueva UI en 3 niveles: *interfaz de usuario abstracta* (independiente del dispositivo), *interfaz de usuario concreta* (orientada a especificar la UI para un dispositivo o grupo específico de dispositivos particular) e *interfaz de usuario final* (representada por el código fuente de aplicación que la implementa). La *interfaz de usuario concreta*, a su vez, divide las especificaciones en tres subniveles distintos: *presentación espacial* (trata aspectos visuales de la UI), *presentación temporal* (especifica características de la UI asociadas a lapsos de tiempo concretos como, por ejemplo, animaciones) e *interacción* (define el comportamiento de la UI frente a eventos de usuario).

Cada uno de estos tres niveles en el método RUX tiene asociado una *biblioteca de componentes*, la cual contiene un conjunto único de *componentes de interfaz* que conformarán la UI en un nivel de abstracción particular. Las conversiones entre componentes pertenecientes a distintos niveles de abstracción son logradas a partir de transformaciones que las mismas bibliotecas definen.

El método RUX define dos etapas de adaptación. La primera, consiste en tomar la interfaz de usuario de una aplicación web tradicional en lo relativo a datos, navegación y, de ser posible, presentación, y convertirla en una *interfaz de usuario abstracta*. La segunda, consiste en adaptar la *interfaz de usuario abstracta* obtenida con el objeto de alcanzar una UI orientada a uno o más dispositivos particulares, asegurando el acceso por parte de la misma a la lógica de negocios original de la aplicación. Finalmente, el proceso es completado con la obtención automática de la *interfaz de usuario final* para una tecnología RIA particular. En la actualidad, RUX-Method permite enriquecer la UI de modelos planteados bajo los lenguajes WebML y UWE, y se está progresando en la definición de adaptaciones para los lenguajes OO-H y OOHDM[31].

La diferencia más notable entre el esquema planteado por el método RUX y el lenguaje RIADL es el grado de detalle con que el primero permite especificar la UI en relación a este último. Esta diferencia subyace en el hecho de que el lenguaje RIADL fue concebido para representar los aspectos primordiales en las RIAs, por lo que no considera la definición de aspectos detallados de la UI como, por ejemplo, *presentación temporal* o *espacial*. El proceso planteado relativo al meta-modelo que aquí se expone asume que estos aspectos de UI serán inferidos o fijados en los derivadores, de acuerdo a características y necesidades particulares del dominio de la aplicación. La segunda distinción destacable entre ambos enfoques es que RIADL no considera para la definición de RIAs, la adaptación de meta-modelos existentes orientados a la definición de aplicaciones web tradicionales, sino que propone sus propios componentes de lenguaje para definir Aplicaciones de Internet Ricas desde cero.

9.3 Refactorings de aplicaciones web tradicionales a RIAs

En [34] se propone un método basado en modelos para transformar sistemáticamente una aplicación web a una RIA a través de lo que se denomina *RIA refactorings.* Estos son definidos como un tipo de *Web Model Refactoring* o *WMR* (serie de

uno o más cambios aplicados al modelo de navegación o interfaz de una aplicación web con el objeto de mejorarla), orientados al diseño de interfaz gráfica de usuario. Como se comenta en la publicación, este tipo de *refactorings* permiten transformar iterativamente una aplicación web tradicional en una Rich Internet Application.

A diferencia de la solución aportada por propuestas como la de RUX-Method, las cuales toman como base una especificación web ya existente y llevan a cabo un trabajo de reingeniería completo a partir de las mismas, en este caso se propone la aplicación incremental de distintos *refactorings* sobre las descripciones de aplicaciones web tradicionales, los cuales transformarán éstas progresivamente adicionándole en cada incremento características inherentes a Rich Internet Applications. La utilidad de la propuesta planteada surge debido a que, en la actualidad, a pesar de las ventajas mencionadas en reiteradas ocasiones que las RIAs brindan frente a las aplicaciones web tradicionales, aún sigue siendo necesario proveer métodos que permitan hacer transiciones incrementales y reversibles desde estas últimas a Rich Internet Applications. Entre otras, dos causas posibles de esta necesidad pueden hallarse en el acostumbramiento por parte de los usuarios a las interfaces web tradicionales y al interés de las compañías u organizaciones en no perder su inversión inicial en aplicaciones web *legacy*.

