UNIVERSIDAD NACIONAL DE CÓRDOBA
FACULTAD DE MATEMÁTICA, ASTRONOMÍA Y FÍSICA

Una Metodología Dirigida Por Modelos Para Desarrollo De Aplicaciones De Internet Ricas Basada En UML

Autor: Juan Pablo MANSOR
Supervisor: Juan E. DURÁN

Resumen

Debido al gran éxito de las aplicaciones de internet ricas (RIA), desde el año 2007 han aparecido numerosas metodologías dirigidas por modelos para el desarrollo de las RIA.

En general se ha estudiado muy poco la transición de requisitos a diseño para las aplicaciones RIA; solo encontramos 4 enfoques; de estos enfoques solo la metodología UWE automatiza (solo parcialmente) la transformación de requisitos a diseño. Además hay una escasez de metodologías RIA que contemple una notación de diseño que contemple componentes de todo tipo además de componentes de UI (una notación así juntaría varios intereses lo que permitirá comprender la aplicación RIA de manera unificada).

UML ha sido usado en la fase de ingeniería de requisitos en varios métodos web tradicionales y recientemente han surgido algunas propuestas de notaciones para describir requisitos para RIAs, basadas en UML. Entre las notaciones para describir casos de uso de UML preferimos los modelos de análisis porque: a) Los modelos de comunicación de análisis se parecen mucho a las descripciones textuales de los casos de uso. b) El patrón model-view-controller (MVC) ha sido propuesto como la base para describir arquitecturas y frameworks para RIA y los modelos de análisis se basan en conceptos muy parecidos a los de MVC. No encontramos ninguna metodología RIA que trabaje con modelos de análisis.

En este trabajo expresamos los requisitos en UML mediante diagramas de casos de uso y modelos de análisis para describir casos de uso. Además para las tecnologías Ajax, GWT, Flex, y Openlaszlo presentamos notaciones para arquitecturas de componentes. Abstrajimos estas 4 arquitecturas dependientes de tecnología mediante una notación de diseño de arquitectura basada en componentes UML que es independiente de tecnología. Además trabajamos con modelos de comunicación UML cuyos objetos son componentes del diseño de arquitectura basado en componentes. También, usando el paradigma de orientación a objetos definimos transformaciones de modelos de comunicación de análisis a arquitecturas de componentes y de modelos de comunicación de análisis a modelo de comunicación de componentes.

Summary

Due to the great success of Rich Internet Applications (RIA), since 2007 a lot of new model-driven methodologies have been created for the development of RIA.

In general, there are few studies about the transition from requirements to design for this type of applications; we have found only 4 approches; of those, only UWE methodology automatizes (partially) the transformation from requirements to design. Besides, there is a lack of RIA methodologies that consider a design notation that contemplates componentes of all kind besides UI components (such a notation would gather several interests, what would allow a unified understanding of a RIA application).

UML has been used in the requirements engineering phase in several traditional web methods and recently, some new proposals to describe RIA requirements based on UML have seen the light. Among the notations to describe UML use cases, we prefer analysis models because: a) the analysis communication models and textual descriptions are very much alike. b) The MVC pattern (MVC), has been proposed as base to describe RIA architectures and frameworks and analysis models are based on concepts very similar to those in the MVC pattern. We did not find any RIA methodology that works with analysis models.

In this work, we express the requirements in UML through use case diagrams and analysis models to discribe use cases. Besides, for Ajax, GWT, Flex and OpenLaszlo technologies, we present component architectures. We abstracted those 4 technology dependent architectures using an architecture design notation based on UML components, that is independent of the technology. Also, we worked with UML communication models whose objects are components of the architecture design based on UML components. Additionally, using the object-oriented paradigm, we defined transformations from analysis communication models to component architectures and to component communication models.

Índice general

Capítulo 1 - Introducción

En los últimos años las aplicaciones de internet ricas (RIA) se han tornado cada vez más útiles y necesarias, y ha ido creciendo su uso. Las aplicaciones RIA proveen interfaces con el usuario (UI) más interactivas y ricas, al estilo de las aplicaciones de escritorio; además para las aplicaciones RIA la ejecución del cliente se suele hacer en forma eficiente al no tener éste que bloquearse cada vez que hace un pedido al servidor web (para lograr esto se usan comunicaciones asincrónicas entre el cliente y el servidor web); además el procesamiento de pedidos del cliente por el servidor web suele ser más liviano, porque el servidor web no tiene que generar la UI de la respuesta a mostrar por el cliente (el servidor web simplemente manda el texto de la respuesta -en algún formato- y el cliente genera la UI para la misma y la inserta donde corresponda en la página web).
Debido a este éxito de las RIA, desde el año 2007 han surgido metodologías dirigidas por modelos para el desarrollo de este tipo de aplicaciones, a modo de ejemplo: metodología de Urbieta y Rossi [1], ADRIA [2], OOH4RIA [3], MARIA [4], RUX [5], UWE-R [6], OOWS 2.0 [7], metodologías basadas en WebML [8, 9], UWE [10], metodología de Casalánguida y Durán [11].
En general se ha estudiado muy poco la transición de requisitos a diseño para las aplicaciones RIA; solo encontramos los enfoques: UWE [10], MARIA [4], ADRIA [2], y el método de Casalánguida y Durán [11]. Salvo UWE ([10]) los demás enfoques no automatizan la transformación de requisitos a diseño; pero UWE considera solo parcialmente la transformación de requisitos a modelos de diseño que tienen que ver con la UI (modelos de estructura de navegación y modelos de presentación). Además hay una escasez de metodologías RIA que contemple alguna notación de diseño basada en componentes que integre componentes de todo tipo (una notación así juntará varios puntos de vista lo que permitirá comprender la aplicación RIA de manera unificada). Solo encontramos una notación de estas en el método OOH4RIA [3]; pero la misma tiene algunas limitaciones y dicho método no contempla la transición de requisitos a diseño. Por lo tanto sería útil contar con metodologías RIA que contemple un buen diseño de componentes (o en otras palabras una arquitectura de componentes). Otra ventaja de tener este tipo de modelos de componentes es que se los podrá mapear a plataformas específicas de RIA.
Dentro del área de ingeniería web el lenguaje más usado para ingeniería de requisitos es UML – ver [12] y [13] - en particular los diagramas de casos de uso y notaciones para describir casos de uso. UML ha sido usado en la fase de ingeniería de requisitos en los siguientes enfoques: WAE [14], WebML [15], OOWS [16], WebRE [17], UWE [18]. Ha habido algunas propuestas de notaciones para describir requisitos para RIAs basados en UML, ellas son: WebRE+ [19], y la propuesta de Casalánguida y Durán

[20].
En este trabajo decidimos apoyarnos en UML, particularmente en los diagramas de casos de uso y considerar la descripción de casos de uso por medio de alguna notación (semi-formal o formal) de UML y no simplemente una notación textual como se usa muy frecuentemente. Entre las notaciones para describir casos de uso preferimos los modelos de análisis (ver [14]) por los siguientes motivos:

- Los modelos de comunicación de análisis se parecen mucho a las descripciones textuales de los casos de uso; con las cuales los analistas/clientes ya están familiarizados.
- El patrón model-view-controller (MVC) ha sido propuesto como la base para describir arquitecturas para RIA (ver Chaparro [21] y Groove [22]); muchos frameworks para Web y RIA (ASP .Net MVC 2 [21], Rails [24], Apache Struts 2 framework [25], The-M-Project [24], Pure MVC [26], Model-Glue [27], etc.) se basan en MVC; y los modelos de análisis se los puede considerar como cercanos o parecidos a MVC (porque poseen objetos frontera - para entre otras cosas referirse a la interacción del usuario en la UI-, objetos de entidad - para manejo de información- y objetos de control - que poseen una interfaz de operaciones que les permite hacer de intermediarios entre los objetos de frontera y los objetos de entidad).

Por lo tanto la notación de los modelos de análisis consideramos que es la mejor entre las notaciones de UML para describir casos de uso y para hacer la transición a modelos de diseño de componentes (por su cercanía con MVC). En las otras notaciones para describir casos de uso que hemos encontrado, o no hay mucha cercanía con las descripciones textuales de casos de uso (statecharts – porque hay que definir estados) o se parecen bastante menos a MVC (diagramas de actividades, diagramas de secuencia de mensajes).
No encontramos ninguna metodología RIA en la literatura que trabaje con modelos de análisis. Solo encontramos un método (algo obsoleto) para las aplicaciones web convencionales Conallen llamado WAE [12]. El objetivo de este trabajo consiste en definir una metodología RIA que permita la transición de requisitos expresados en UML a diseño de componentes considerando tanto aspecto estático como dinámico. La transición entre requisitos y diseño deberá ser automatizada mediante el uso de transformaciones de modelos (sobre el tema de transformaciones de modelos en general ver [29, 30] y sobre enfoque MDA ver [29, 30]).
En este trabajo expresamos los requisitos en UML mediante diagramas de casos de uso y modelos de análisis para describir casos de uso. La forma de usar los modelos de análisis ha sido planteada de modo que se capturen distintos patrones de comunicación que son típicos de las aplicaciones RIA. Además definimos una notación de diseño de arquitectura basada en componentes UML que es independiente de tecnología de implementación RIA. Como visión dinámica a nivel de diseño consideramos modelos de comunicación UML donde los objetos son componentes del diseño de arquitectura basado en componentes. También definimos una transformación de modelos de comunicación de análisis a arquitectura de componentes (visión estática del diseño) y otra transformación de modelos de comunicación de análisis a modelo de comunicación de componentes; estas transformaciones están expresadas usando el paradigma de programación orientado a

objetos. Adicionalmente, para las tecnologías Ajax [33], GWT [34], Flex [35], y Openlaszlo [36] presentamos notaciones para arquitecturas de componentes. En este trabajo la notación de arquitectura independiente de tecnología que presentamos es una abstracción de estas arquitecturas dependientes de tecnología que definimos y además se basa en algunos patrones y esquemas de diseño conocidos.
El trabajo está organizado como sigue: El Capítulo 2 introduce los conceptos básicos en que nos apoyamos (principalmente conceptos de UML y de transformación de modelos); luego el capítulo 3 presenta un perfil UML que extiende los diagramas de componentes UML para modelos de diseño de arquitectura independiente de tecnología RIA; el capítulo 4 estudia cómo describir en general casos de uso de aplicaciones RIA mediante modelos de análisis; a continuación en el capítulo 5 se considera la descripción de patrones de comunicación para el dominio de las RIA, los cuales son expresados usando modelos de comunicación de análisis; en el capítulo 6 se presenta brevemente el proceso de desarrollo propuesto, que considera la transición de requisitos a diseño de arquitectura (en sus visiones tanto estática como dinámica); luego el capítulo 7 presenta una transformación de modelos de comunicación de análisis a diagramas de componentes de arquitectura RIA y una transformación de diagramas de comunicación de análisis a diagramas de comunicación de componentes; en el capítulo 9 se presentan perfiles UML para arquitecturas de componentes para 4 tecnologías RIA: Ajax, Flex, OpenLaszlo y GWT; finalmente se presenta un capítulo de conclusiones y trabajo futuro.

Capítulo 2 - Conceptos Básicos

2.1 Aplicaciones de Internet Enriquecidas

Como se menciona en [3], tradicionalmente, las aplicaciones web se construían de manera que la mayor parte del procesamiento y la lógica se realizaba en el servidor, y el navegador solo se utilizaba para mostrar contenido estático. Esto representaba serias limitaciones en la interactividad y el dinamismo de las interfaces de usuario. Este tipo de aplicaciones han sido reemplazadas gradualmente por RIAs (Rich Internet Applications, aplicaciones de internet enriquecidas, en español), las cuales proveen interfaces de usuario mucho mas interactivas, similares a las ofrecidas por las aplicaciones de escritorio.

Las RIAs proveen una nueva arquitectura que reduce el tráfico de la red significativamente, utilizando conexiones asincrónicas y consumiendo pequeñas cantidades de datos cuando estos son necesarios. Por otro lado, cambian el paradigma incluyendo la mayor parte del procesamiento en el cliente, dejando en el servidor la menor cantidad de lógica necesaria.

2.2 UML

De acuerdo con la definición dada en [37], UML es un lenguaje estándar de modelado para describir la estructura de una aplicación. Puede ser utilizado para visualizar, especificar, construir y documentar los artefactos de un sistema de software, desde un sistema de manejo de información hasta aplicaciones web distribuidas e incluso sistemas en tiempo real.

Como lenguaje, provee un vocabulario y reglas para combinar las palabras de ese vocabulario con el propósito de comunicar. Como lenguaje de modelado su vocabulario y reglas hacen foco en la represetación conceptual y física de un sistema.

Como lenguaje de especificación permite construir modelos precisos, sin ambigüedad y completos. En particular, UML contempla la especificación de todas las decisiones de análisis, diseño e implementación que deben ser tomadas en el desarrollo y despliegue de un sistema de software.

Como lenguaje para construir, los modelos expresados mediante UML pueden ser mapeados a lenguajes de programación tales como Java, C++ o Visual Basic, o incluso a tablas en una base de datos relacional u orientada a objetos.

Como lenguaje para documentar permite describir la arquitectura de la aplicación y todos sus detalles: Requerimientos, Arquitectura, Diseño, Código fuente, Tests, Prototipos, entre otros.

2.2.1 Casos de Uso

Como se indica en [37], ningún sistema existe en aislamiento. Todos ellos interactúan con actores humanos o automatizados que utilizan el sistema para algún propósito, y

esos actores esperan que el sistema se comporte de manera predecible.

Un caso de uso especifica el comportamiento de un sistema o parte de él describiendo un conjunto de secuencias de acciones, incluyendo variantes, que el sistema realiza para devolver al actor un resultado o valor observable. Se utilizan para capturar el comportamiento deseado de la aplicación en desarrollo, sin tener que indicar como se va a implementar ese comportamiento. Facilitan la comunicación entre los desarrolladores y los usuarios finales. Además, ayudan a validar la arquitectura y verificar el sistema a medida que evoluciona.

2.2.2 Diagramas de Casos de Uso

Tomando la definición de [37], estos diagramas son utilizados para modelar los aspectos dinámicos de los sistemas. Son una pieza central del modelado del comportamiento de un sistema o parte de él. Cada uno muestra un conjunto de casos y actores y como interactúan entre sí. Permiten visualizar, especificar y documentar el comportamiento de un elemento del sistema en un contexto determinado.

De acuerdo con [37], definimos los siguientes elementos.

Actor: un actor representa un rol que puede ser tomado por una persona, proceso o lo que sea que interactue con el sistema.

Relación <<extend>>: extiende el comportamiento de otro caso de uso sin que este dependa de él, es decir, permite definir una ruta opcional para el caso de uso extendido.

Relación <<include>>: Relación de dependencia entre dos casos de uso que denota la inclusión del comportamiento de un caso de uso en otro.

Relación de generalización: Indica que un caso de uso es una especialización de otro, es decir, indica de manera más concreta que se trata el caso de uso dando más detalles.

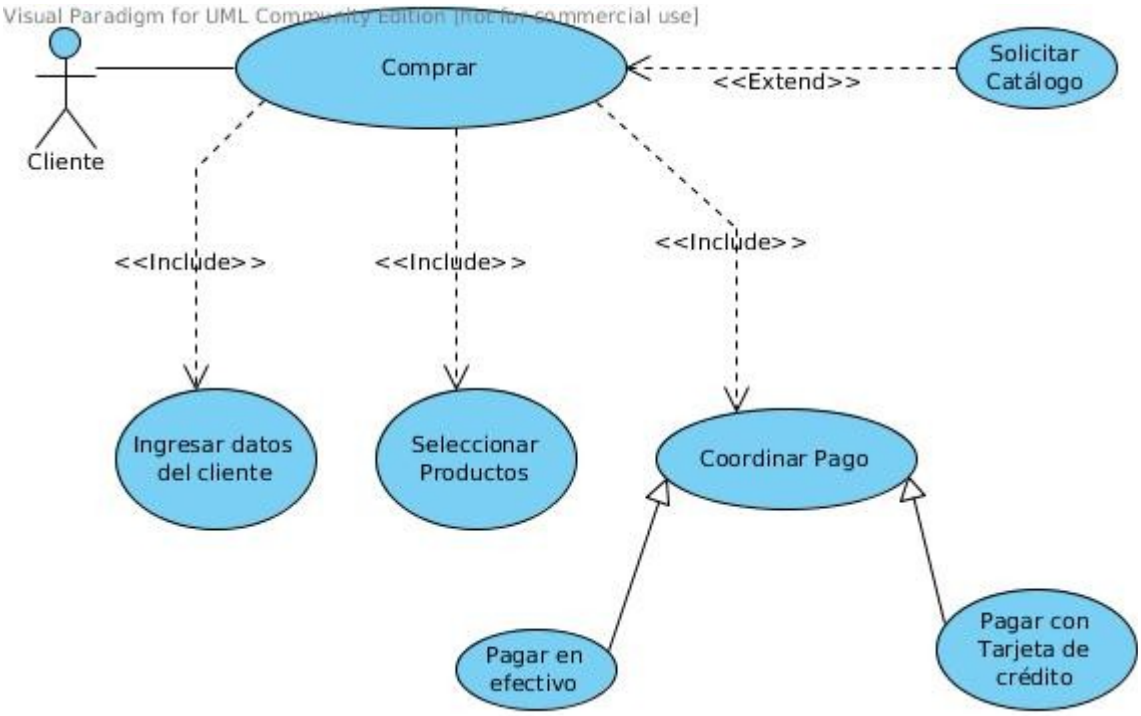

Figura 2-1: Ejemplo de diagrama de caso de uso

En el ejemplo de la Figura 2-1 podemos ver representados los diferentes tipos de relaciones utilizados en los diagramas de casos de uso. El ejemplo describe un escenario en el cual un cliente realiza la compra de algún artículo en una tienda online. Esto es representado por el caso de uso *Comprar.* La relación <<include>> permite expresar como este caso de uso está compuesto por otros tres casos de uso, que permiten al cliente ingresar sus datos, seleccionar los productos que desea comprar y coordinar el pago. Ya que el pago se puede realizar por diferentes medios, podemos expresar a cada uno de ellos con casos de uso separados, *Pagar en efectivo* y *Pagar con Tarjeta de Crédito* que se conectan con el caso de uso *Coordinar Pago* mediante una relación de generalización, ya que son especializaciones de este último. Por último, tenemos el caso de uso *Solicitar Catálogo*, unido al caso de uso Comprar mediante una relación de tipo <<extend>>, para reflejar que el usuario puede decidir si solicita o no que le envíen un catálogo de productos como parte de la compra.