En términos más detallados, los mencionados r*efactorings* son descriptos como composiciones de interfaces RIA y formalizados con *Abstract Data Views* (*ADV*s), objetos de interfaz compuestos los cuales representan el *look and feel* de elementos navegacionales existentes y permiten especificar aspectos de presentación de estos últimos, sin intervenir en su lógica de negocios. El método propuesto permite aplicar cada uno de estos *refactorings* a través de especificaciones de integración, las cuales hacen posible la introducción de los mismos de manera incremental y favorecen la separación de aspectos. Esto facilita, entre otras cosas, que los cambios llevados a cabo por cada r*efactoring* puedan deshacerse si la funcionalidad introducida no es aceptada por los usuarios y el hecho de que puedan definirse en forma de *templates* para ser reusados en distintas partes de la aplicación.

Aunque en la presente Tesina se propone desde cero un lenguaje directamente orientado a la definición de Rich Internet Applications, los aportes hechos en relación a *refactorings* y a su representación planteados en [34] pueden servir para capturar nuevos aspectos del dominio de las RIAs y enriquecer con estos el meta-modelo aquí propuesto. Asimismo, la capacidad de introducción de cambios incremental comentada puede ser de utilidad para el agregado de nuevas características a RIADL de manera no intrusiva.

9.4 Eventos distribuidos

Como ya se ha comentado, una de las características de interés que proveen las RIAs es la posibilidad de implementar distribución de eventos producidos entre clientes y en el mismo servidor, haciendo uso de alguna tecnología que facilite la comunicación desde el server- hacia el client-side.

En [33] se hace notar, según sus autores, la falta de conceptos orientados a modelar eventos distribuidos en las metodologías de modelado web actuales. Partiendo de la base planteada por el lenguaje WebML propiamente extendido como se menciona con anterioridad en esta sección, se propone enriquecerlo a fin de brindar la posibilidad de modelar distribución de eventos en Rich Internet Applications.

Bajo el esquema de distribución de eventos planteado, la faceta cliente de una RIA puede, además de disparar un evento a través de una invocación al server-side, recibir notificaciones desde este último, las cuales pueden haber sido generadas desde otro cliente *online* o bien desde el mismo servidor. En la publicación se destacan tres aspectos de interés a ser definidos en lo respectivo a distribución de eventos: la forma de determinar cuáles clientes tienen que ser notificados frente a la ocurrencia de un determinado evento, dónde se implementará la lógica que lleve a cabo esta tarea y el tipo de persistencia del evento enviado.

La solución propuesta consiste en extender el *modelo de datos* del lenguaje WebML permitiendo que puedan definirse una clase especial de entidades que definen eventos a ser disparados en la aplicación. De esta manera, las mismas operaciones básicas que pueden llevarse a cabo en el meta-modelo para manipular instancias de datos pueden utilizarse con instancias de eventos particulares. A su vez, el *modelo de hipertexto* es extendido brindando dos nuevos tipos de operaciones: envío y recepción de eventos. La primera dispara una nueva instancia de un evento y permite determinar uno o más clientes destinatarios del mismo. La segunda, funciona como receptora de eventos a través de la cual pueden especificarse una o más a acciones a ser ejecutadas frente a la instanciación de un nuevo evento de su tipo específico y que tiene a su cliente como destinatario. También pueden determinarse recepciones de eventos en el server-side, existiendo en este último caso la posibilidad de especificar acciones que solo pueden ser ejecutadas en esta faceta como, por ejemplo, aquellas relacionadas con persistencia de datos.

Finalmente, las funcionalidades planteadas en el artículo, si bien no fueron consideradas en RIADL, pueden ser implementadas de una manera similar a la comentada en el mismo extendiendo el *modelo de datos* y de *acciones* que en el lenguaje se definen.

10 CONCLUSIÓN

En el presente trabajo de grado se aplicó la metodología de Modelado Específico de Dominio en el área de las Rich Internet Applications orientadas a manejo intensivo de datos. Como resultado, se confeccionó un lenguaje de alto nivel de abstracción a través del cual las mismas pueden especificarse y se construyeron derivadores de código que permiten obtener, a partir de estas especificaciones, implementaciones totalmente funcionales.

El lenguaje aquí propuesto y todo lo relativo al ambiente de modelado implementado surge como resultado de la aplicación del proceso promovido por la metodología DSM en el dominio comentado. En primer lugar, se analizan aspectos teóricos, prácticos y tecnológicos de las Rich Internet Applications como su definición conceptual, características fundacionales, tecnologías que facilitan su especificación, patrones de diseño específicos en el área, entre otros. Posteriormente, se bosqueja una versión informal de un lenguaje que permita definirlas el cual luego es plasmado en un meta-modelo formal haciendo uso de una herramienta de meta-modelado particular. Con ésto se cumplen dos preceptos fundamentales que la metodología DSM promueve: el focalizar las especificaciones de software a un dominio particular y la construcción de un lenguaje de alto nivel de abstracción haciendo uso de herramientas de meta-modelado extensibles. En esta etapa, se obtiene como resultado un entorno que facilita la tarea de construir modelos con el lenguaje aquí propuesto.

Finalmente, se confecciona la derivación automática de la implementación final a través de la especificación de transformaciones *modelos-a-texto*. Éstas toman como entrada modelos RIADL y generan distintos artefactos de software como, por ejemplo, archivos de configuración, scripts y código fuente, etc., los cuales conformarán la aplicación ejecutable final. Asimismo, se desarrolla un framework de dominio con el objeto de incrementar el nivel de abstracción en las representaciones de esta última y así simplificar la complejidad de las transformaciones que permiten obtenerla. Con esto, se completa el proceso DSM.

Se espera que lo tratado en esta Tesina contribuya, en primer lugar, a comprender la metodología de Modelado Específico de Dominio, su aplicación, ventajas, restricciones, filosofía y motivación subyacente. En el caso particular aquí tratado se puede observar que, con un lenguaje de relativa complejidad y correctamente acotado a un dominio específico de interés, la cantidad de artefactos de software que pueden ser generados a partir de instancias del mismo es notable y, por ende, pueden llegar a obtenerse importantes incrementos en productividad. Por otro lado, en el presente trabajo de grado se intenta hacer un aporte en la captura y abstracción de las características fundamentales inherentes al campo de las Rich Internet Applications a través de la investigación llevada a cabo, características las cuales son resumidas en el meta-modelo formal propuesto.

11 TRABAJOS FUTUROS

Como se observó en distintas ocasiones al analizar extensiones propuestas a diferentes lenguajes de modelado web existentes, es posible el enriquecimiento de un meta-modelo ya definido a través de otro que permita decorar algunas de sus características. Esto provee un método viable para reusar tanto los conceptos del meta-modelo en sí como los modelos que se confeccionaron con el mismo. Siguiendo este mismo patrón, gran parte del trabajo futuro relativo a lo que en Tesina se desarrolla puede consistir en extensiones del lenguaje RIADL que contemplen aspectos que no han sido considerados al momento de su confección como, por ejemplo:

- Definición de eventos distribuidos, como se comenta en [33].
- Definición de tareas y flujos de trabajo, enriqueciendo el *meta-modelo de acciones* y agregando operaciones clásicas de control de flujo de acuerdo a parámetros provistos por los *meta-modelos de datos* y *de vistas*.
- Definición detallada de aspectos de presentación, de manera similar a la propuesta en [31].
- Enriquecimiento del *meta-modelo de datos*, proveyendo capacidad de herencia entre *entidades*, *atributos* derivados, etc.
- Adicionar operadores de mayor riqueza a los *filters*, de manera de poder especificar restricciones de filtrado más complejas (por ejemplo, cláusulas de existencia, agrupamiento, posibilidad de incluir atributos de instancias relacionadas, etc.).

Si bien el lenguaje aquí propuesto funciona como especificación de alto nivel para definir Rich Internet Applications centradas en datos, se estima que con un estudio más exhaustivo de los patrones presentes en éstas puede incrementarse aún más la abstracción alcanzada. Un ejemplo de este tipo de incrementos podría lograrse estableciendo la estructura general de interfaz de usuario que se desea generar en base a las *entidades* de datos definidas y las acciones que se buscan realizar con las mismas. Utilizando esta especificación, podrían derivarse las *vistas* y sus sub-componentes asociados en el lenguaje RIADL que permitan realizar las operaciones básicas de manipulación de datos. Luego, la aplicación final podría ser generada haciendo uso de los derivadores ya confeccionados.

Otro trabajo futuro a ser realizado sobre lo que en este documento se propone es un proceso de derivación más elaborado a fin de reducir la complejidad requerida en los generadores de código fuente o bien en el framework de dominio subyacente. Este tipo de proceso podría consistir en una o más transformaciones *modelo-a-modelo*, la cual resulte en sucesivos modelos de menor abstracción cada uno, hasta alcanzar descripciones más cercanas a la implementación o, inclusive, a la tecnología particular para la cual se derivará código ejecutable.

La confección de los editores gráficos asistidos que permitan apreciar la sintaxis concreta del lenguaje es también otra tarea que puede considerarse como trabajo futuro en lo que a implementación respecta. La complejidad inherente a este aspecto en la herramienta de meta-modelado elegida fue el motivo por el cual no fue llevada a cabo. También se deja

como tarea pendiente la derivación de las restricciones de acceso a *vistas* de acuerdo a *roles de usuario*, característica la cual no fue implementada dada la poca relación que guarda con los aspectos de las RIAs en sí.