2.2.3 Diagramas de Clase

De acuerdo con la definición dada en [37], un diagrama de clases muestra un conjunto de clases, interfaces y colaboraciones y la relación entre todos estos. Son utilizados para modelar la vista estática del diseño de un sistema. A grandes rasgos esto incluye el modelado del vocabulario, las colaboraciones o los esquemas. Además sirve de base para otros diagramas como el diagrama de componentes. A continuación, explicaremos brevemente cada uno de los elementos que componen a este diagrama.

2.2.3.1 Clases

Según [37], una clase describe un conjunto de objetos que comparten los mismos

atributos, operaciones, relaciones y semánticas. Una clase implementa una o más interfaces.

Son utilizadas para capturar el vocabulario del sistema que se pretende desarrollar. Estas clases pueden incluir abstracciones que son parte del dominio del problema como así también clases que forman parte de la implementación. Con ellas es posible representar elementos de software, hardware e incluso conceptos.

2.2.3.2 Relaciones

En el modelado orientado a objetos, hay cinco tipos de relaciones especialmente importantes (ver [37]):

- Dependencia: indica una relación semántica entre dos o más clases, de tal manera que un cambio en el elemento del cual se depende puede impactar en la semántica del objeto que lo declara como dependencia. Existen varios tipos de dependencias. En este trabajo utilizaremos especialmente la dependencia de tipo <<use>>, que indica que un elemento requiere de la presencia de otro para su correcto funcionamiento. En la Figura 2-2 puede verse expresado como una Persona necesita de un Pulmón para respirar.

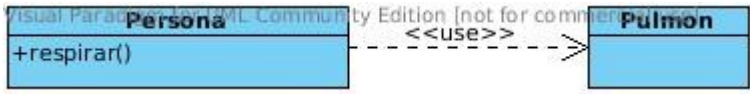

Figura 2-2: Ejemplo de relación de dependencia en un diagrama de clases.

- Generalización o Herencia: es una relación padre/hijo entre dos clases, donde la clase hija hereda todos los atributos, miembros y relaciones de la clase padre y además define información adicional (atributos, miembros y relaciones propios). La Figura 2-3 muestra un ejemplo de este tipo de relación. Se puede calcular la superficie tanto de un rectángulo como de un círculo, pero ambos tienen atributos diferentes.

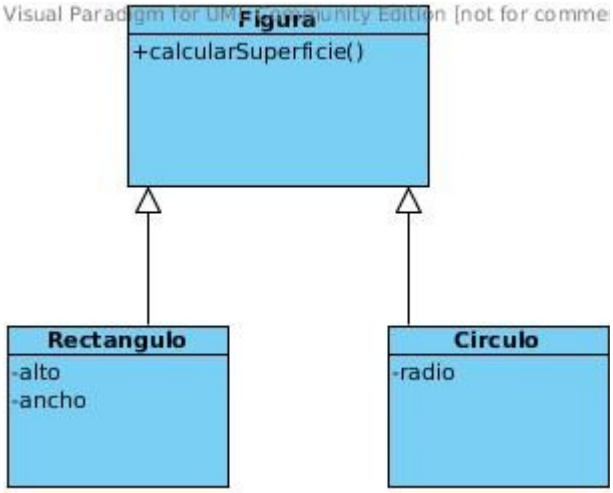

Figura 2-3: Ejemplo de relación de herencia en un diagrama de clases.

- Asociación: describe conexiones concretas entre objetos u otras instancias en un sistema, permitiendo también especificar la multiplicidad de la relación

(uno a uno, uno a muchos, muchos a muchos). Se representa con una línea continua. La multiplicidad se indicar al extremo del conector, con un número o un asterisco para indicar "cero o más". La Figura 2-4 muestra una relación de este tipo en la que un Maestro puede tener cero o más alumnos.

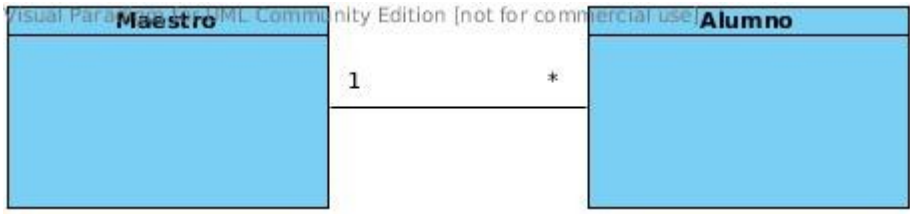

Figura 2-4: Ejemplo de asociación en un diagrama de clases.

- Agregación: es un tipo de asociación que permite expresar una relación *parte/todo* donde la *parte* puede sobrevivir independientemente del *todo.* La Figura 2-5 describe un escenario donde una rueda es parte de un automóvil, pero la rueda puede sobrevivir por si misma, sin necesidad del automóvil.

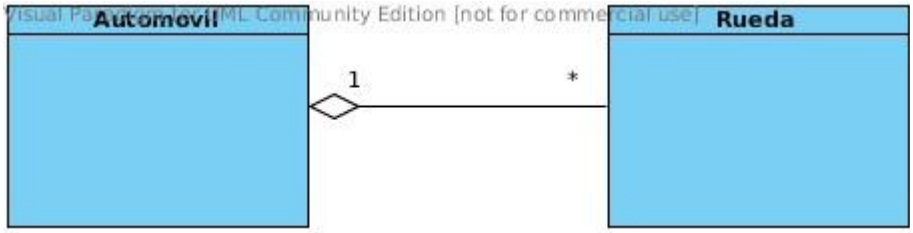

Figura 2-5: Ejemplo de agregación en un diagrama de clases.

- Composición: es un tipo de asociación que permite expresar una relación *parte/todo* donde la *parte* no puede sobrevivir independientemente del *todo.* La Figura 2-6 describe un escenario donde un Edificio tiene uno o más departamentos, y al mismo tiempo los departamentos dependen de la existencia del edificio, es decir, si el edificio es destruido también lo son los departamentos.

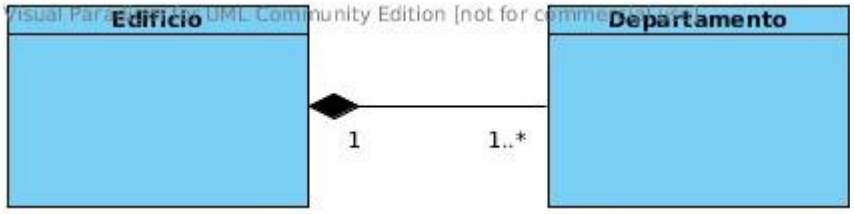

Figura 2-6: Ejemplo de composición en un diagrama de clases.

Cada una de estas relaciones provee una manera diferente de combinar abstracciones.

2.2.4 Diagramas de Componente

De acuerdo con la definición dada en [38], un componente es una parte encapsulada, reutilizable y reemplazable de un software. Se puede pensarlos como bloques de construcción: se combinan para formar el sistema. Normalmente se utilizan para modelar los elementos que llevarán a cabo las tareas principales dentro de la

aplicación.

En UML, un componente se comporta de manera similar a una clase: puede generalizarse y asociarse con otros componentes. La mayor diferencia es que por lo general, los componentes tienen más responsabilidad que una clase.

Un diagrama de componentes muestra los componentes en un sistema, como así también las dependencias entre ellos de manera de poder predecir el impacto de un cambio en la estructura. Cada componente expone una o más interfaces. Para cada componente se indica que interfaces provee, y que interfaces requiere. Una interfaz se define como una declaración de un conjunto coherente de funcionalidades públicas y obligatorias. Puede entenderse como un contrato entre proveedores de una funcionalidad y los consumidores de la misma.

2.2.5 Diagramas de Comunicación

Un diagrama de comunicación modela la interacción entre objetos en términos de secuencias de mensajes para describir el comportamiento dinámico del sistema. Los mensajes deben tener un nombre y deben ser numerados cronológicamente. Suponiendo que A, B y C son objetos diferentes y m1, m2 son mensajes dentro del diagrama de comunicación, entonces si A envía m1 a B y esto provoca que B envíe m2 a C, entonces se dice que m1 y m2 son anidados. Los mensajes anidados utilizan numeración anidada, con lo cual en el ejemplo anterior, si m1 tiene el número 1 entonces m2 tiene el número 1.1.

Opcionalmente los mensajes pueden tener asociada una guarda o iteración. Además, también permite expresar la ejecución de mensajes concurrentes. Las siguientes definiciones fueron tomadas de [39].

2.2.5.1 Objetos

Son una instancia de una clase. Una entidad discreta con un alcance bien definido que encapsula estado y comportamiento.

2.2.5.2 Mensajes

Representan la comunicación de un objeto con otro. Puede ser una señal o un llamado a una operación.

2.2.5.3 Guardas

Representan una condición la cual debe cumplirse para que un objeto envíe un determinado mensaje a otro.

2.2.5.4 Iteraciones

Representan la repetición de la ejecución de un mensaje hasta que la condición indicada deje de ser cierta.

Las Figuras 2-7 hasta 2-10 muestran los diferentes ejemplos de mensajes que pueden aparecer en este tipo de diagramas.

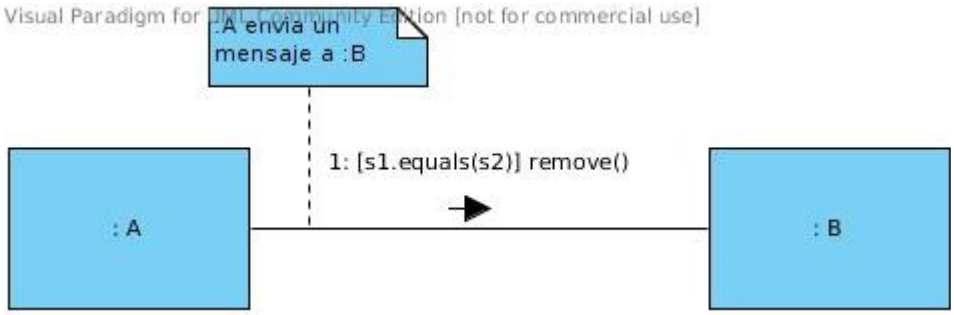

Figura 2-7: Ejemplo de mensaje con guarda en un diagrama de comunicación. :A llama a remove() en :B sólo si s1.equals(s2)

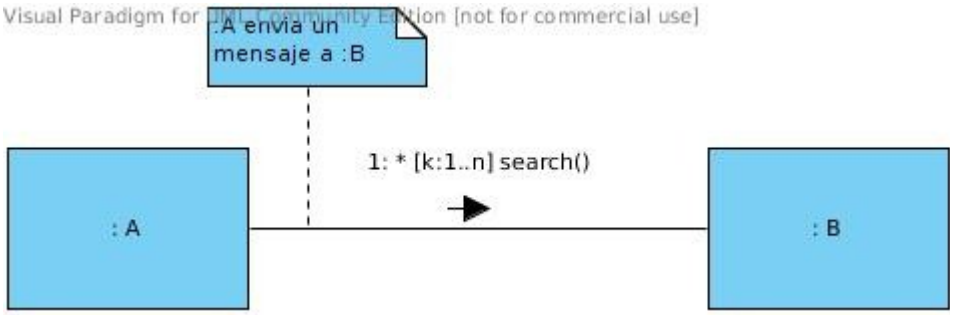

Figura 2-8: Ejemplo de iteracion de mensajes en un diagrama de comunicación. :A llama a search() en :B n veces.

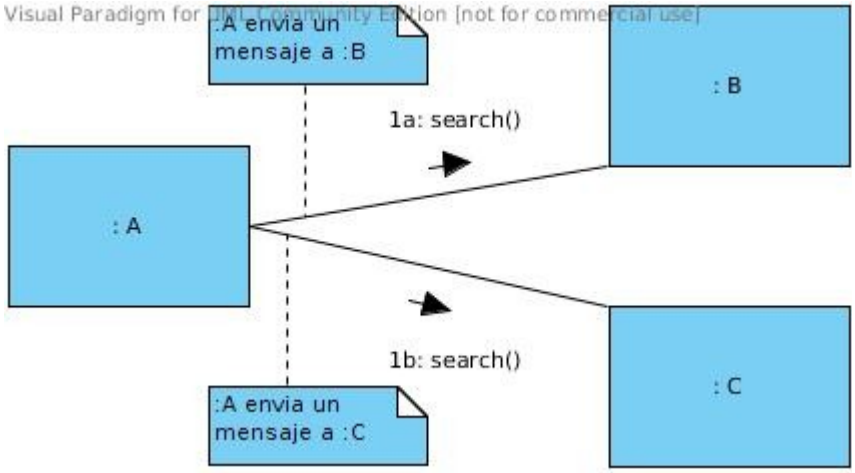

Figura 2-9: Ejemplo de mensajes concurrentes en un diagrama de comunicación. :A llama a search() en :B y en :C concurrentemente.

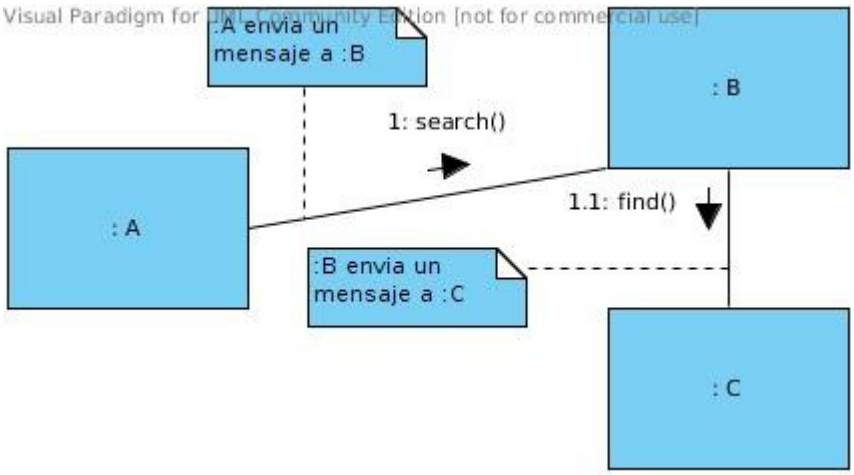

Figura 2-10: Ejemplo de mensajes anidados en un diagrama de comunicación. :A llama a search() en :B y :B llama a find() en :C.

2.2.6 Perfiles UML

UML se define mediante un metamodelo, es decir, un modelo del propio lenguaje de modelado. Los perfiles permiten extender UML sin necesidad de modificar el metamodelo, y adaptarlo a dominios específicos o plataformas manteniendo la interoperabilidad entre las herramientas. Una de las construcciones provistas por los perfiles son los estereotipos. Un estereotipo es un nuevo tipo de elemento de modelado agregado por el modelador y basado en algún tipo existente de elemento de modelado. A continuación damos las definiciones provistas por [39].

2.2.6.1 Estereotipos

El estereotipo es uno de los mecanismos que provee UML para extender el metamodelo. Permiten definir nuevos elementos de modelado más cercanos al dominio particular de aplicación que se quiere describirse.

Los estereotipos se aplican mediante un valor etiquetado. La notación utilizada puede ser o bien el nombre del esterotipo entre << y >> dentro de figura de elemento de modelado o bien solo un ícono para el estereotipo.

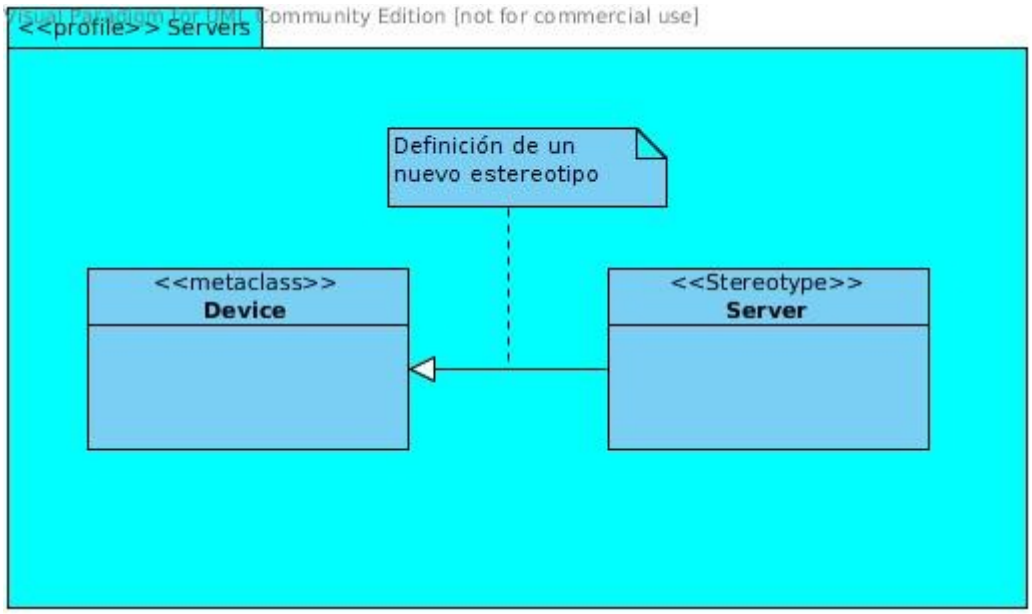

Figura 2-11: Definición de un nuevo estereotipo.