12 APÉNDICE: EJEMPLO DE APLICACIÓN

12.1 Introducción

En este breve apéndice se mostrará el proceso de desarrollo de una aplicación real de dimensiones reducidas utilizando las herramientas confeccionadas. Se comenzará comentando el dominio del problema de software a tratar y definiendo los objetos de datos que considerará, prosiguiendo luego con el modelado asistido a través de los editores generados con la herramienta de meta-modelado utilizada. Finalmente, se mostrarán fracciones de la Rich Internet Application generada a partir de los modelos de alto nivel.

12.2 Modelo de dominio del problema a tratar

El software a implementar consistirá en una versión extendida del breve ejemplo de una aplicación para manejo de bibliotecas ya introducido en secciones anteriores. El enunciado del mismo será el siguiente:

> La aplicación permitirá la administración de una o más bibliotecas. Cada una de éstas agrupará una colección de libros. Todo libro poseerá un autor, pudiendo cada autor haber escrito uno o más de los libros presentes en cada biblioteca particular. A su vez, un libro tiene un género literario asignado, y un autor se caracteriza por escribir dentro de un conjunto concreto de estos géneros. Los casos de uso más comunes de la aplicación serán:
>
> 1. Mantenimiento de bibliotecas, autores y géneros (brindar operaciones que permitan crear, eliminar y modificar instancias de éstos).
> 2. Alta de libros, asignándoles su autor y género correspondientes.
> 3. Modificación de las propiedades básicas de un libro, así como también su género y autor.
> 4. Consulta de libros por título.
> 5. Obtener todos los libros de un género literario particular.
> 6. Obtener todos los autores que escriban bajo un género literario particular.

12.2.1 Modelo de datos

En la Figura 52 se muestra, a través de la sintaxis concreta ya descripta, un esquema del modelo de datos de la aplicación a implementar. En el mismo se incluyen las 2 *entidades* `Libro` y `Autor` ya introducidas, siendo la última enriquecida con un *atributo* `fotografia` de tipo `image`. En el modelo también se definen `Bibliotecas`, las cuales poseerán un `nombre` y podrán contener uno o más `Libros` (relación `enBiblioteca`). Todo `Libro` deberá pertenecer a una `Biblioteca` concreta, así como también tendrá que ser catalogado bajo un `Genero` literario particular (relación `generoDelLibro`). Finalmente, toda

instancia de `Autor` podrá catalogarse bajo uno o más de estos géneros (relación `generosDelAutor`).

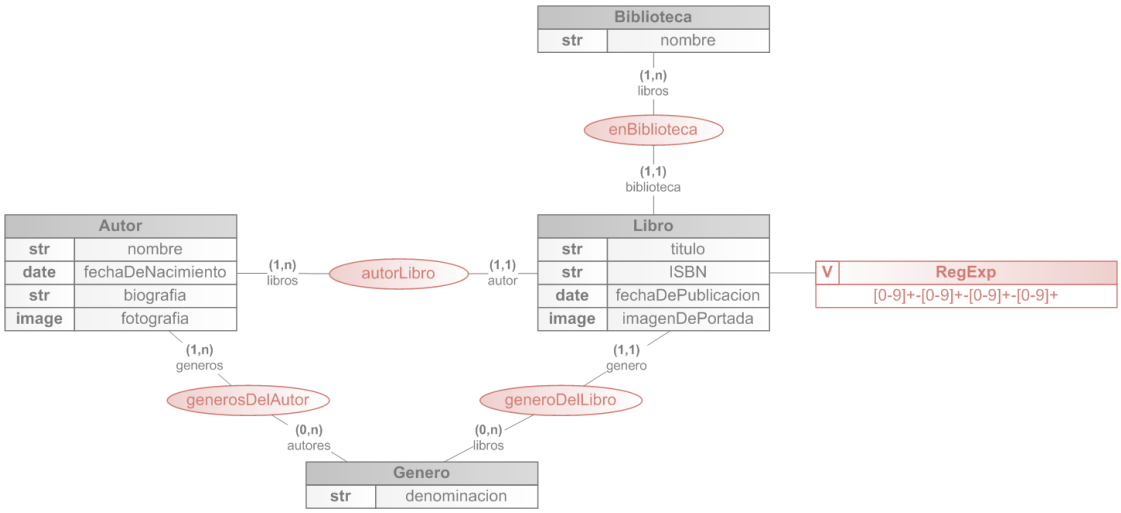


Figura 52. Modelo de datos de la aplicación de ejemplo a confeccionar

12.3 Confección de las vistas

En esta sub-sección se procederá a definir las *vistas* que implementarán cada uno de los casos de uso introducidos. Se partirá creando una *vista* principal a partir de la cual se accederá a las distintas funciones de la aplicación. Luego, cada caso de uso será especificado de la siguiente manera:

- El caso de uso (1) implica la especificación de *vistas* que permitan la instanciación, modificación y eliminación de `Biblioteca`s, `Autore`s y `Genero`s. Para cada una de estas *entidades* se generarán 2 *vistas*: una, la cual definirá un *constructor* que permitirá la creación de instancias y otra, la cual incluirá una *lista* que hará posible la modificación o eliminación de instancias existentes.
- El caso de uso (2) será implementado en una *vista* aparte, la cual contendrá un *constructor* que permita la creación de `Libro`s. El mismo deberá contener 3 *listas* a través de las cuales puedan seleccionarse las instancias de `Genero`, `Autor` y `Biblioteca` a asociar obligatoriamente al `Libro` en cuestión.
- El caso de uso (3) se especificará en dos *vistas*. La primera permitirá observar el `Autor` relacionado al `Libro` de interés y mostrará una *lista* de `Autore`s a partir de la cual podrá asignarse uno nuevo. La segunda será análoga a la primera, solo que orientada al `Genero` del `Libro` particular. En ambos casos, la instancia de `Libro` en cuestión será provista a través de un *parámetro* de *vista*.
- El caso de uso (4) se implementará en una *vista* aparte, la cual contendrá una *lista* de `Libro`s y un *filtro* asociado a la misma. Este último especificará un componente *textbox* el cual permitirá determinar el título del `Libro` a buscar.

La *vista* definida para satisfacer este caso de uso permitirá seleccionar un `Libro` particular en la *lista* ya mencionada y contendrá un link para cada una de las dos *vistas* definidas en el caso de uso (3), a partir de los cuales las mismas podrán ser invocadas. En ambos casos, se le proveerá a la *vista* a mostrar el `Libro` seleccionado como *parámetro*. Asimismo, se posibilitará el efectuar modificaciones sobre las propiedades de los `Libro`s listados a través de los componentes definidos en el interior de la *lista*, cumpliendo de esta manera con el requerimiento planteado en (3).

- Los casos de uso (5) y (6) serán implementados a través de la especificación de 3 *vistas* distintas. Una de ellas contendrá una *lista* de `Genero`s y permitirá seleccionar uno en particular. Las otras dos serán invocadas por la primera y mostrarán los `Autor`es y `Libro`s asociados al `Genero` seleccionado, el cual será provisto en forma de *parámetro* por ésta.

En la Figura 53 se denota un esquema de *vistas* el cual describe la implementación de parte de los requerimientos planteados en los casos de uso (3) y (4). La *vista* principal de la aplicación a través de la cual podrán accederse a todas sus funcionalidades es denominada `Home`. A través del link `Buscar Libro` presente en la misma puede navegarse hacia la *vista* homónima, la cual facilitará la búsqueda de `Libro`s por título. En este punto ya puede comenzar a apreciarse la especificación de características RIA a través del meta-modelo propuesto. En primer lugar, la *lista* de `Libro`s no realiza *prefetching* de datos, dada la cantidad de instancias de estos que pueden existir en la aplicación y la consecuente demora a la que conllevaría su carga. Asimismo, se habilita la sugerencia interactiva de títulos de `Libro`.

Una vez seleccionado un `Libro` en la *vista* `Buscar Libro`, a través del link `Modificar género` puede mostrarse (en modo *popup*) la *vista* homónima, la cual permitirá observar y cambiar el `Genero` asociado al `Libro` de interés. La *vista* `Modificar género` recibe a la instancia de `Libro` a través de un *parámetro*, el cual funcionará como origen de datos para un *context* declarado en su interior. Este último mostrará el `nombre` del `Genero` al cual el `Libro` se asocia. En la misma *vista* se provee una *lista* de `Genero`s la cual permite la selección de uno en particular, y un link que al ser activado producirá la asociación de éste con el `Libro` proveniente del ya mencionado *parámetro* y el posterior cierre de la *vista*. Finalmente, se provee un link denominado `Cancelar` a través del cual el usuario puede retornar a la *vista* anterior sin efectuar acción alguna.