2.2.6.2 Valores Etiquetados

UML permite definir atributos para los estereotipos. Estos atributos se conocen como *etiquetas*. Cuando se aplica un estereotipo a un elemento del modelo, este obtiene dichos atributos. Para cada *etiqueta,* el modelador puede definir un valor, llamado *valor etiquetado.*

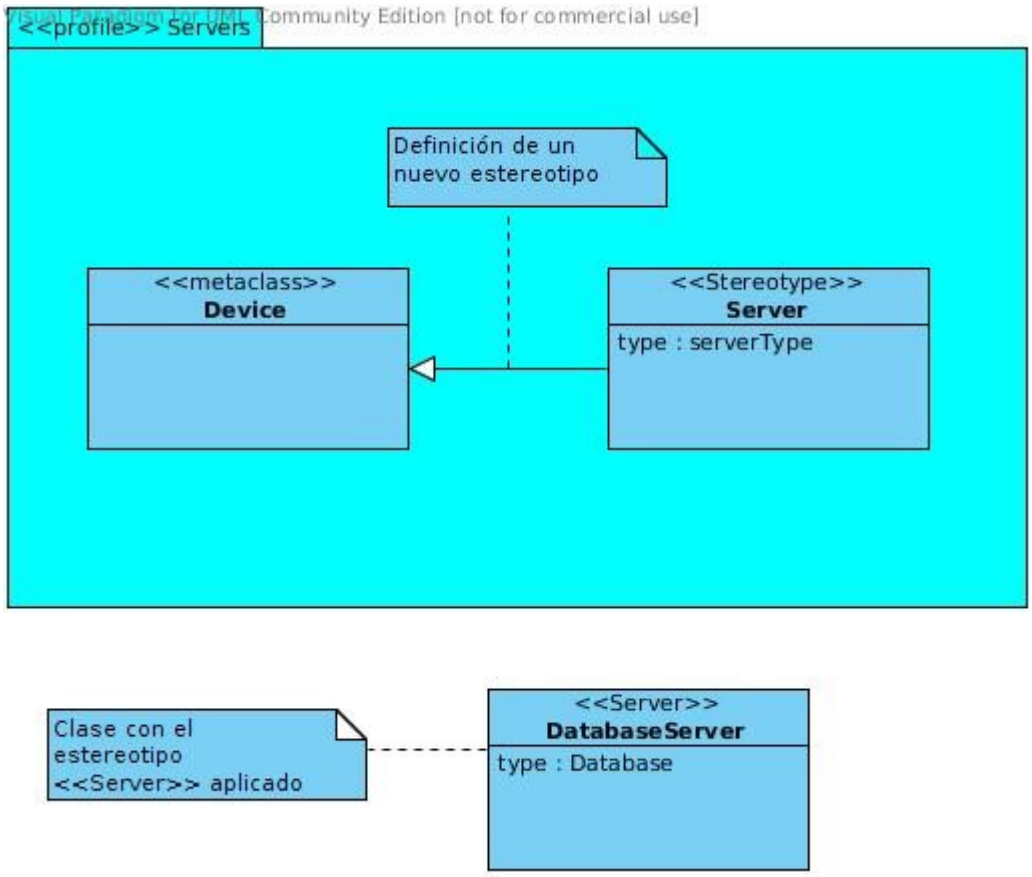

Figura 2-12: Ejemplo de aplicación de estereotipo y valor etiquetado.

2.3 Modelo de Análisis

El modelo de análisis representa el comportamiento de la aplicación a desarrollar a un nivel de abstracción mucho más alto que el de implementación física del sistema. Se deben determinar los objetos de software de la aplicación web. No hay una forma única de descomponer un sistema de aplicación en objetos. Dicho trabajo se basa en el juicio del analista y en las características del problema. Nosotros en este trabajo nos basamos en los modelos de análisis de Jacobson (ver [14]) que consiste de un perfil UML para las clases con los siguientes estereotipos: <<boundary>>, <<control>> y <<entity>>. Estos modelos de análisis tienen una parte estática expresada mediante un diagrama de clases y una parte dinámica usualmente expresada usando diagramas de comunicación UML. Ahora definimos los estereotipos mencionados:

- *Clases Boundary:* sirven para representar la interacción de la aplicación con diversos actores de su entorno (p.ej. usuarios, aplicaciones externas, dispositivos, temporizadores, etc.) Estas clases tienen estereotipo <<boundary>>. Un objeto frontera es una instancia de una clase frontera.
    - En el caso de las aplicaciones web un objeto frontera sirve para representar una parte de la interfaz de la aplicación web, ya sea con un tipo usuario o con un sistema externo. Un objeto frontera puede proveer la interfaz con un usuario humano vía dispositivos estándares de E/S; en una aplicación web,

estos objetos pueden ser vistos como agrupando páginas web o pantallas. También un objeto frontera puede interactuar con un sistema externo que se comunica con la aplicación web que se está desarrollando.

- *Clases de Control:* usan estereotipo <<control>> y se refieren a los procesos que entregan la funcionalidad del sistema. Las operaciones de las clases de control suelen representar funcionalidades a invocar. Un objeto de control representa la coordinación, secuenciación y control de otros objetos de análisis.
- *Objeto Entity:* es un objeto de larga vida que almacena información. Objetos de entidad son instancias de clases de entidad con estereotipo <<entity>>. Los objetos de entidad almacenan datos y permiten acceso limitado a los mismos vía operaciones que proveen. En algunos casos un objeto de entidad puede necesitar acceder a otros objetos de entidad para actualizar la información que encapsula. En muchas aplicaciones la información encapsulada por los objetos de entidad es almacenada en una base de datos.

La Figura 2-13 y 2-14 describen la recepción de una notificación de un nuevo precio para una acción, desde una aplicación externa. Esa notificación es procesada por un objeto de control que almacena la información del evento mediante un objeto entity y luego actualiza la interfaz del usuario, un objeto de tipo boundary, para reflejar el cambio acontecido.

La Figura 2-13 describe la vista estática mediante un diagrama de clases.

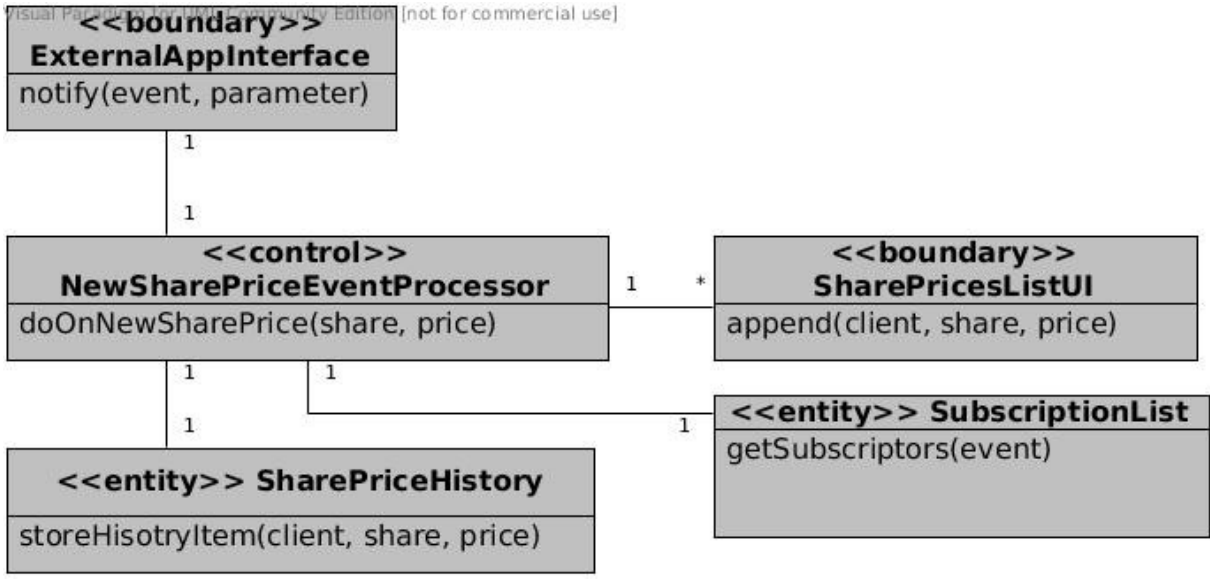

Figura 2-13: Ejemplo de modelo de análisis. Vista estática.

La Figura 2-14 describe la vista dinámica mediante una diagrama de comunicación.

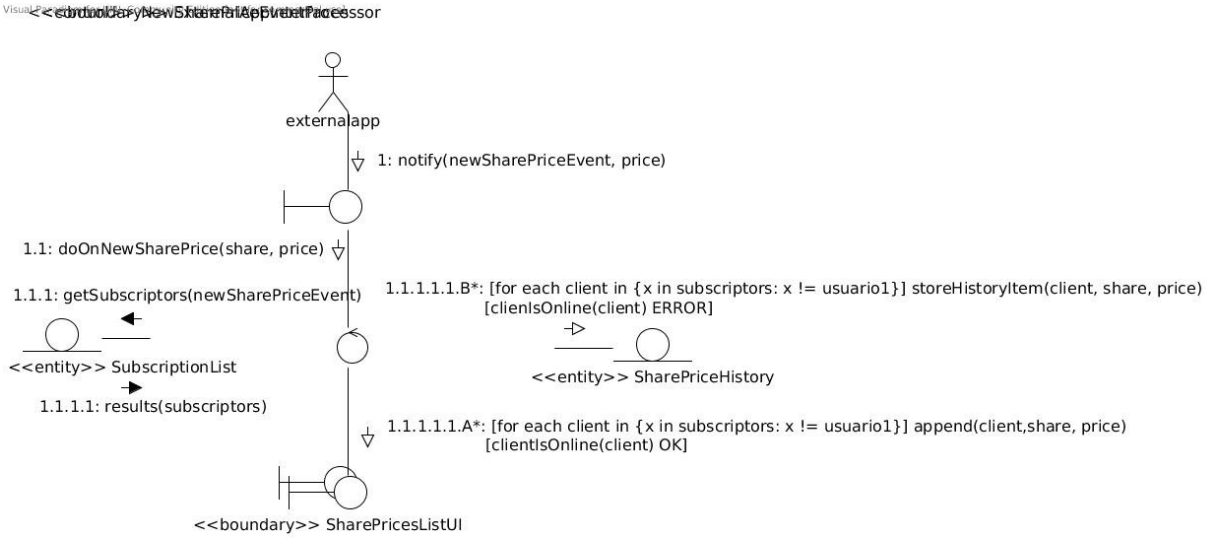

Figura 2-14: Ejemplo de modelo de análisis. Vista dinámica.

2.4 Marcas

En algunos casos, para aplicar una transformación no basta con el modelo de origen, sino que necesitamos tener entradas adicionales las cuales, entre otras cosas, pueden ayudar a decidir qué regla aplicar o cómo aplicar una regla. Estas entradas de mapeo adicional toman la forma de *marcas* (ver [32]), las cuales son livianas, extensiones no intrusivas a modelos que capturan la información requerida para modelar transformaciones sin poluir esos modelos.

Un mapeo puede usar varias marcas diferentes asociadas con los modelos de origen; recíprocamente, una marca puede encargarse de varios mapeos diferentes. Sin embargo, las marcas no deben estar integradas en el modelo de origen, porque son específicas al mapeo y varias reglas diferentes de mapeo pueden existir, cada una de las cuales requiere diferentes marcas. Se puede pensar en las marcas como un conjunto de notas adjuntadas a los elementos del modelo de origen, que dirigen el transformador de modelos.

2.5 Modelos de marcas

Como se explica en [32], una marca es definida por un *modelo de marcas*, el cual describe la estructura y semántica de un conjunto de tipos de marcas. Una función de mapeo especifica los modelos de marcas cuyos tipos de marcas requiere en las instancias de los metamodelos origen.

Si una función mapeo puede usar más de un modelo de marcas para un metamodelo fuente, entonces uno puede reutilizar modelos de marcas para varias funciones de mapeo.

Capítulo 3 - Perfil UML para Modelos de Arquitectura RIA Independientes de Tecnología

Un modelo de arquitectura para aplicaciones RIA se describe en este trabajo estáticamente mediante un diagrama de componentes y dinámicamente mediante un diagrama de comunicación que muestra el comportamiento de instancias de las componentes y cómo se comunican ellas entre sí. Mediante un perfil de UML, crearemos estereotipos apropiados para describir este tipo de aplicaciones. Con estos mismos mecanismos definiremos también los modelos de arquitectura de las tecnologías que son objeto de análisis en este trabajo.

Además, definimos tres paquetes en los cuales pueden usarse estos componentes: Cliente, Servidor y AdministradorEventosServidor. Cada uno de los componentes está destinado a ser utilizado en solo uno de estos paquetes. Así tenemos componentes para el paquete Cliente, otros para el paquete Servidor y otros más para el paquete AdministradorEventosServidor.

El paquete Cliente:

Este paquete representa al navegador web. Es el encargado de presentar la interfaz a los usuarios y responder a los eventos que estos generan durante su interacción.

El paquete Servidor:

Como su nombre lo indica, representa al servidor. En este paquete van los componentes que se encargan de atender los pedidos asincrónicos provenientes de los clientes y aquellos que se encargan de implementar la persistencia de los datos.

El paquete AdministradorEventosServidor:

Contiene los componentes encargados de manejar los eventos que comunican a clientes con otros clientes o aplicaciones externas. Se encarga tanto de permitir que estos se subscriban a eventos como de notificar a los subscriptores de la ocurrencia de un evento determinado.

En la Figura 3-1 muestra la definición del perfil UML para aplicaciones RIA.

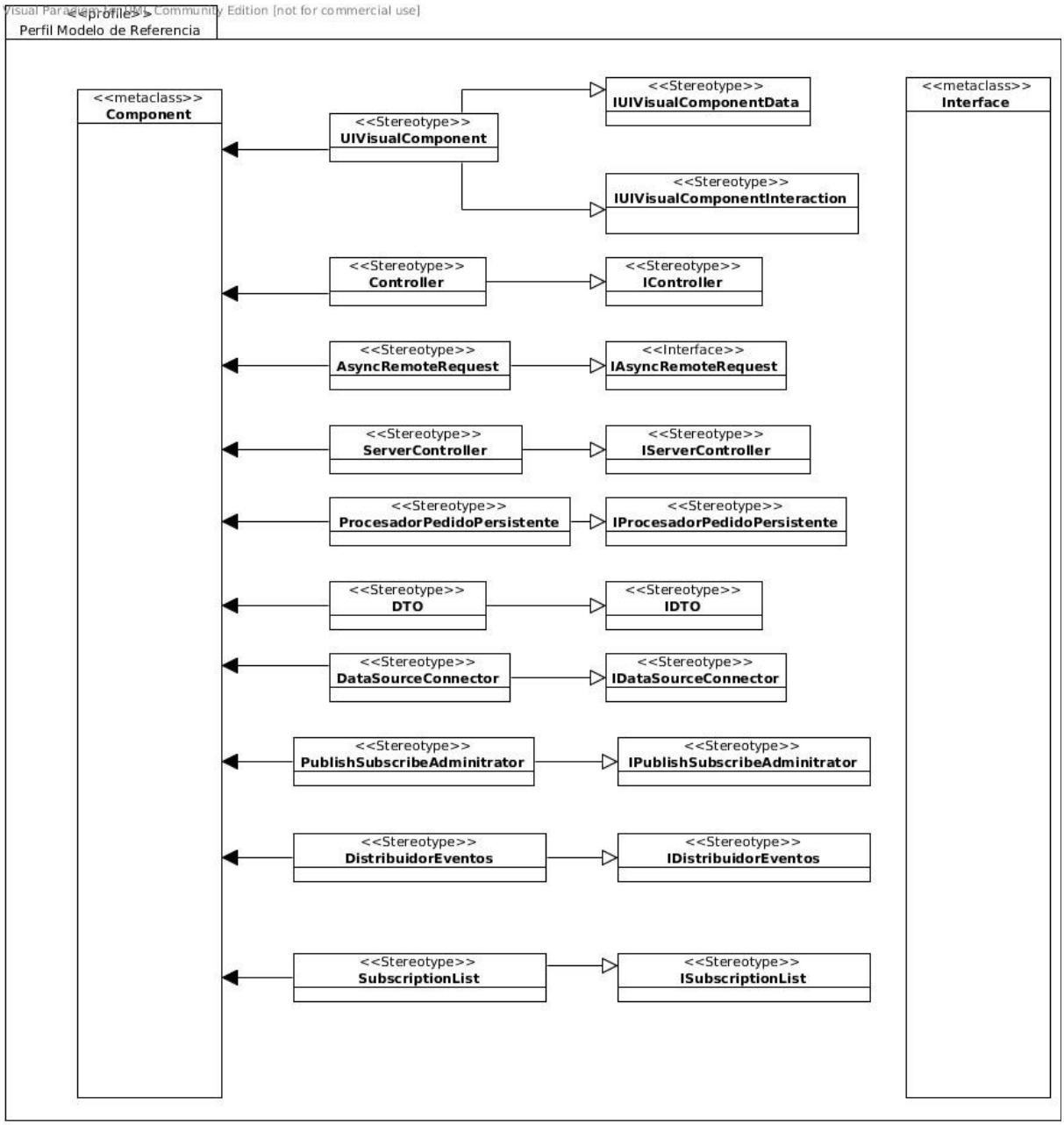

Figura 3-1: Ejemplo de perfil UML utilizado en este trabajo.

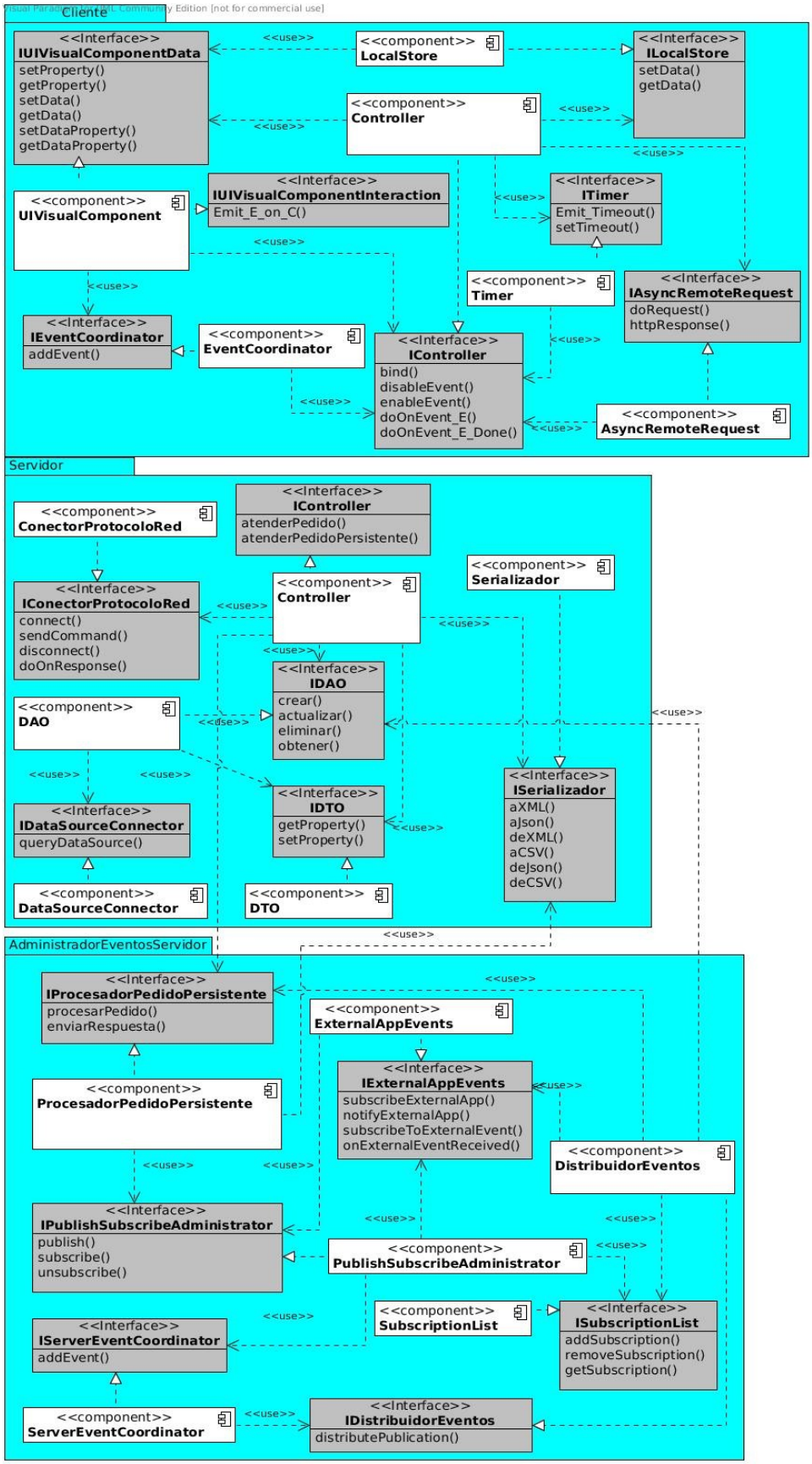

Figura 3-2: Vista estática de los componentes del modelo de arquitectura.