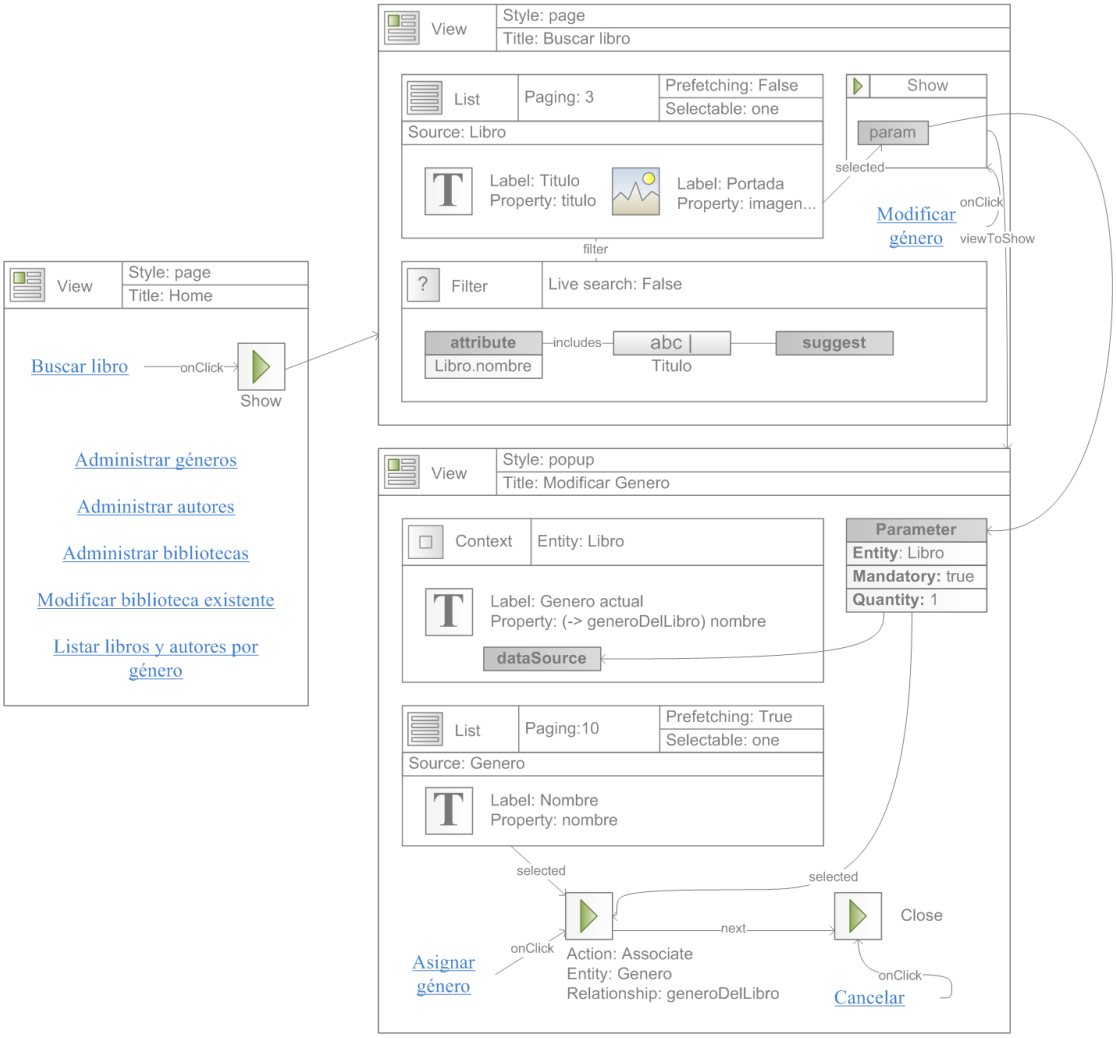


Figura 53. Esquema de vistas que especifican parte de los casos de uso (3) y (4)

12.4 Implementación final

Como última etapa del proceso, la aplicación final es obtenida automáticamente a partir de los modelos RIADL confeccionados, ejecutando sobre los mismos las transformaciones MOFScript definidas. En la Figura 54 se detalla una captura de pantalla en la que se puede observar el uso del entorno de modelado generado con Eclipse EMF, así como también algunos artefactos de software que han sido derivados desde los modelos especificados con el mismo. En la misma se detallan:

1. Los editores no-gráficos RIADL que permiten definir instancias del lenguaje, trabajando a nivel de su sintaxis abstracta.
2. Clases Java generadas.
3. Archivos JavaScript derivados que definen la *vistas* que conformarán la implementación client-side.
4. Archivos XML generados los cuales configuran el framework ORM Hibernate.
5. Definición de la clase Java `Autor`, derivada a partir de la *entidad*

homónima.