El diagrama muestra los estereotipos que fueron definidos para ser utilizados en los modelos de arquitectura de este trabajo.

La arquitectura independiente de plataforma se abstrae de:

- tecnología específica de implementación RIA;
- cómo se diseña un coordinador de eventos a partir de otras componentes más simples, y esto tanto del lado del cliente como del servidor;
- componentes conectoras a bases de datos y componentes conectoras a web services mediante una sola componente conectora a fuentes de datos del lado del servidor;
- del uso de componentes especiales cuyo fin es el de actualizar componentes visuales de la UI (en su contenido y propiedades);
- del uso de componentes especiales que solo actualizan datos locales para el almacenamiento volátil en el cliente; y
- componentes para conexiones persistentes y para pedidos asincrónicos al servidor mediante una sola componente del lado del cliente.

Además nuestra arquitectura de componentes permite simplificar el diseño al:

- evitar reflejar interacción con aplicaciones externas mediante varias componentes; en lugar de eso se usa una sola componente;
- evitar el uso de componentes específicas para tratar eventos específicos (p.ej. en componentes de UI);
- evitar tener un manejador de eventos para cada componente visual, en lugar de eso tenemos uno solo para todas las componentes.

La arquitectura independiente de plataforma se apoya en los siguientes patrones/esquemas bien conocidos:

- **Esquema de interacción publish-subscribe**: (ver [40] para más detalles) hay *suscriptores* con la habilidad de expresar su interés en un evento o patrón de eventos, con el fin de ser notificados, de cualquier evento generado por un *productor* (publisher) que concuerde con su interés registrado. Productores publican información en un *gestor de eventos* y consumidores suscriben a la información que quieren recibir de este bus de eventos. El modelo básico de publish-subscribe confía en un *servicio de notificación de eventos* que provee almacenamiento y gestión para suscripciones y una entrega eficiente de eventos. Nosotros implementamos el servicio de notificación de eventos por medio de las siguientes componentes: DistribuidorEventos (usada para notificar un evento a todos los clientes suscriptos al mismo), PublishSubscribeAdministrator (que controla las publicaciones de los eventos del servidor, como asi tambien la subscripcion y desubscripcion a los mismos), y SubscriptionList (que mantiene las asociaciones entre eventos y sus suscriptores). El desacoplamiento que el servicio de eventos provee entre suscriptores y publishers puede descomponerse en 3 dimensiones: *desacoplamiento de espacio* (las partes que interactúan – publishers y suscriptores -no necesitan conocerse entre sí), *desacoplamiento de tiempo* (las partes que interactúan no necesitan estar activamente participando en la

interacción en el mismo momento; en particular el Publisher puede publicar algunos eventos cuando el suscriptor está desconectado, y recíprocamente el suscriptor puede ser notificado de la ocurrencia acerca de un evento mientras el Publisher original está desconectado), *desacoplamiento de sincronización* (los publishers no están bloqueados mientras producen eventos y los suscriptores pueden ser notificados asincrónicamente – por medio de un callback- de la ocurrencia de un evento mientras hacen alguna actividad concurrente). Nosotros también contemplamos estos tipos de desacoplamientos en el modelo de componentes .

- **DAO pattern (Data Access Object)**: (ver [41] para más detalles) Un objeto de acceso a datos (DAO) implementa el mecanismo de acceso requerido para trabajar con una fuente de datos. Independientemente de tipo de fuente de datos se considere, el DAO siempre provee una API uniforme a sus clientes. Una componente de negocios que necesita acceder a datos usa la interfaz expuesta por el DAO. El DAO oculta los detalles de implementación de la fuente de datos. La interfaz expuesta por el DAO no cambia cuando la implementación de la fuente de datos subyacente cambia. Luego se puede cambiar la implementación del DAO sin cambiar la implementación del cliente del DAO. Además la componente de negocios crea/usa objetos de transferencia que son usados/creados por los DAO. Nosotros solo contemplamos las componentes DAO, objeto de transferencia (a la que llamamos DTO) Data source (a la que llamamos DataSourceConnector); la parte de business object no se representa explícitamente y está escondida dentro de la componente controller.
- **Model-View-Controller pattern** (MVC): MVC (ver [22]) separa el modelo de dominio (model), el modelo de presentación (view) y los comportamientos (controller). El modelo básicamente incluye el estado de la aplicación y las operaciones que pueden cambiar el estado. El modelo mantiene dependencias con el modelo de presentación y la componente controller, a los cuales notifica de los cambios de estado. El modelo de presentación presenta información al usuario a través de una UI. El modelo de presentación puede consultar el modelo para obtener información que necesita presentar. El controlador responde a acciones de usuario vía la UI y es el responsable de pasar transacciones al modelo para su ejecución. En [21] se proponen 2 patrones MVC para RIA. En el primero de ellos se tienen componentes model, view y controller tanto en el cliente como en el servidor web; la idea es que el model está distribuido tanto en el cliente como en el servidor y que cuando un evento en la view debe procesarse del lado del servidor, debe pasar por view, controller (y tal vez model) del cliente, para luego pasar por controller y model del servidor (y tal vez view); los autores dicen que se usa para programar en Ajax o con el uso de plug-ins del lado del cliente. En el segundo patrón (llamado RIA MVC) el cliente tiene componentes model, view y controller y el servidor solo tiene componente model; la comunicación se hace entre models del cliente y del servidor y puede ser asincrónica. Nosotros proponemos un enfoque en el cuál tenemos del lado del cliente componentes model (para datos

volátiles), view y controller, y del lado del servidor tenemos solo componentes model (para datos persistentes) y controller; la razón de tener un controller en el servidor es poder soportar acceso a otras facilidades como acceso a protocolos de aplicaciones de redes, poder notificar de eventos en el cliente a suscriptores (i. e. otros clientes de la aplicación web o aplicaciones externas).

- **Deserialization-serialization**: (ver [42] para más detalles) serialización es el proceso de convertir objetos complejos en un stream de bytes que puede ser fácilmente transmitible por la red o puede ser almacenado persistentemente. Deserialización es su proceso reverso, esto es, desempacar streams de bytes a su forma original. En nuestra arquitectura el serializamos los datos de respuesta del servidor web antes de enviarlos al cliente y deserializamos los datos desde un formato apropiado para transferencia HTTP a un formato apropiado para el servidor web. Para serializar y deserializar usamos una componente llamada serializador.

A continuación, describiremos cada uno de los estereotipos.

3.1 Descripción y propósito de cada estereotipo

3.1.1 Componentes del paquete Cliente

Nombre	**UIVisualComponent**
Descripción	Se encarga de la representación visual de los datos de la aplicación. Toda la interacción del usuario con la aplicación ocurrirá a través de ellos. Provee dos interfaces: IUIVisualComponentData y IUIVisualComponentInteraction. La primera permite manipular las propiedades visuales e indicar que datos son los que se muestran. La segunda es una interfaz de control, a través de la cual el usuario puede interactuar con el UIVisualComponent. El objetivo principal de los UIVisualComponent es proveer una interfaz de usuario de mayor interactividad que la permitida por el HTML estándard. La comunicación entre los UIVisualComponents con el resto de los componentes de la aplicación ocurre a través de eventos, lo cual eleva el grado de desacoplamiento y facilita la reutilización. En general, podemos decir que existen dos tipos de UIVisualComponent. Por un lado tenemos aquellos que ofician de contenedores, los cuales se ocupan de la estructura visual de la interfaz de usuario, principalmente el posicionamiento dentro de la pantalla. Por otro lado tenemos los que se encargan de capturar las acciones del usuario. Este tipo de UIVisualComponent presenta, generalemente, controles como campos de texto, checkboxes, botones, etc. o bien permiten al usuario algún tipo de interacción, como arrastrar y soltar. Entre los UIVisualComponent más comunes podemos encontrar listas de elementos con acciones, formularios, toolbars, menúes, listas con elementos arrastrables, galería de imágenes, ventanas de diálogo. Los UIVisualComponent permiten fijar los datos a mostrar.
Interfaces Provistas	IUIVisualComponentInteraction • Emit_<event>_on_<ComponentName>() - Es llamado como resultado de una interacción <event> (p.e: onClick) del usuario con el Component <ComponentName>. El evento es pasado al EventCoordinator o a Controller dependiendo de si el evento puede formar parte de un evento compuesto o no, respectivamente. IUIVisualComponentData • setProperty(propertyName, propertyValue) – fija una propiedad visual del UIVisualComponent (ancho, alto, color, ocultar/mostrar una sección, etc.) • getProperty() – Devuelve el valor de una propiedad visual del UIVisualComponent (ancho, alto, color, visibilidad de una sección, etc.) • getData() – devuelve los datos asociados al

	UIVisualCompoennt incluyendo cualquier cambio realizado a través de la interfaz de usuario respecto de los datos originales • setData(data) – setea los datos que el UIVisualComponent debe mostrar
Dependencias	IEventCoordinator IController ILocalStore
Comportamiento	• **EventCoordinator:** Cuando un usuario de la aplicación interactúa con el UIVisualComponent que implementa la interfaz IUIVisualComponentInteraction, este captura dicho evento y, si es un posible candidato a formar parte de un evento compuesto, se lo delega a EventCoordinator llamando a addEvent(). • **Controller:** Cuando un usuario de la aplicación interactúa con el Component que implementa la interfaz IUIVisualComponentInteraction, este captura dicho evento y, si NO es un posible candidato a formar parte de un evento compuesto, se lo delega a Controller llamando a doOnEvent_E. • **LocalStore:** A través del sistema de enlace de datos, se pueden asociar las propiedades de un LocalStore con las propiedades de un UIVisualComponent, de manera que, cuando ocurre algún cambio en el estado del LocalStore, automáticamente se refresca el UIVisualComponent asociado para reflejar dicho cambio llamando a setData.
# de instancias	Una o más

Nombre	**Controller**
Descripción	Este componente es el encargado de manejar los eventos que resultan de la interacción del usuario con los UIVisualComponent interactivos y los que son disparados desde la misma aplicación, como el evento timeout de Timer.
Interfaces Provistas	IController • doOnEvent_E() – implementa la respuesta al evento E. • disableEvent(eventType) - permite desactivar un eventos • enableEvent(eventType) - permite activar un evento • doOnEvent_E_Done(response) – función de callback que será llamada tras haber concluido el pedido al servidor solicitado por el controller
Dependencias	IUIVisualComponentData IEventCoordinator IAyncRemoteRequest ILocalStore

	ITimer
Comportamiento	• **UIVisualComponent:** como respuesta a los eventos generados a causa de la interacción del usuario con un IUVisualComponent, se llama a doOnEvent_E, siempre y cuando dichos eventos no sean candidatos a formar parte de un evento compuesto. • **EventCoordinator:** cuando EventCoordinator detecta un evento compuesto, es Controller el encargado de responder a tal evento llamando a doOnEvent_E. • **AyncRemoteRequest:** para obtener los datos remotos para llenar el LocalStore o solicitar algun procesamiento por parte del servidor, Controller llamará a doRequest y este llamará a doOnEvent_E_Done de Controller como función de callback luego de que el pedido al servidor ha concluido. • **LocalStore:** Controller utilizará LocalStores para almacenar los datos recibidos del servidor de manera local cuando sea necesario, llamando a setData(). • **Timer:** cuando Controller necesite hacer uso de un temporizador, llamará a setTimeout de Timer. Si ocurriera el evento timeout, Timer llamará a doOnEvent_E correspondiente.
# de instancias	Una o más

Nombre	**LocalStore**
Descripción	Este componente unifica el formato de los datos de la aplicación independientemente de la fuente de datos. Ya sea que el servidor envíe sus datos en formato XML, CSV, JSON o cualquier otro, siempre que sea neceario, serán almacenados en LocalStores, y serán tratados como tales a lo largo de toda la aplicación del lado del cliente.
Interfaces Provistas	ILocalStore • setData(data) – permite agregar datos al conjunto de datos. • getData() – permite obtener datos del conjunto de datos.
Dependencias	IController IUIVisualComponentData
Comportamiento	• **Controller:** Controller utilizará LocalStores para almacenar del lado del cliente, datos de la aplicación recibidos del servidor, cuando sea necesario. • **UIVisualComponent** Los LocalStores se comunican con los UIVisualComponents a través del sistema de Data Binding. Al hacer data binding, los UIVisualComponents son notificados ante cualquier cambio de estado en los LocalStores a los cuales están asociados, y se actualizan para

Nombre	**LocalStore**
	reflejar dicho cambio llamando a setData.
# de instancias	Una o más

Nombre	**AsyncRemoteRequest**
Descripción	Permite enviar pedidos asincrónicos al servidor y asociar una funcion de callback para manejar la respuesta.
Interfaces Provistas	IAsyncRemoteRequest • doRequest(request, callback) – envía el pedido al servidor y llama a callback con el resultado del pedido como parámetro • respuestaHttp() - es llamado como funcion de callback por el browser una vez que concluye la transferencia de la respuesta desde el servidor.
Dependencias	• IController
Comportamiento	• **Controller:** cuando Controller necesita realizar un pedido al servidor, llama a doRequest de AsyncRemoteRequest, pasándole como parámetros los datos para el pedido y una función de callback para procesar la respuesta.
# de instancias	Una o más

Nombre	**EventCoordinator**
Descripción	Se encarga de detectar eventos compuestos dentro de la aplicación cliente, como por ejemplo, arrastrar y soltar.
Interfaces Provistas	IEventCoordinator • addEvent(event) – mediante este método se le indica a EventCoordinator que analice un evento en relación con eventos ocurridos recientemente en busca de eventos compuestos
Dependencias	• IController
Comportamiento	• **Controller:** cuando EventCoordinator detecta un evento compuesto, dispara tal evento y esto produce la ejecución del manejador asociado doOnEvent_E de Controller
# de instancias	Una o más

Nombre	**Timer**
Descripción	Permite establecer tareas temporizadas, que se ejecutarán después de cierto intervalo de tiempo.
Interfaces Provistas	ITimer • setTiemout(time) – Permite iniciar un temporizador • Emit_Timout() - Dispara un evento avisando que el tiempo seteado ha transcurrido
Dependencias	• IController
Comportamiento	• **Controller:** cuando un Controller necesita de un temporizador como parte del procesamiento de un evento, llamará a setTimeout() del componente Timer. Cuando el evento timout ocurre, se llama a doAction correspondiente de Controller
# de instancias	Una o más

3.1.2 Componentes del paquete Servidor

Nombre	**ServerController**
Descripción	Representa la interfaz del servidor y expone todas las acciones disponibles. Cada pedido del cliente ejecuta una o mas acciones que generan una respuesta la cual es devuelta al cliente.
Interfaces Provistas	IController • atenderPedido(request) – es llamado como consecuencia de un pedido del cliente. Se encarga de procesar el pedido y generar una respuesta para el cliente. • atenderPedidoPersistente(request) – cuando un cliente solicita una conexión persistente para ser notificado de eventos disparados por el servidor, se llama a este método.
Dependencias	IAdministradorConexionesPersistentes IDAO Iserializador IConectorProtocoloRed
Comportamiento	• **ProcesadorPedidoPersistente:** Cuando Controller necesita establecer una conexión persistente con el cliente para poder notificarle sobre eventos ocurridos en el servidor, llama a procesarPedido de ProcesadorPedidoPersistente. • **DAO:** En el caso que Controller necesite obtener o persistir datos, llamará a alguna de las operaciones de DAO. • **Serializador:** Con el objeto de dar un formato a los datos para transferirlos mediante HTTP, Controller puede llamar a la funciones de serialización y deserialización de Serializador. Cuando los datos son recibidos desde cliente, Controller deberá deserializarlos a un formato que pueda ser entendido

	por el lenguaje nativo del servidor. Del mismo modo, antes de enviar la respuesta al cliente, los datos deben ser serializados para enviarlos al cliente a través de HTTP. • **ConectorProtocoloRed:** Si Controller necesita utilizar algún servicio de red como SMTP, IMAP, FTP o cualquier otro, puede acceder a tales conexiones mediante las operaciones de ConectorProtocoloRed. • **DTO:** el Controller se comunica con los DAO mediante DTOs. Para persistir datos, Controller llenara un DTO llamando a setData y se lo pasará a alguna operación de escritura del DAO. Si el Controller solicita datos al DAO, este los devolverá en un DTO o bien en una colección de DTO.
# de instancias	Una o más

Nombre	**ConectorProtocoloRed**
Descripción	Permite a la aplicación del lado del servidor acceder a servicios de red como SMTP, IMAP, FTP entre otros.
Interfaces Provistas	IConectorProtocoloRed • connect(protocolo) – permite establecer una conexión de red mediante el protocolo indicado • sendCommand(command) – permite enviar un comando del protocolo para el cual se estableció la conexión. • disconnect() - cierra la conexión de red • doOnResponse() - función de callback llamada luego de que la respuesta a un comando del protocolo para el cual se estableció la conexión ha sido recibida completamente.
Dependencias	IController
Comportamiento	• **Controller:** Si Controller necesita utilizar algún servicio de red como SMTP, IMAP, FTP o cualquier otro, puede acceder a tales conexiones mediante las operaciones de ConectorProtocoloRed.
# de instancias	Una o más

Nombre	**Serializador**
Descripción	Permite a la aplicación del lado del servidor deserializar los datos enviados por el cliente y serializar los datos de respuesta al pedido antes de enviarlos al cliente mediante HTTP
Interfaces Provistas	ISerializador • a<formatoSerializado>(data) – serializa los datos pasándolos del formato del lenguaje nativo del servidor a cierto formato (XML, CSV, etc) más apropiado para la transferencia mediante HTTP

	• de<formatoSerializado>(data) – deserializa los datos desde un formato más apropiado para transferencia mediante HTTP a un formato más apropiado para el lenguaje nativo del servidor (un array, una colección, etc)
Dependencias	IController
Comportamiento	• **Controller:** Con el objeto de dar un formato a los datos para transferirlos mediante HTTP, Controller puede llamar a la funciones de serializacion y deserializacion de Serializador. Cuando los datos son recibidos desde cliente, Controller deberá deserializarlos a un formato que pueda ser entendido por el lenguaje nativo del servidor. Del mismo modo, antes de enviar la respuesta al cliente, los datos deben ser serializados para enviarlos al cliente a través de HTTP.
# de instancias	Una o más

Nombre	**DAO**
Descripción	Provee una abstracción de acceso a los datos de la aplicación. En su interfaz encontramos operaciones para manipular dichos datos.
Interfaces Provistas	IDAO • crear(data) – permite crear una nueva entrada en la fuente de datos • actualizar() • eliminar() • obtener()
Dependencias	IController IDTO
Comportamiento	• **Controller:** En el caso que Controller necesite obtener o persistir datos de la aplicación como resultado del procesamiento de un pedido del cliente, llamará a alguna de las operaciones de DAO. • **DTO:** el Controller se comunica con los DAO a mediante DTOs. Para persistir datos, Controller llenara un DTO llamando a setData y se lo pasara a alguna operación de escritura del DAO. Si el Controller solicita datos al DAO, este los devolverá en un DTO o bien en una colección de DTO.
# de instancias	Una o más

Nombre	DTO
Descripción	Este componente abstrae el formato de los datos que se pasan entre la capa de negocios (Controller) y la capa de datos (DAO). La información entre estas dos capas se realiza mediante DTOs. Son componentes simples, que constan de propiedades, cada una con su valor, y métodos para leer y escribir tales propiedades.
Interfaces Provistas	IDTO • setProperty(propertyName, propertyValue) – setea el valor de una propiedad del DTO • getProperty(propertyName) - devuelve el valor de una propiedad del DTO
Dependencias	IController IDAO
Comportamiento	• **Controller y DAO:** el Controller se comunica con los DAO a mediante DTOs. Para persistir datos, Controller llenara un DTO llamando a setData y se lo pasara a alguna operación de escritura del DAO. Si el Controller solicita datos al DAO, este los devolverá en un DTO o bien en una colección de DTO.
# de instancias	Una o más

Nombre	DataSourceConnector
Descripción	Permite a la aplicación servidor conectarse a la fuente de datos como una base de datos relacional, un web service, etc.
Interfaces Provistas	IDataSourceConnector • queryDataSource() – Permite ejecutar un consulta de escritura o lectura en la fuente de datos
Dependencias	IDAO
Comportamiento	• **DAO:** Para acceder a la fuente de datos, DAO ejecutará consultas llamando a queryDataSource de DataSourceConnector
# de instancias	Una o más

3.1.3 Componentes del paquete AdministradorEventosServidor

Nombre	ProcesadorPedidoPersistente
Descripción	Se encarga de mantener abiertas las conexiones persistentes entre el cliente y el servidor y de notificar al cliente cada vez que ocurre en evento en el servidor y al cual el cliente esta subscripto. A través de este componente, los clientes pueden enviar pedidos de subscripción a un evento o publicar eventos del servidor.
Interfaces Provistas	IProcesadorPedidoPersistente • procesadorPedido(request) – realiza las acciones necesarias

	para procesar un pedido persistente, como mantener la conexión abierta, subscribir a un cliente a un evento del servidor, etc. • enviarRespuesta(response) – envia una respuesta al cliente a través de una conexión persistente, sin cerrarla al finalizar.
Dependencias	IController IPublishSubscribeAdministrator IDistribuidorEventos
Comportamiento	• **Controller:** Cuando Controller recibe un pedido del cliente que requiere de una conexión persistente, delega el procesamiento del mismo a ProcesadorPedidoPersistente, llamando a procesarPedido. • **PublishSubscribeAdministrator:** El cliente realiza pedidos persistentes para subscribirse a eventos del servidor. Esto ocurre cuando ProcesadorPedidoPersistente llama a subscribe de PublishSubscribeAdministrator • **DistribuidorEventos:** cuando ocurre un evento en el servidor al cual están subscriptos uno o mas clientes, DistribuidorEventos llama a enviarRespuesta de ProcesadorPedidoPersistente para notificar a cada uno de los subscriptores.
# de instancias	Una o más

Nombre	**PublishSubscribeAdministrator**
Descripción	Controla las publicaciones de los eventos del servidor, como así tambien la subscripción y desubscripción a los mismos.
Interfaces Provistas	IPublishSubscribeAdministrator • publish(event) – dispara un evento del lado del servidor • subscribe(client, event) – subscribe a un cliente en particular a un evento del servidor determinado, de modo que cada vez que el evento ocurra, el cliente será notificado • unsubscribe(client, event) – revoca la subscripción de un cliente a un evento determinado, de manera que ya no recibirá notificaciones sobre la ocurrencia de este evento
Dependencias	IProcesadorPedidoPersistente IExternalAppEvents ISubscriptionList IServerEventCoordinator
Comportamiento	• **ProcesadorPedidoPersistente:** Como parte del procesamiento de un pedido persistente, ProcesadorPedidoPersistente puede hacer tres cosas: llamar a subscribe de PublishSubscribeAdministrator para subscribir a un cliente a un evento determinado, o bien llamar a unsubscribe de PublishSubscribeAdministrator

	para revocar la subscripción de un cliente a un evento determinado, o bien llamar a publish de PublishSubscribeAdministrator, para indicar la ocurrencia de un evento a nivel del servidor. • **ExternalAppEvents:** Cuando nuestra aplicación servidor esta subscripta a un evento de una aplicación externa y tal evento ocurra, ExternalAppEvents llamará a publish de PublishSubscribeAdministrator. En el caso de que el servidor necesite subscribirse a un evento de un aplicación externa, o una aplicación externa solicite subscribirse a un evento del servidor, PublishSubscribeAdministrator llamará a subscribeExternalApp o subscribeToExternalEvent de ExternalAppEvents • **SubscriptionList:** Cada vez que se llama a subscribe o unsubscribe de PublishSubscribeAdministrator para asociar o remover la asociación de un cliente con un evento del servidor, este deberá llamar a addSubscription o removeSubscription de SubscriptionList, respectivamente. • **ServerEventCoordinator:** Cada vez que se llama a publish de PublishSubscribeAdministrator, este llama a addEvento de ServerEventCoordinator para que este analice si el evento forma parte o no de un evento compuesto del servidor.
# de instancias	Una o más

Nombre	**ServerEventCoordinator**
Descripción	Cada vez que ocurre un evento del servidor, este componente analiza la posibilidad de ocurrencia de un evento compuesto.
Interfaces Provistas	IServerEventCoodinator • addEvent(event) – dado un evento, basándose en ocurrencias recientes de otros eventos, analiza si forma parte de un evento compuesto.
Dependencias	IPublishSubscribeAdministrator IDistribuidorEventos
Comportamiento	• **PublishSubscribeAdministrator:** cada vez que se llama a publish de PublishSubscribeAdministrator, este llama a su vez, a addEvent de ServerEventCoordinator para que este analice si el nuevo evento ocurrido forma parte o no de un evento compuesto. • **DistribuidorEventos:** luego de que ServerEventCoodinator analiza si ha ocurrido un evento compuesto o no, le indica a DistribuidorEvento cual es el evento que ha ocurrido en realidad, llamando a distributePublication
# de	Una o más

instancias	

Nombre	**DistribuidorEventos**
Descripción	Es el encargado de, cada vez que ocurre un evento en el servidor, notificar a todos los clientes que estan subscriptos al mismo
Interfaces Provistas	IDistribuidorEventos • distributePublication(event) – se encarga de buscar todos los clientes subscriptos a un determinado evento y notificarles la ocurrencia del mismo
Dependencias	IServerEventCoordinator IProcesadorPedidoPersistente IExternalAppEvents
Comportamiento	• **ServerEventCoodinator:** luego de que ServerEventCoodinator analiza si ha ocurrido un evento compuesto o no, le indica a DistribuidorEvento cual es el evento que ha ocurrido en realidad, llamando a distributePublication • **ProcesadorPedidoPersistente:** cuando ocurre un evento en el servidor al cual están subscriptos uno o mas clientes, DistribuidorEventos llama a enviarRespuesta de ProcesadorPedidoPersistente para notificar a cada uno de los subscriptores. • **ExternalAppEvents:** cuando DistribuidorEventos encuentra que un subscriptor de cierto evento ocurrido es una aplicación externa, la notifica llamando a notifyExternalApp de ExternalAppEvents.
# de instancias	Una o más

Nombre	**ExternalAppEvents**
Descripción	Permite a aplicaciones externas subscribirse y desubscribirse a eventos de la aplicación servidor. También permite a la aplicación servidor subscribirse y desubscribirse a eventos de aplicaciones externas.
Interfaces Provistas	IExternalAppEvents • subscribeExternalApp(appId, event) – permite que una aplicación externa se subscriba a un evento del servidor • notifyExternalApp(appId, event) – notifica a una aplicación externa, previamente subscripta a cierto evento del servidor, la ocurrencia de tal evento • subscribeToExternalEvent(eventSource, event) – permite a la aplicación servidor subscribirse a un evento disparado por una aplicación externa • onExternalEventReceived(event) – esta operación es llamada cuando ha ocurrido un evento de una aplicación externa, al que la aplicación servidor se había subscripto

	previamente
Dependencias	IPublishSubscribeAdministrator IDistribuidorEventos
Comportamiento	• **PublishSubscribeAdministrator:** Cuando nuestra aplicación servidor esta subscripta a un evento de una aplicación externa y tal evento ocurra, ExternalAppEvents llamará a publish de PublishSubscribeAdministrator. En el caso de que el servidor necesite subscribirse a un evento de un aplicación externa, o una aplicación externa solicite subscribirse a un evento del servidor, PublishSubscribeAdministrator llamará a subscribeExternalApp o subscribeToExternalEvent de ExternalAppEvents • **DistribuidorEventos:** cuando DistribuidorEventos encuentra que un subscriptor de cierto evento ocurrido es una aplicación externa, la notifica llamando a notifyExternalApp de ExternalAppEvents.
# de instancias	Una o más

Nombre	**SubscriptionList**
Descripción	Este componente se encarga de mantener la asociación entre subscriptores y eventos del servidor. Permite agregar y remover subscriptores.
Interfaces Provistas	IPublishSubscribeAdministrator • addSubscription(client, event) – asocia un cliente con un evento del servidor • removeSubscription(client, event) – remueve la asociación entre un cliente y un evento del servidor • getSubscription(event) – devuelve todos los clientes que estan subscriptos a un evento particular
Dependencias	IPublishSubscribeAdministrator IDistribuidorEventos
Comportamiento	• **PublishSubscribeAdministrator:** cada vez que se llama a subscribe o unsubscribe de PublishSubscribeAdministrator, este llamará a addSubscription o removeSubscription de SubscriptionList para reflejar el cambio en la asociación entre un cliente y un evento. • **DistribuidorEventos:** Para saber a que clientes debe notificar luego de la ocurrencia de un evento del servidor, DistribuidorEventos llama a getSubscription de SubscriptionList
# de instancias	Una o más

3.2 Trabajo Relacionado

Hay escasez de notaciones de diseño de componentes para RIAs. Solo encontramos el enfoque OOH4RIA (ver [3]) que define modelo de features con features de cliente y servidor. El modelo de features tiene features asociadas a tecnologías específicas. Modelo de features se transforma a esqueleto de arquitectura; luego se completa esqueleto de arquitectura con relaciones y atributos y se obtiene modelo de arquitectura que es de componentes y estático. Pero este modelo de arquitectura es demasiado genérico, solo dice tipos de componentes a usar, y no define la arquitectura específica de la aplicación que se está desarrollando. Además no considera modelado de arquitectura dinámico. No es una arquitectura independiente de tecnología porque incluye elementos de tecnología.

Uno de los trabajos más detallados sobre construcción de arquitecturas para RIA fue producido por Microsoft (ver [23]). En esta propuesta se define la arquitectura a groso modo con las capas a considerar; a cada capa la divide en partes; además se dan patrones que se pueden considerar. Este trabajo no considera notaciones de arquitectura para modelar la arquitectura; no explica en qué interfaces se basa cada capa, ni cómo se conectan las capas entre sí, ni subdivide cada capa en suficientes detalles; tampoco se explica comportamiento dinámico de la arquitectura. Este trabajo de Microsoft se puede usar para desarrollar aplicaciones RIA desde pequeñas a grandes; sin embargo para aplicaciones RIA medianas o pequeñas es más fácil de aplicar y de entender nuestro enfoque.

Este trabajo de Microsoft considera montones de patrones que los desarrolladores no suelen conocer bien y es fácil comprenderlos mal, usarlos mal y cometer errores, y además se los puede componer mal; nosotros en cambio consideramos un set reducido de componentes posibles; nosotros en lugar de dejar librado al desarrollador la elección de patrones hacemos que el desarrollador tenga que escoger componentes de ciertos estereotipos e indicamos cómo interactúan entre sí estas componentes.

Luego encontramos un paper sobre cómo debería ser patrón MVC para RIAs (ver [21]); a este paper ya lo tratamos en la introducción de este capítulo.

Finalmente, hay varios frameworks para RIA basados en tecnologías específicas; pero a estos no se los considera aquí, porque hablamos de arquitectura independiente de tecnología.

Capítulo 4 - Representación de una aplicación RIA mediante modelos de Análisis

El uso de estereotipos de análisis para el caso de las aplicaciones RIA sigue ciertas reglas que aclaramos a continuación:

- <<boundary>>: son aquellos objetos que permiten la interacción de los diferentes actores con el sistema. Para el caso de las aplicaciones RIA tenemos 4 tipos de objetos frontera:
    - Los objetos <<boundary>> de tipo UI que corresponden a elementos de la interfaz con el usuario.
    - Los objetos <<boundary>> de tipo Timer corresponden a elementos que permiten ejecutar acciones temporizadas o recibir eventos de timer.
    - Los objetos <<boundary>> de tipo ConectorProtocoloRed permiten al sistema interactuar con servicios de red como FTP, SMTP, POP3, etc.
    - Los objetos <<boundary>> de tipo ExternalApp son interfaces que permiten la interacción de aplicaciones externas con la aplicación web y viceversa.

  Este tipo de objetos provee operaciones de tipo evento, que permiten la interacción de los actores con la aplicación y operaciones utilizadas por los objetos <<control>> para que la aplicación interactúe con los actores.

- <<entity>>: objetos usados para referirse a las informaciones accedidas o actualizadas por la aplicación; una misma información de la aplicación puede ser almacenada en forma persistente en el servidor, en forma volátil en el cliente o de ambas maneras simultáneamente. Este tipo de objeto provee operaciones de tipo save*() para guardar datos tanto en almacenamiento volátil como persistente y get*() para obtener esos datos.

4.1 Abstracción de diagramas de comunicación de análisis mediante grafos

Para poder describir formalmente un patrón de comunicación RIA vamos a apoyarnos en la representación de los modelos de análisis mediante grafos. Un modelo de análisis

MA será representado mediante un grafo (M, R) donde M es el conjunto de mensajes de MA y R es una relación binaria, con R contenido en MxM:

- m1 R m2 equivale a decir que m2 aparece inmediatamente después de m1 en MA (el número de m2 es el número siguiente a m1 en el MA).
- m1 R* m2 equivale a decir que existe una secuencia de mensajes en MA que comienza en m1 y termina en m2

Para clasificar los diferentes tipos de mensajes utilizamos la expresión m:T(M) para decir que m es un mensaje de tipo T tomado del conjunto M. Ejemplos de estos tipos de mensajes son:

- tcm:timerMessage(M) – Son mensajes que provienen de temporizador a objeto de control.
- controlMessage(M) – Son mensajes de objeto de frontera a objetos de control.

En general, <tipo-objeto>Message son mensajes que van a <tipo-objeto>.

4.2 Patrones de Comunicación para el dominio de las RIA

Para el dominio de las aplicaciones RIA, los patrones de comunicación involucran a tres participantes: clientes, servidor y aplicaciones externas. El cliente corresponde a la interfaz gráfica de la aplicación con la cual interactúa el usuario. Puede haber diferentes tipos de clientes que desempeñan roles distintos. Las aplicaciones externas proveen o consumen datos del sistema en desarrollo. El servidor se encarga de persistir los datos de la aplicación y de comunicar diferentes clientes entre sí, diferentes aplicaciones externas entre sí, o de comunicar clientes con aplicaciones externas.

Cada caso de uso comprende un patrón de comunicación. Los distintos patrones imponen ciertas reglas que debe respetar la comunicación de objetos en el modelo de análisis de ese patrón. Estas reglas serán descritas mediante fórmulas lógicas que expresan propiedades que deben cumplir los grafos asociados a los modelos de análisis para estar de acuerdo con un patrón determinado.

Estos patrones de comunicación son importantes porque en el siguiente capítulo se define un mapeo del modelo de análisis de aplicaciones RIA a arquitecturas de referencia del dominio RIA; dicho mapeo define reglas para generar partes del modelo de arquitectura a partir de patrones de comunicación RIA presentes en el modelo de análisis de la aplicación RIA. Para simplificar el mapeo, asumiremos que cada caso de uso analizado por el algoritmo se corresponde con un único patrón de comunicación.

Capítulo 5 - Descripción de los Patrones de Comunicación para las RIA

Primero describiremos los patrones en lenguaje natural y luego, se dará una descripción precisa y rigurosa, utilizando notación relacional que involucra el concepto de clausura transitiva de una relación, y que usa cuantificadores y conectivos de lógica de primer orden.