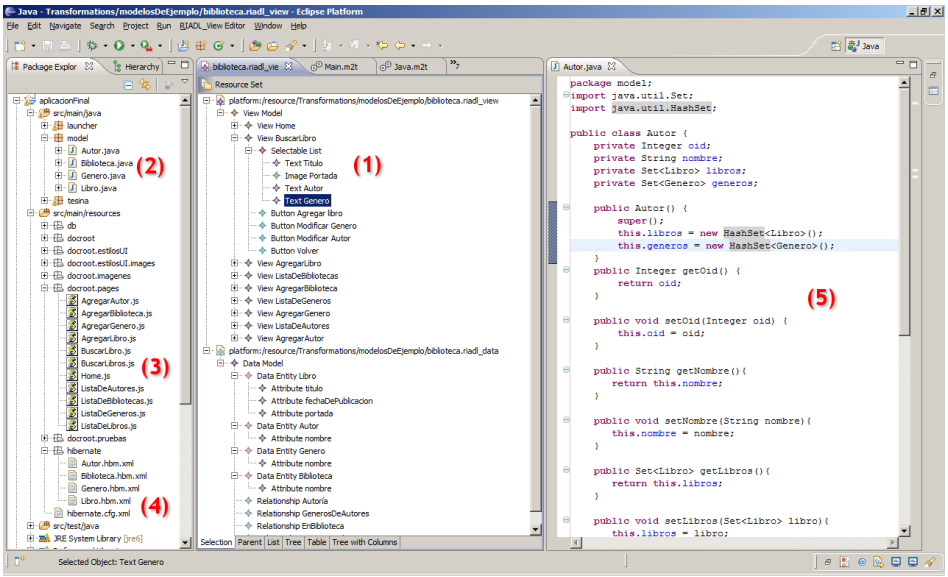


Figura 54. Entorno de modelado construido con Eclipse EMF y artefactos de software generados a partir de modelos RIADL

En la captura de pantalla mostrada en la Figura 55 puede apreciarse la visualización de la *vista* `Buscar Libro` en una versión corriente de la RIA derivada. En la misma se provee la *lista* de libros ya mencionada con capacidad de selección, así como también componentes de UI que permiten realizar la búsqueda por título de los mismos. Puede notarse también:

- La sugerencia interactiva del título de libro a buscar.
- La capacidad de paginación que la *lista* de libros ofrece.
- La posibilidad de modificar el título de los libros listados utilizando el mismo *widget* de UI a través del cual el mismo es mostrado.
- La inclusión de imágenes en la UI, las cuales fueron directamente modeladas como *atributos* de tipo `image` en el modelo de datos.

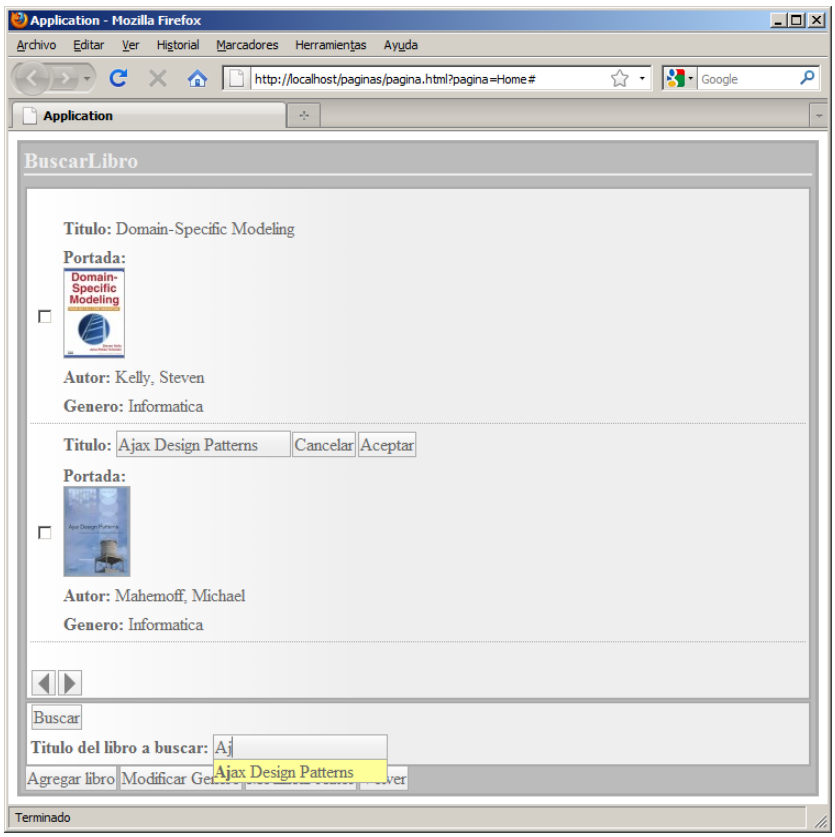


Figura 55. Vista `Buscar Libro` en la RIA generada directamente desde los modelos definidos en la captura mostrada en la Figura 54.

REFERENCIAS BIBLIOGRÁFICAS

[1] Domain Specific Modeling: Enabling Full Code Generation. Kelly, S., Tolvanen, J. Wiley-IEEE. 2008.

[2] Rich Internet Applications with Adobe Flex and Java. Yakov, F. SYS-CON Media. 2006.

[3] Ajax Design Patterns. Mahemoff, M. O'Reilly. 2006.

[4] Service-Oriented Architecture - Concepts, Technology and Design. Erl, T. Prentice Hall. 2005.

[5] OpenLaszlo Developer's Guide. URL: http://www.openlaszlo.org/lps4.1/docs/developers/. Accedido el 31 de Octubre de 2008.

[6] OpenLaszlo: An Open Architecture Framework for Advanced Ajax Applications. 2006.

[7] Patterns of Enterprise Applications Architecture. Fowler, M. Addison-Wesley. 2002.

[8] Design Patterns: Elements of Reusable Object-oriented Software. Gamma, E. Addison-Wesley. 1994.

[9] Model-Driven Generation of Web Applications in UWE. Koch, N., Kraus, A., Knapp, A. MDWE. 2007.

[10] Systematic hypermedia application design with OOHDM. Schwabe, D., Rossi, G., Barbosa, S. Proceedings of the the seventh ACM conference on Hypertext. 1996.

[11] Designing Web Applications with WebML and WebRatio. M. Brambilla, S. Comai, P. Fraternali, M. Matera. Springer. 2007.

[12] Applying Interaction Patterns: Towards a Model-Driven Approach for Rich Internet Applications Development. Valverde, F., Pastor, O. ICWE 2008 Workshops.

[13] RESTful Web Services. Richardson L., Ruby, S. O'Reilly. 2007.

[14] RIA Patterns. Best Practices for Common Patterns of Rich Interaction. Scott, B. URL: http://billwscott.com/share/presentations/2007/e7/RIA-Best-Practice-UI11WebApps.pdf. Accedido el 13 de abril de 2009.

[15] Web Services and Service-Oriented Architecture: The Savvy Manager's Guide. Barry, D. Morgan Kaufmann. 2003.

[16] Web Services Glossary. URL: http://www.w3.org/TR/ws-gloss/. Accedido el 2 de Febrero de 2009.

[17] Migrating Multi-page Web Applications to Single-page Ajax Interfaces. Mesbah. A., Van Deursen, A.

[18] On the Integration of Web Modeling Languages – Preliminary Results and Future

Challenges. Wimmer, M., Schauerhuber, A., Schwinger W., Kargl H. MDWE. 2007.

[19] Designing Rich Internet Applications with Web Engineering Methodologies. Preciado, J.C. Linaje, M. Comai, S. Sánchez-Figueroa, F. WSE. 2007.

[20] Macromedia Flash MX – A next-generation rich client. Allaire, J. 2002.

[21] Rich Internet Applications – IDC Opinion. Duhl, J. 2003. IDC.

[22] Eclipse GMF – http://www.eclipse.org/modeling/gmf/

[23] MetaEdit+ – http://www.metacase.com/

[24] GME 5 User's Manual. Institute for Software Integrated Systems. Vanderbilt University.

[25] Meta Object Facility (MOF) Core Specification (Version 2.0). URL: http://www.omg.org/docs/html/06-01-01/Output/06-01-01.htm. Accedido el 4 de marzo de 2009.

[26] MOFScript Home Page. URL: http://www.eclipse.org/gmt/mofscript/. Accedido el 4 de marzo de 2009.

[27] Eclipse Modeling Framework: A Developer's Guide. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R. Addison Wesley. 2003.

[28] From Extreme Programming to Extreme Non-programming: Is it the Right Time for Model Transformation Technologies?. Pastor, O. DEXA. 2006.

[29] Carousel Pattern – Yahoo! Design Pattern Library. URL: http://developer.yahoo.com/ypatterns/pattern.php?pattern=carousel. Accedido el 13 de abril de 2009.

[30] Necessity of methodologies to model Rich Internet Applications. Preciado J.C., Linaje M., Sánchez F., and Comai, S. IEEE Press. 2005.

[31] On the implementation of Multiplatform RIA User Interface Components. Linaje, M., Preciado, J.C., Morales-Chaparro, R., Sanchez-Figueroa, F.

[32] Object Constraint Language, OMG Available Specification, Version 2.0. OMG. URL: http://www.omg.org/cgi-bin/doc?formal/06-05-01.pdf. Accedido el 16 de julio de 2009

[33] Modeling data-intensive Rich Internet Applications with server push support. Carughi Toffetti, G. MDWE. 2007.

[34] Refactoring to Rich Internet Applications. A Model-Driven Approach. Rossi, G., Urbieta, M., Guinzburg, J., Distante, D., Garrido, A. ICWE. 2008.