5.1 El patrón CLIENTE-SERVIDOR

5.1.1 Descripción en lenguaje natural

- El usuario interactúa con un objeto <<boundary>> de tipo UI, el cual dispara un evento.
- Este evento es procesado por un objeto <<control>> de tipo EventProcessor.
- Como parte del procesamiento del evento, se llama a un operación de un objeto <<entity>> para obtener o almacenar datos persistentemente.
- Una vez que se obtienen los datos, o son persistidos, el objeto <<control>> actualiza la interfaz. (Opcional)

5.1.2 Descripción en lenguaje formal

M conjunto de Mensajes respeta el patrón cliente-servidor sí y sólo si:

- $\exists$ cm:controlMessage(M), e:uiEvent(M), em:entityMessage(M): (e R cm) $\wedge$ (cm R* em)
- $\forall$ pcm:protocolConnectorMessage(M) $\Rightarrow$ pcm:protocolConnectorMessageFromControl(M)
- $\forall$ tcm:timerMessage(M) $\Rightarrow$ tcm:timerMessageFromControl(M)
- $\exists$ e:timeoutEvent(M) $\Rightarrow$ $\exists$ cm:controlMessage(M): (e R cm)
- |ControlObject(M)| = 1

- $\nexists$ cm:controlMessage(M), eam:externalAppEventMessage(M): (cm R eam)
- $\nexists$ cm1:controlMessage(M), cm2:controlMessage(M): (cm1 R cm2)
- $\nexists$ eam:messageFromExernalApp(M)

5.1.3 Ejemplos

La Figura 5-1 ilustra un caso de uso que respeta el patrón Cliente-Servidor. En un sitio de e-commerce, un usuario agrega un producto a su carrito de compras. Esta acción es representada por los mensajes Emit_SelectItem_on_ProductList, Emit_AddSelectedItem_on_ShoppingCart y doOnItemAddedToShoppingCart. El

estado del carrito de compras es persistido, lo cual está representado por el mensaje addItem y se actualiza la interfaz del usuario para reflejar que el producto está en el carrito, mediante el mensaje appendItemToShoppingCart. Como el producto ya está en el carrito, lo removemos de la lista de productos seleccionables. En el caso de que haya un error, el mensaje setStatus muestra una leyenda describiendo el error.

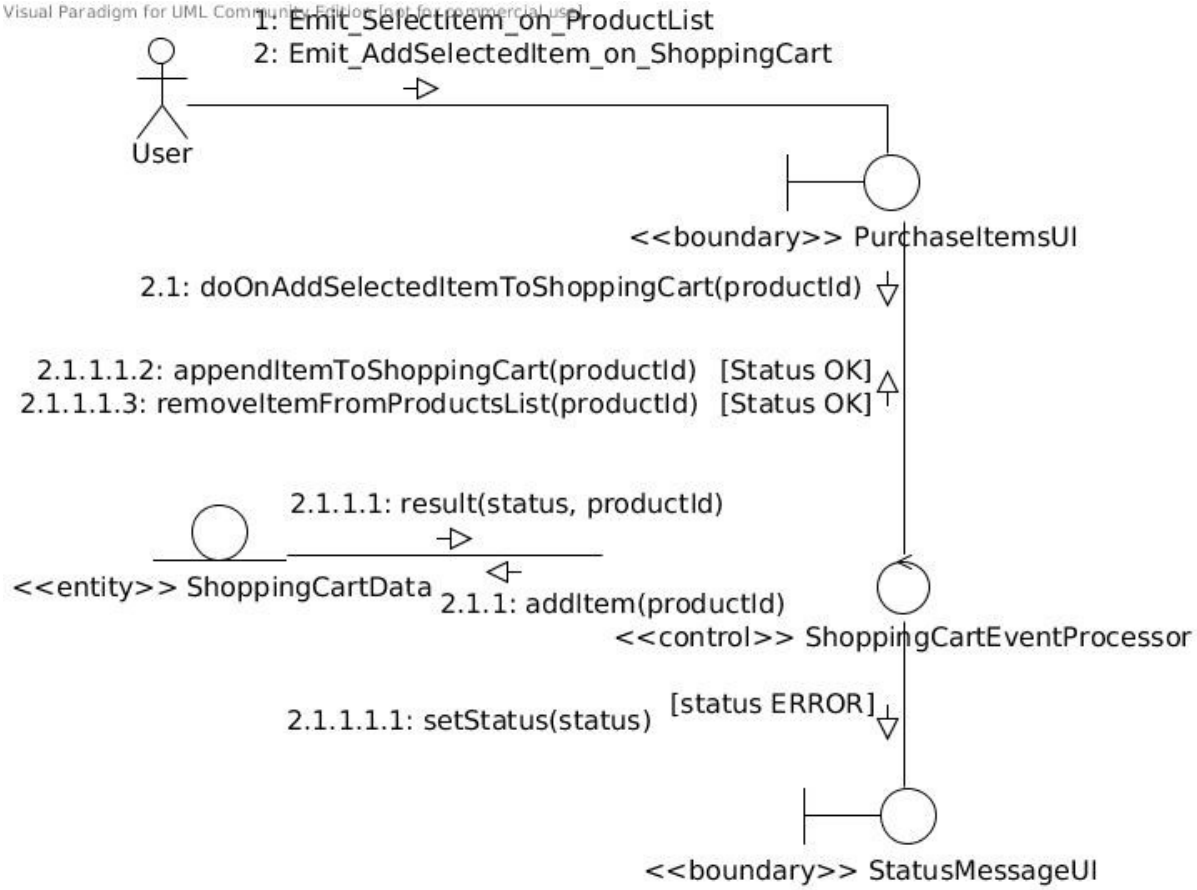

Figura 5-1: Ejemplo del patrón de comunicación cliente-servidor

5.2 El patrón CLIENTE-SERVIDOR-CLIENTES

5.2.1 Descripción en lenguaje natural

1. El usuario interactúa con un objeto <<boundary>> de tipo UI, el cual dispara un evento.
2. Este evento es procesado por un objeto <<control>> de tipo EventProcessor.
3. Se genera un evento del servidor E que depende del evento disparado por el objeto <<boundary>> en el paso 1.
4. Se obtiene de un objeto <<entity>> la lista de clientes subscritos a un evento. Cada cliente de la lista se procesa como sigue:
    - Si el cliente está en línea sucede al menos una de las siguientes:
        - Se guarda la información de la notificación en un objeto

<<entity>>

- Cada usuario que recibe la notificación procesa la información del evento y/o muestra un mensaje o alerta visual en la interfaz de usuario.

- Si el cliente no está conectado al momento de ocurrencia del evento, este puede ser descartado, es decir, nunca se notifica a tal cliente o bien queda almacenado en un objeto <<entity>> del servidor que funciona como historial de eventos no entregados.

5.2.2 Descripción en lenguaje formal

M conjunto de Mensajes respeta el patrón cliente-servidor-clientes sí y sólo si:

- ∃ e:uiEvent(M),
  cm:controlMessage(M),
  (e R cm) ∧
  (∃ E:serverEvent(M):
  (∃ em: getSubscriptors(E, M),
  emgq: getQueue():
  (∀ s:Results(em):
(cm R emgq) ∧ (emgq R* cleanQueueMessage(E, M)) ∧ (cleanQueueMessage(E, M) R em) ∧
(E = toDE(toSE(e, M), Results(emgq))) ∧
(online(s) ⇒
(∃ uim2(s, E): uiEventNotificationMessage(M) :
(cm R uim2(s,E)) ∧ target(uim2(s, E)) in Object(s)) ∨
(∃ em2(s, E):storeEntityMessage(M):
(cm R em2) ∧ target(em2) in Object(s))) ∨
(offline(s) ⇒ cm R storeInEventHistoryEntity(s, E)))))

- ∀ pcm:protocolConnectorMessage(M) ⇒ pcm:protocolConnectorMessageFromControl(M)
- ∀ tcm:timerMessage(M) ⇒ tcm:timerMessageFromControl(M)
- ∃ e:timeoutEvent(M) ⇒ ∃ cm:controlMessage(M): (e R cm)
- |ControlObject(M)| = 1
- ∄ cm:controlMessage(M), eam:externalAppEventMessage(M): (cm R eam)
- ∄ eam:messageFromExternalApp(M)

5.2.3 Ejemplos

La Figura 5-2 muestra un ejemplo de este patrón mediante un sistema de chat. Un usuario llamado Usuario1 escribe un mensaje en un campo de texto y aprieta Enter (mensajes 1 y 2). Este evento es capturado, se obtiene la lista de subscriptores al evento (mensajes 2.1 y 2.2) y se notifica de la ocurrencia del evento a todos los usuarios que previamente se subscribieron y está online al momento de ocurrencia del evento. Como resultado de la notificación, el mensaje enviado aparece en sus ventanas de chat (mensaje 2.2.1.B*). Para aquellos subscriptores que no se encuentran en línea, se persiste la ocurrencia del evento para notificarlo cuando el subscriptor este en línea (mensaje 2.2.1.A*).

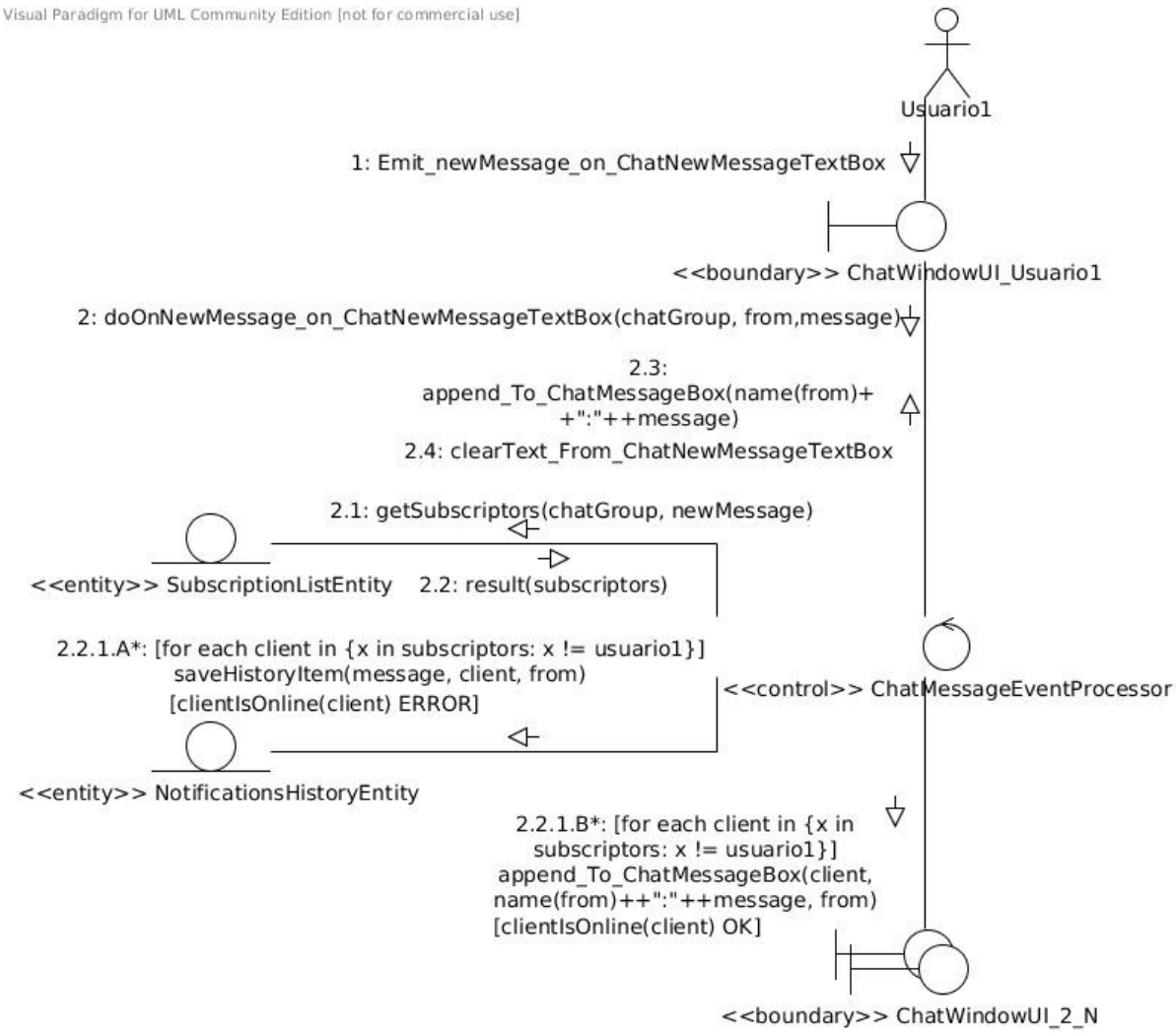

Figura 5-2: Ejemplo del patrón de comunicación cliente-servidor-clientes

5.3 APLICACIÓN EXTERNA-SERVIDOR-CLIENTES

5.3.1 Descripción en lenguaje natural

- Una aplicación externa interactúa con un objeto <<boundary>> de tipo ExternalApp, el cual dispara un evento.
- Este evento es procesado por un objeto <<control>> de tipo EventProcessor.
- Se genera un evento del servidor E que depende del evento disparado por el objeto <<boundary>> en el paso 1.
- Se obtiene de un objeto <<entity>> la lista de clientes subscriptos a un evento. Cada cliente de la lista se procesa como sigue:
    - Si el cliente esta online sucede al menos una de las siguientes:
        - Se guarda la información de la notificación en un objeto <<entity>>

- Cada usuario que recibe la notificación procesa la información del evento y/o muestra un mensaje o alerta visual en la interfaz de usuario.

- Si el cliente no está conectado al momento de ocurrencia del evento, este puede ser descartado, es decir, nunca se notifica a tal cliente o bien queda almacenado en un objeto <<entity>> del servidor.

5.3.2 Descripción en lenguaje formal

M conjunto de Mensajes respeta el patrón aplicación externa-servidor-clientes sí y sólo si:

- ∃ eaem:externalAppEventsMessage(M),
  cm: controlMessage(M):
  (eaem R cm) ∧
  (∃ E:serverEvent(M):
  (∃ em: getSubscriptors(E, M),
  emgq: getQueue():
  (∀ s:Results(em):
  (cm R emgq) ∧ (emgq R* cleanQueueMessage(E, M)) ∧
  (cleanQueueMessage(E, M) R em) ∧ (E = toDE(toSE(e, M), Results(emgq))) ∧
  (online(s) ⇒
  (∃ uim2(s, E): uiEventNotificationMessage(M) :
  (cm R uim2(s, E)) ∧ target(uim2(s, E)) in Object(s)) ∨
  (∃ em2(s, E):storeEntityMessage(M):
  (cm R em2) ∧ target(em2) in Object(s))) ∨
  (offline(s) ⇒ (cm R storeInEventHistoryEntity(s, E)))))))

- ∀ pcm:protocolConnectorMessage(M) ⇒ pcm:protocolConnectorMessageFromControl(M)
- ∀ tcm:timerMessage(M) ⇒ tcm:timerMessageFromControl(M)
- ∃ e:timeoutEvent(M) ⇒ ∃ cm:controlMessage(M): (e R cm)
- |ControlObject(M)| = 1
- |ExternalAppEventsObject(M)| = 1
-
- ∄ eam:externalAppEventMessage(M)
- ∄ cm1:controlMessage(M), cm2:controlMessage(M): (cm1 R cm2)

5.3.3 Ejemplos

La Figura 5-3 presenta un escenario que respeta el patrón aplicación externa-servidor-clientes. El ejemplo asume que los usuarios se han suscrito al evento enviado por la aplicación externa mediante la aplicación principal. La aplicación externa notifica a la aplicación principal sobre un cambio en el precio de alguna acción en particular de la bolsa de valores (mensajes 1, 1.1). Se obtiene la lista de subscriptores a este evento (mensajes 1.1.1 y 1.1.1.1). Los usuarios suscritos a este evento que están en línea al momento de la ocurrencia, reciben tal notificación, la cual actualiza la lista donde ven el precio de las acciones (mensaje 1.1.1.1.1A*). Para los usuarios que no están suscritos se persiste la información del evento (mensaje 1.1.1.1.1B*) para notificarlos cuando vuelvan a estar en línea.

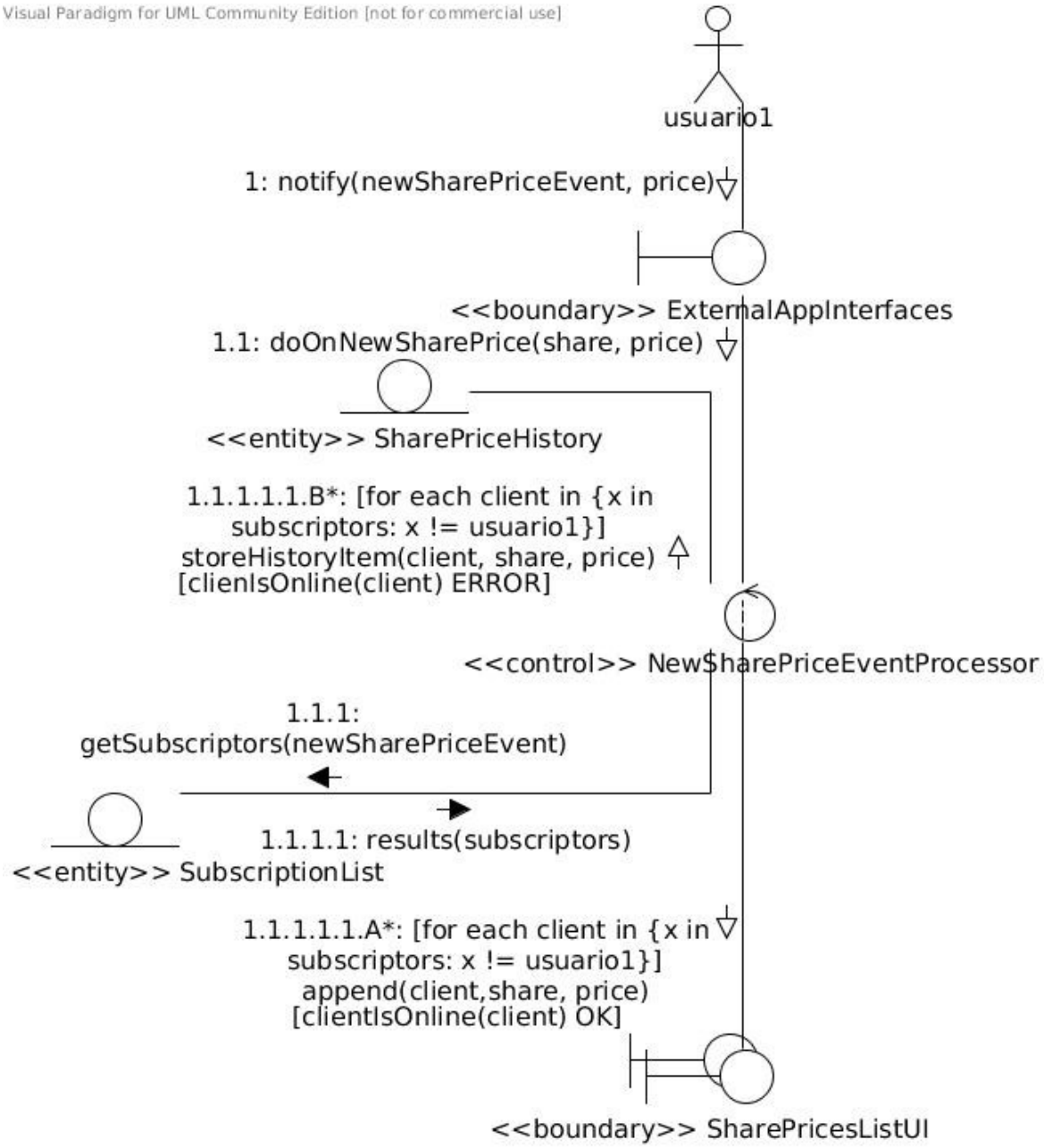

Figura 5-3: Ejemplo de patrón de comunicación aplicación externa-servidor-clientes

5.4 CLIENTE-SERVIDOR-APLICACIONES EXTERNAS

5.4.1 Descripción en lenguaje natural

- Un usuario interactúa con un objeto <<boundary>> de tipo UI, el cual dispara un evento.
- Este evento es procesado por un objeto <<control>> de tipo EventProcessor.
- Se genera un evento del servidor E que depende del evento disparado por el objeto <<boundary>> en el paso 1.

- Se obtiene de un objeto <<entity>> la lista de aplicaciones extermas subscriptas a un evento. Cada aplicación externa de la lista se procesa como sigue:
    - Si está online:
        - Se envía la notificación a la aplicación externa a través de un objeto de tipo <<boundary>> ExternalApp, la cual procesa el evento.
    - Si la aplicación externa no está conectada al momento de ocurrencia del evento, este puede ser descartado, es decir, nunca se notifica a tal cliente o bien queda almacenado en un objeto <<entity>> del servidor.

5.4.2 Descripción en lenguaje formal

M conjunto de Mensajes respeta el patrón cliente-servidor-aplicaciones externas sí y sólo si:

- ∃ e:uiEvent(M),
  cm:controlMessage(M):
  (e R cm) ∧
  (∃ E: serverEvent(M):
  (∃ em: getSubscriptors(E, M),
  emgq: getQueue(),
  (∀ s:Result(em):
  (cm R emgq) ∧ (emgq R* cleanQueueMessage(E, M)) ∧
  (cleanQueueMessage(E, M) R em) ∧ (E = toDE(toSE(e, M), Results(emgq))) ∧
  (online(s) ⇒ ∃ eaem(s, E): notifyExternalApp(M): (cm R eaem)) ∨
  (offline(s) ⇒ (cm R storeInEventHistoryEntity(s, E)))
)
)
)
- ∀ pcm:protocolConnectorMessage(M) ⇒ pcm:protocolConnectorMessageFromControl(M)
- ∀ tcm:timerMessage(M) ⇒ tcm:timerMessageFromControl(M)
- ∃ e:timeoutEvent(M) ⇒ ∃ cm:controlMessage(M): (e R cm)
- |ControlObject(M)| = 1
- |ExternalAppEventsObject(M)| = 1
- ∄ cm:controlMessage(M), eam:externalAppEventMessage(M): (cm R eam)
- ∄ eam:messageFromExternalApp(M)

5.4.3 Ejemplos

La Figura 5-14 muestra un caso de uso que respeta el patrón cliente-servidor-aplicaciones externas. Este ejemplo asume que las aplicaciones externas ya se han subscripto al evento a través de la aplicación principal. Un usuario cambia el precio de una acción de la bolsa de valores (mensajes 1 y 1.1). Este evento es capturado y se obtiene los subscriptores (mensajes 1.1.1 y 1.1.1.1). Los subscriptores que están en línea en el momento de la ocurrencia del evento son notificados inmediatamente y se actualiza su interfaz para reflejar el cambio (mensaje 1.1.1.1.1A*). Para los que no están en línea, persiste la información del evento para notificarlo cuando el suscriptor vuelva a estar en línea (mensaje 1.1.1.1.1B*).

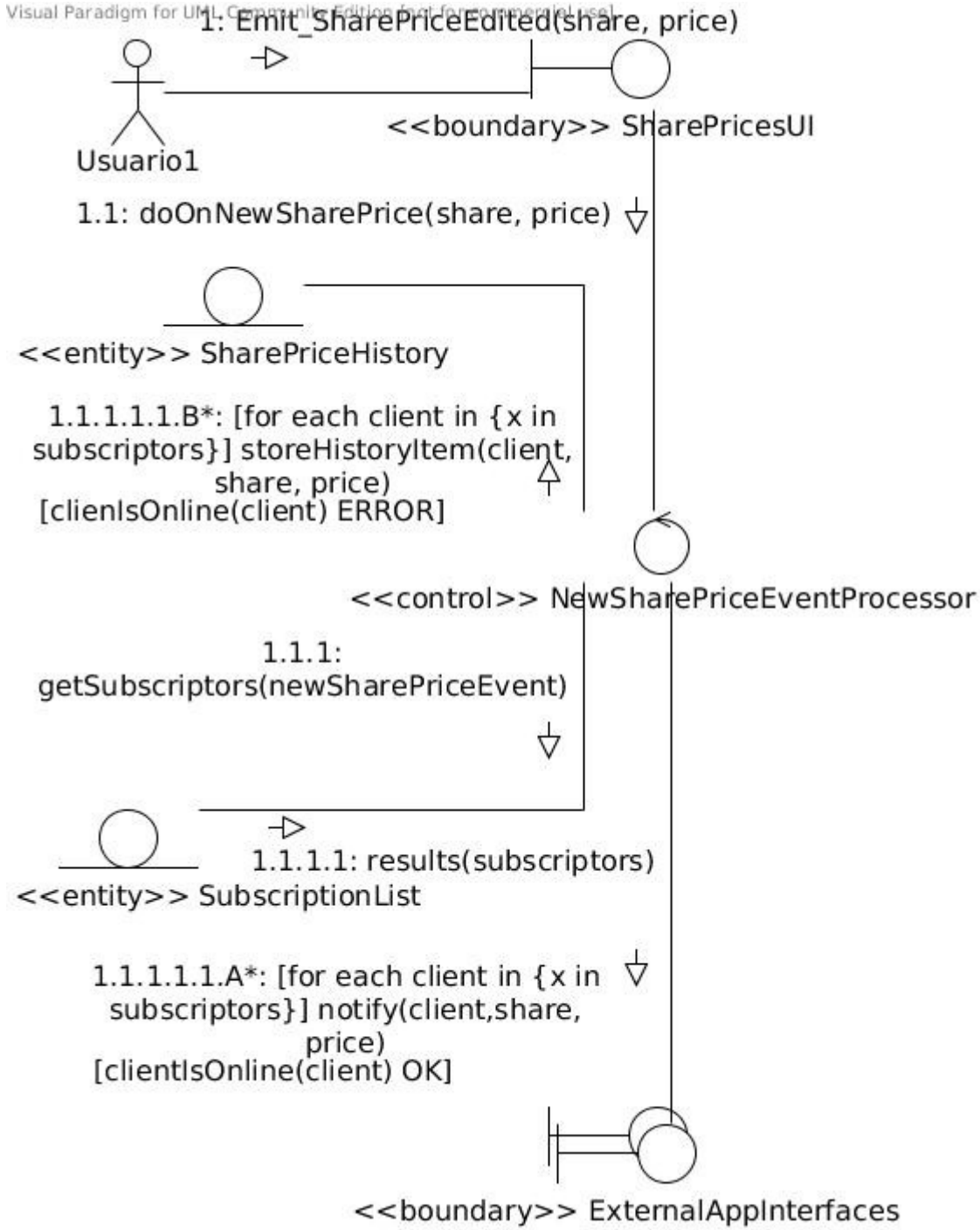

Figura 5-4: Ejemplo del patrón de comunicación cliente-servidor-aplicaciones externas

Capítulo 6 - Proceso de Desarrollo

El proceso de desarrollo involucra a tres actores diferentes: analista, diseñador y transformador de modelos.

El proceso comienza con la definición del diagrama de casos de uso de la aplicación por parte del analista. A continuación, utiliza un modelo de análisis para describir esos casos de uso. El modelo de análisis se representa con la construcción de un diagrama de clases para la vista estática y un diagrama de comunicación para la vista dinámica.

Antes de poder comenzar las transformaciones de los modelos, es necesario definir el modelo de marcas. Esta tarea es llevada a cabo por el diseñador.

Finalmente el modelo de análisis y el modelo de marcas son enviados al Transformador de Modelos. Este realiza cuatro transformaciones en dos etapas:

Transformación de modelo de análisis a modelo de arquitectura independiente de plataforma

1. Primero hace una transformación de diagrama de comunicación de modelo de análisis a diagrama de componentes de modelo de arquitectura.
2. Luego realiza la transformación de diagrama de comunicaciones de modelo de análisis a diagrama de comunicaciones de modelo de arquitectura.

Transformación de modelo de arquitectura independiente de plataforma a modelo de arquitectura de tecnología específica

1. Utilizando los diagramas de componentes que resulta de la transformación anterior, realiza una transformación de diagrama de componentes de arquitectura independiente de plataforma a diagrama de componentes de arquitectura de una plataforma específica. (Ajax, GWT, etc.)
2. Utilizando los diagramas de comunicaciones que resulta de la transformación anterior, realiza una transformación de diagrama de comunicación de arquitectura independiente de plataforma a diagrama de comunicaciones de arquitectura de una plataforma específica. (Ajax, GWT, etc.)

Es importante aclarar que el mapeo a modelo de arquitectura específico mencionado se incluye en el proceso de desarrollo por completitud pero no será contemplado en éste trabajo.

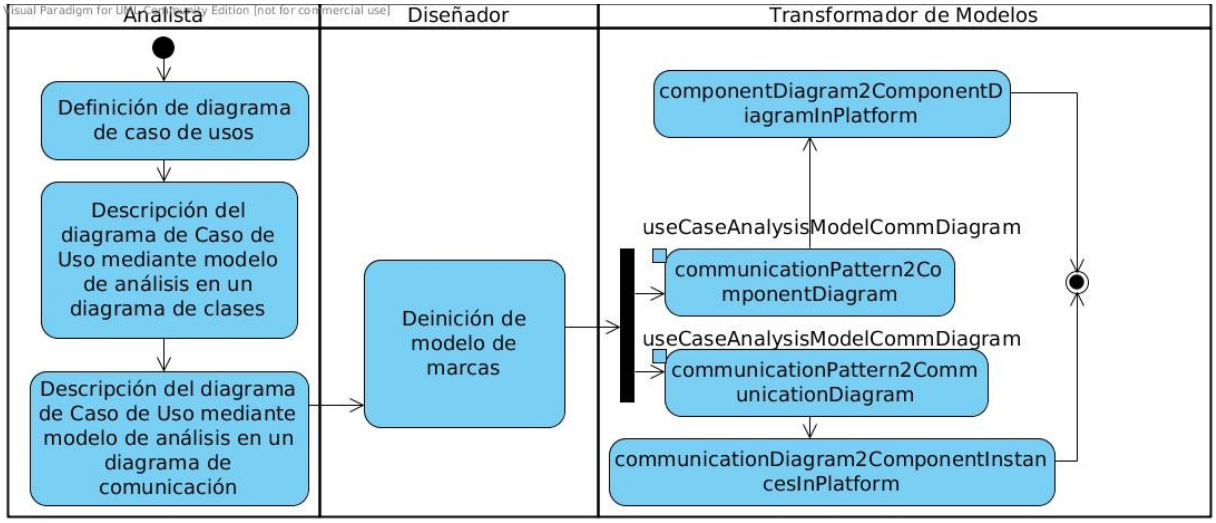

Figura 6-1: Proceso de desarrollo representado a través de un diagrama de actividades UML

Capítulo 7 - Transformación de Modelos de Análisis a Modelos de Arquitectura

7.1 Introducción

Uno de los objetivos de este trabajo es el de generar una aplicación RIA de manera automática a partir del modelo de análisis. Sólo contemplaremos el primer paso hacia este objetivo: transformar el modelo de análisis a un modelo de arquitectura RIA independiente de tecnología.

Intuitivamente, puede pensarse que la transformación debería ser desde un diagrama de clases del modelo de análisis a un diagrama de componentes del modelo de arquitectura para la vista estática y de un diagrama de comunicación del modelo de análisis a un diagrama de comunicación del modelo de arquitectura para la vista dinámica. Sin embargo, para este trabajo se decidió que ambas transformaciones partan desde un diagrama de comunicación del modelo de análisis. La razón es que los diagramas de comunicación contienen información sobre el orden de los mensajes, lo cual permite identificar con mayor facilidad cuál es el patrón de comunicación subyacente.

Una vez identificado, podemos iterar los mensajes del diagrama de análisis en orden e ir generando los componentes del modelo de arquitectura a medida que es necesario. Esto es cierto tanto para la transformación de la vista estática como de la dinámica. Por otro lado, este tipo de diagramas provee información sobre la cantidad de instancias que hay de cada objeto en tiempo de ejecución, y con esa información podemos saber cuantas instancias debemos generar para el modelo de arquitectura.

Este capítulo considera dos transformaciones:

- Transformación de diagramas de comunicación de análisis a diagramas de componentes
- Transformación de diagramas de comunicación de análisis a diagramas de comunicación de arquitectura.

7.2 Modelo de marcas empleado

Esta transformación tiene como entrada un modelo de comunicación de análisis, representado mediante un grafo: Los nodos del grafo representan los objetos del modelo de análisis. Los arcos representan los mensajes entre los objetos.

Además la transformación se apoya en un modelo de marcas que contempla las siguientes informaciones:

- Asociación entre entity y storage type (entity, storageType) que provee información sobre que tipo de almacenamiento debe generar cada elemento <<entity>> del modelo de análisis. El primer elemento del par es un objeto entity del modelo de análisis. El segundo elemento del par puede tener tres valores: 'persistent', 'volatil' o 'both'. El primero indica que el objeto <<entity>> mapea a un elemento de almacenamiento persistente en el servidor. El segundo, que mapea a un elemento de almacenamiento temporal en el cliente. El tercero indica que mapea tanto a un elemento de almacenamiento persistente en el servidor como a un elemento de almacenamiento temporal en el cliente.
- Asociación entre entity y actor (entity, actor) que provee información sobre qué tipo de objetos <<entity>> que deben generarse para cada actor. Debido a que en un caso de uso pueden participar varios actores, es necesario indicar a cuál corresponde determinado objeto generado a partir de un objeto <<entity>>. Por ejemplo, en el patrón client-server-clients, debemos saber que objetos <<entity>> generan componentes para al cliente generador del evento y que cuales generan componentes para los clientes subscriptores.
- Valor booleano true o false, indica si hay eventos compuestos del lado del servidor.
- Asociación de actor a propiedad booleana composedEvent (actor, {true | false}), indica en qué clientes hay eventos compuestos y en cuales no.

La transformación se implementa mediante un algoritmo imperativo que para un modelo de comunicación que satisface las reglas de algún patrón de comunicación genera los elementos correspondientes del diagrama de arquitectura de componentes. En general, los objetos del modelo de análisis se mapean a componentes y los mensajes se mapean a funciones exportadas por las interfaces de esas componentes.

7.3 Transformación de Modelos de Comunicación de Análisis a Diagramas de Componentes de Arquitectura RIA.

7.3.1 Detección e iteración del patrón de comunicación a partir del modelo de análisis

En transformación de modelos, las transformaciones se dividen en reglas. Cada regla tiene un input, una condición y un output. Para cada patrón de comunicación tenemos una regla cuya condición es que el modelo de entrada respete ese patrón. La función communicationPattern2ComponentDiagram es la función principal de este mapeo. Su propósito es generar cada una de las componentes del modelo de arquitectura a partir de los objetos del modelo de análisis. Lo primero que hace es

identificar cuál es el patrón de comunicación correspondiente al modelo de análisis. Luego, dependiendo del patrón detectado, delega la transformación a una función específica para ese patrón.

```
function communicationPattern2ComponentDiagram(analysisModel,
storageTypes, storageTypeActor, serverComposedEvents,
clientComposedEvents)
begin
    var communicationPatterns = parseCommunicationPatterns(analysisModel)

    if communicationPattern->type of 'Client-Server-Clients' then

clientServerClientsCommPattern2ComponentDiagram(communicationPattern,
storageTypes, storageTypeActor, serverComposedEvents,
clientComposedEvents)
    elseif communicationPattern->type of 'ExternalApp-Server-Clients' then

exteralAppServerClientsCommPattern2ComponentDiagram(communicationPattern,
storageTypes, storageTypeActor, serverComposedEvents,
clientComposedEvents)
    elseif communicationPattern->type of 'Client-Server' then
        clientServerCommPattern2ComponentDiagram(communicationPattern,
storageTypes, storageTypeActor, serverComposedEvents,
clientComposedEvents)
    elseif communicationPattern->type of 'Client-Server-ExternalApp' then

clientServerExternalAppsCommPattern2ComponentDiagram(communicationPattern,
storageTypes, storageTypeActor, serverComposedEvents,
clientComposedEvents)
    fi
end
```

Tabla 7-1: Función principal de transformación de diagrama de comunicaciones de análisis a diagrama de componentes de arquitectura indepentendiente de plataforma.

7.3.2 Transformación de los patrones de comunicación

En cada patrón de comunicación, los objetos interactúan de manera diferente. Para cada objeto del modelo de análisis, se generan uno o más componentes del modelo de arquitectura, dependiendo del tipo de objeto, ya sea <<boundary>>, <<control>> o <<entity>>. En primer lugar se generan los paquetes del lado del cliente y del lado del servidor donde se van a colocar las componentes generadas para el modelo de arquitectura. Para cada uno de esos paquetes se genera un grafo que contiene como nodos las componentes y las interfaces ofrecidas por estas componentes y como arcos las relaciones de dependencia <<use>> y de herencia entre componentes e interfaces.

Primero se generan las componentes para los objetos <<boundary>>, luego para los objetos <<control>> y finalmente para los objetos <<entity>>. Finalmente se crean las relaciones de tipo <<inherit>> y de tipo <<use>> entre los componentes y las interfaces. Para cada componente generado, se crean las interfaces que ofrece y un objeto.

7.3.3 Generación de Componentes de Arquitectura para el patrón Client-Server

La Figura 7-1 y 7-2 muestran el resultado del mapeo de un diagrama de comunicaciones del modelo de análisis a diagrama de componentes del modelo de arquitectura. El escenario utilizado es un usuario que agrega un producto a su carrito de compras y el estado del carrito es almacenado.

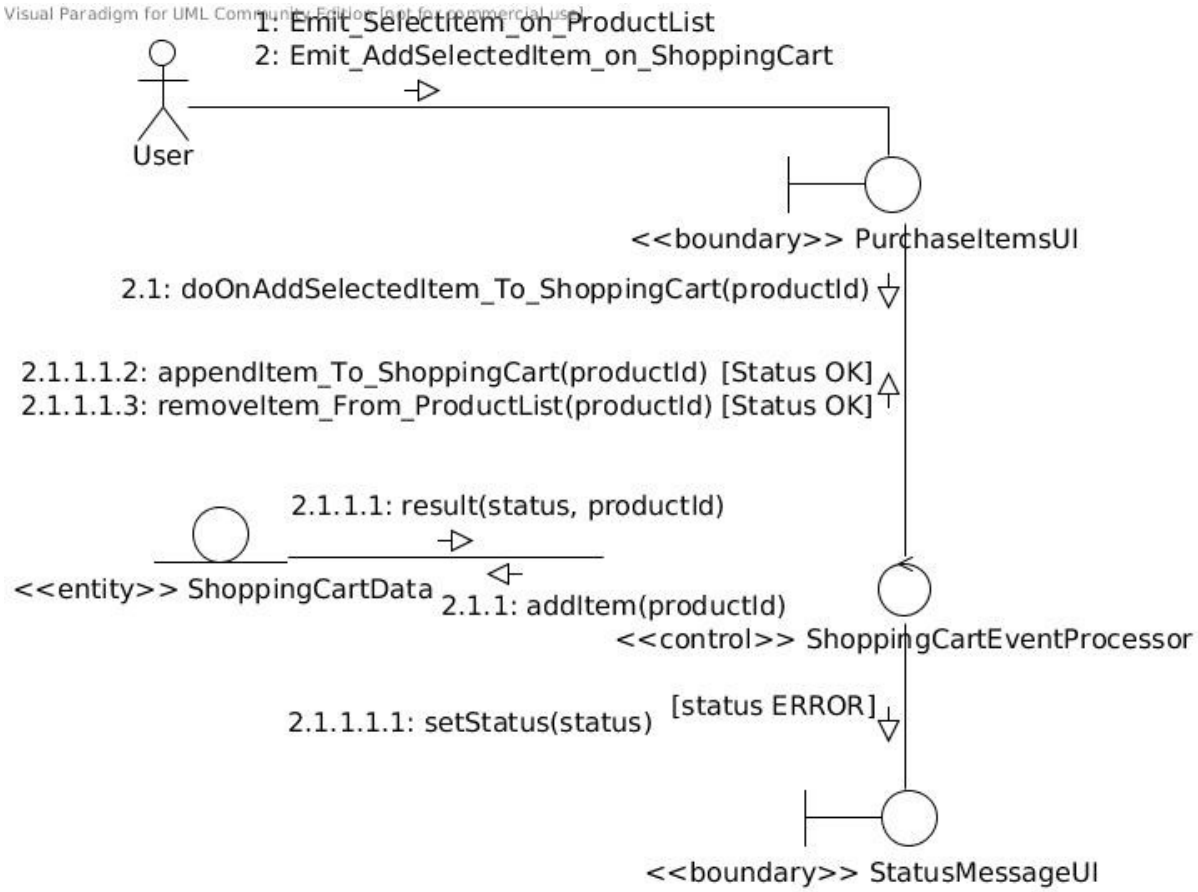

Figura 7-1: Diagrama de comunicación de un caso de uso que sigue el patrón client-server.

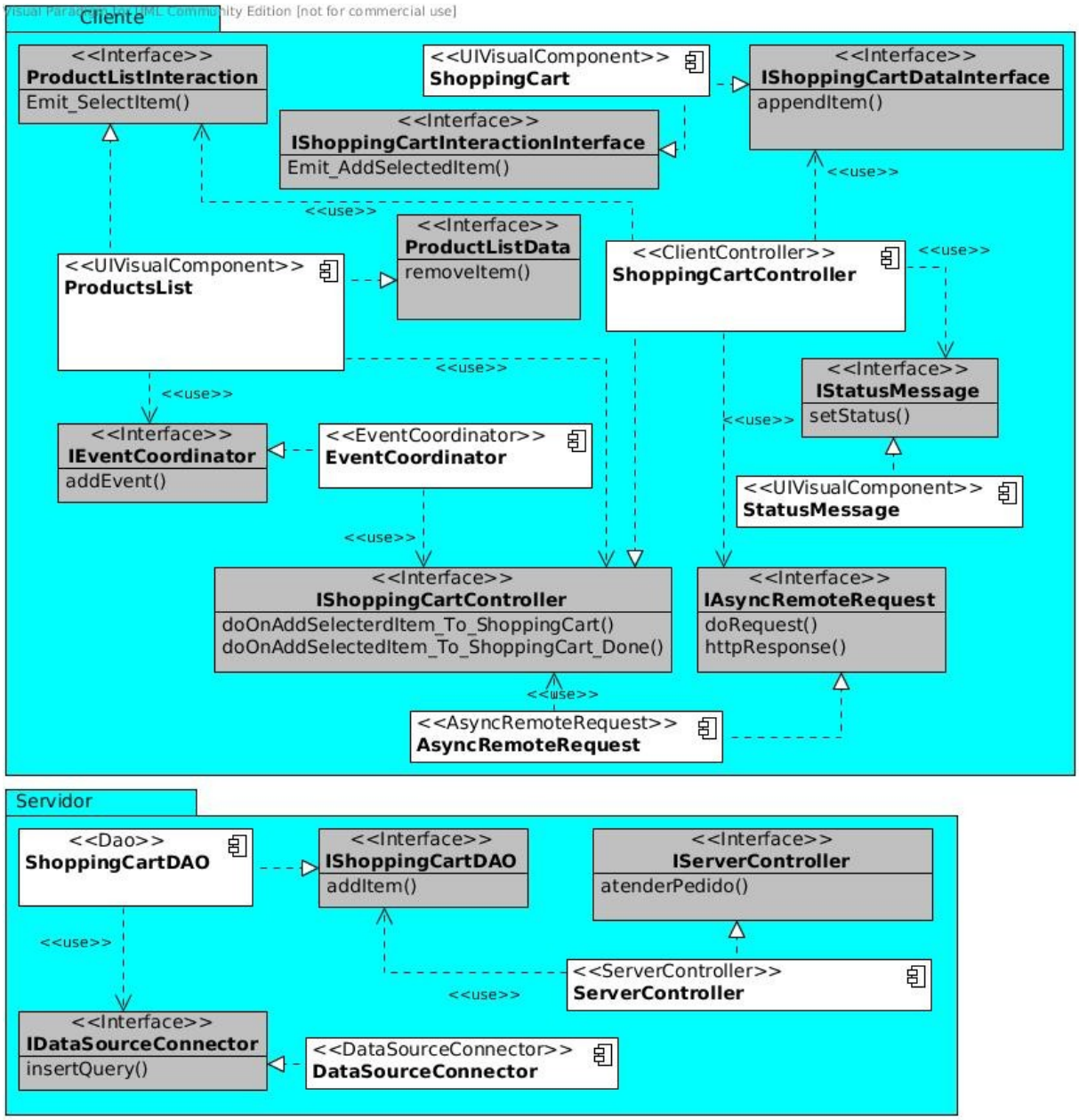

Figura 7-2: Resultado del mapeo a componentes del diagrama de la Figura 7-1.

Aplicando la regla de transformación boundaryObjects2UIVisualComponents al objeto <<boundary>> ShoppingCartUI se transforma en ProductList, ShoppingCart y StatusMessage ambos con estereotipo UIVisualComponent. Luego, aplicando controlObject2ComponentsForClientServerPattern al objeto <<control>> ShoppingCartEventProcessor se generan las componentes EventCoordinator, ShoppingCartController, AsyncRemoteRequest y ShoppingCartServerController. Por último, en este caso el modelo de marcas indica que el objeto <<entity>> ShoppingCartData utiliza almacenamiento persistente. Luego, aplicando la regla de transformación generateEntityObject2PesistentStorageComponentComponent se transforma en ShoppingCartDAO y DataSourceConnector.

A continuación podemos ver el pseudo-código correspondiente a esta transformación.

```
function clientServerCommPattern2ComponentDiagram(communicationPattern,
storageTypes, storageTypeActor, serverComposedEvents,
clientComposedEvents)
begin
    var clientPackages = generateClientPackages(communicationPattern)
    var serverPackage = generateServerPackage()

    var generatedUIVisualComponents = []
    foreach object in communicationPattern->objects
    do
        if object->stereotype == 'boundary' and object->type == 'UI' then
            boundaryObjects2UiVisualComponents(object, clientPackages,
generatedUIVisualComponents)
        elseif object->stereotype == 'boundary' and object->type ==
'Timer' then
            boundaryObject2TimerComponent(object, clientPackages)
        elseif object->stereotype == 'boundary' and object->type ==
'ConectorProtocoloRed' then
            boundaryObject2ProtocoloRedComponent(object, serverPackage)
        elseif object->stereotype == 'control' then

controlObject2ComponentsForClientServerPattern(communicationPattern,
clientPackages, serverPackage, clientComposedEvents)
            fi
        elseif object->stereotype == 'entity' then
            if storageTypes[entity] == 'persistent' or
storageTypes[entity] == 'both' then

generateEntityObject2PesistentStorageComponent(serverPackage, object)
            fi
            if storageTypes[entity] == 'volatil' or storageTypes[entity]
== 'both' then

entityObject2VolatilStorageComponent(storageTypeActor[entity],
clientPackages, object)
            fi
        fi
    od
    generateClientUseRelationships(communicationPattern, clientPackages)
    generateServerUseRelationships(serverPackage,
serverEventAdministatorPackage)
end
```

Tabla 7-2: Función de transformación de diagrama de comunicaciones de modelo de análisis a diagrama de componentes de modelo de arquitectura intependiente de plataforma para el patrón de comunicaciones Cliente-Servidor.

7.3.4 Transformación de modelos de análisis respetando el patrón Client-Server-Clients a arquitectura de componentes

Las Figura 7-3 y 7-4 ilustran el resultado del mapeo de un caso de uso que sigue este patrón de comunicación. En este caso, es escenario es una aplicación chat, donde un usuario envía un mensaje a uno o más usuarios.

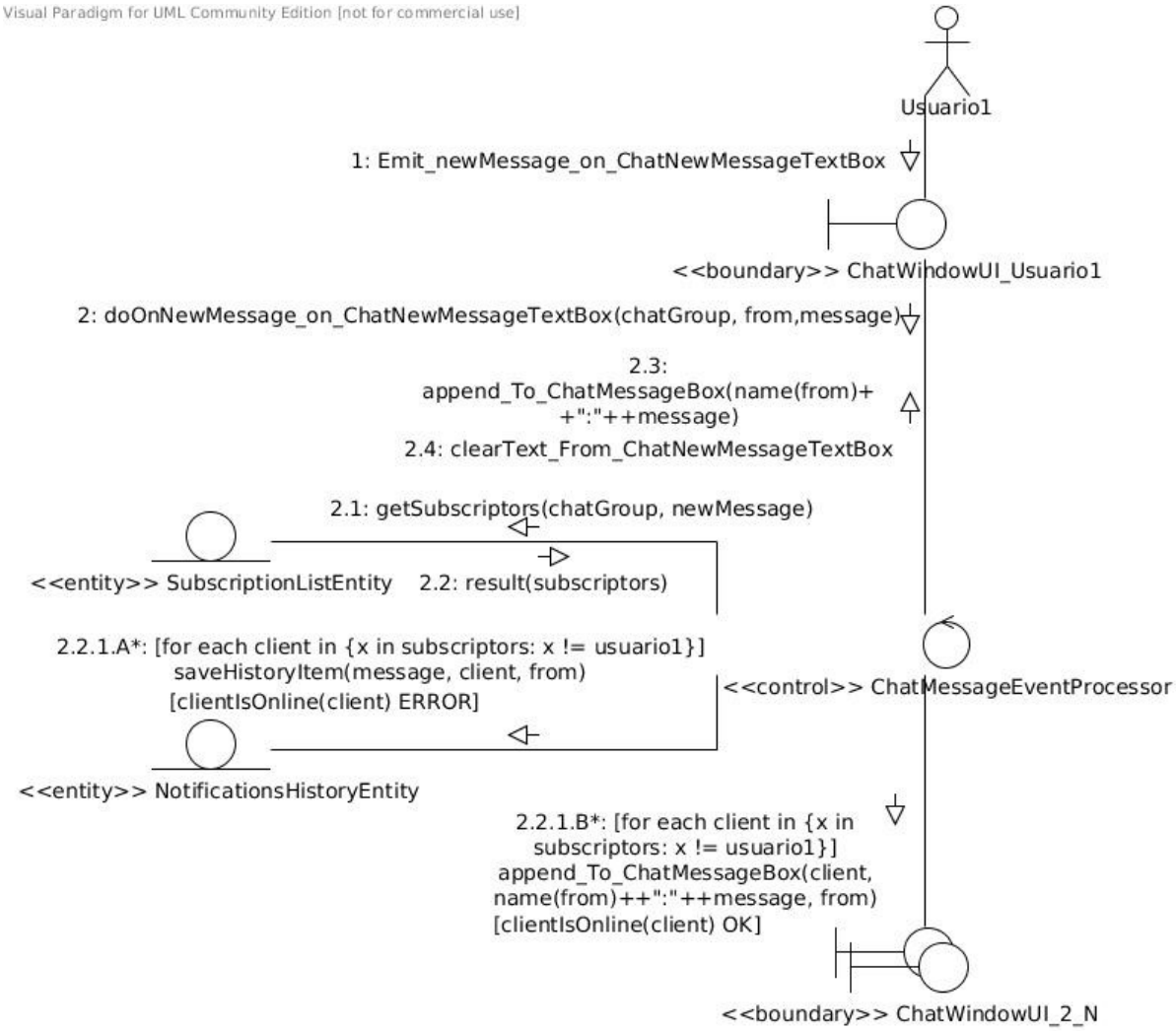

Figura 7-3: Diagrama de comunicación de un caso de uso que sigue el patrón client-server-clients.

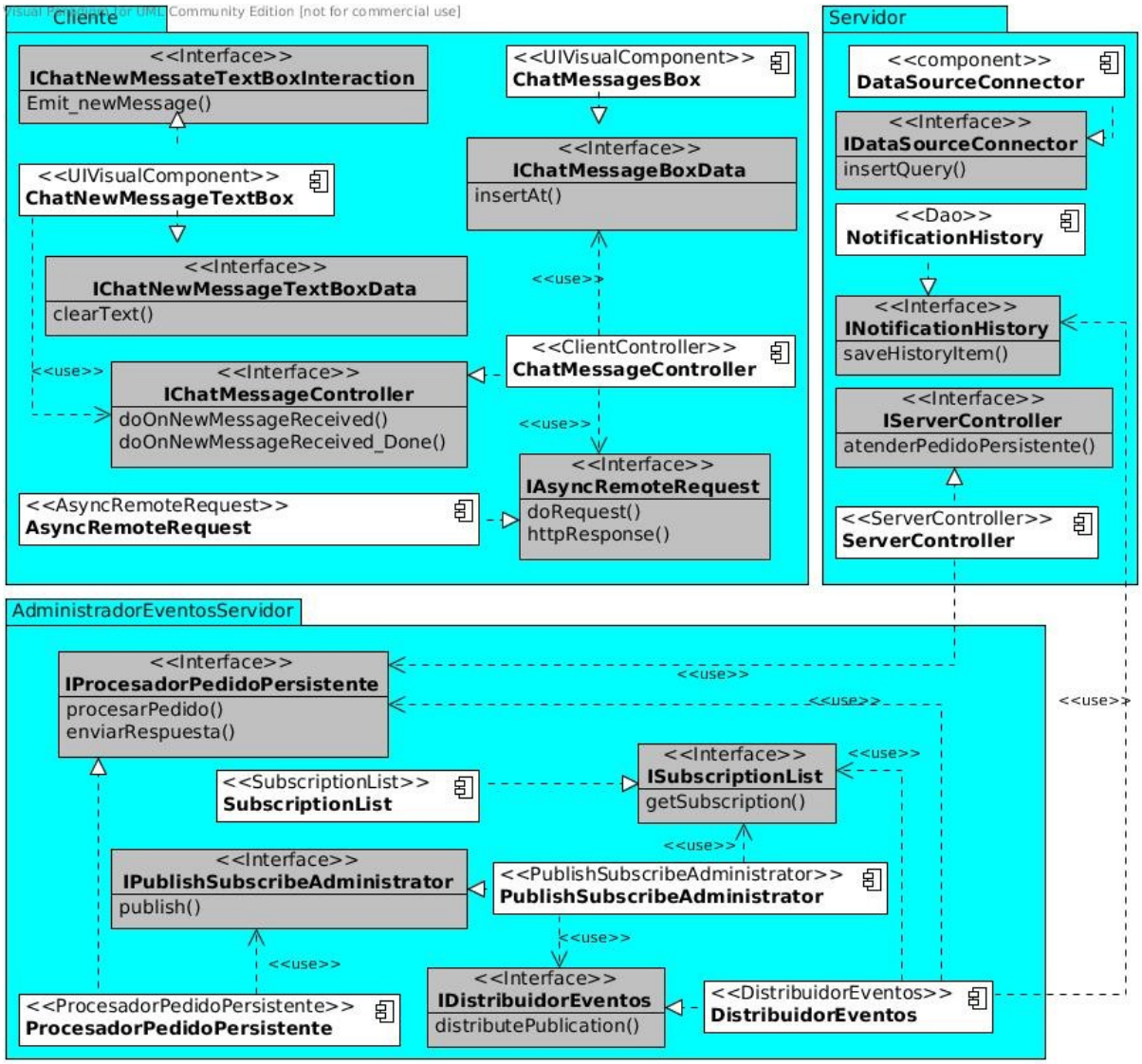

Figura 7-4: Resultado del mapeo de componentes del diagrama de la Figura 7-3.

Aplicando la regla de transformación boundaryObjects2UIVisualComponents al objeto <<boundary>> ChatWindowUI_Usuario1 se transforma en <<UIVisualCompo-Component>> ChatNewMessageTextBox y <<UIVisualComponent>> ChatMessageBox. Luego, aplicando controlObject2ComponentsForClientServerClientsPattern al objeto <<control>> ChatMessageEventProcessor se generan las componentes <<ClientController>> ChatMessageController, <<ProcesadorPedidPersistente>> ProcesadorPedidoPersistente, <<PublishSubscribeAdministrator>> PublishSubscribeAdministrator y <<DistribuidorEventos>> DistribuidorEventos. Por último, aplicando la transformación generateEntityObject2PesistentStorage-ComponentComponent, el objeto <<entity>> SubscriptionList se transforma en <<SubscriptionList>> SubscriptionList y el objeto <<entity>> Notifications-HistoryEntity se transforma en <<DAO>>NotificationsHistory.

```
function
clientServerClientsCommPattern2ComponentDiagram(communicationPattern,
storageTypes, storageTypeActor, serverComposedEvents,
clientComposedEvents)
begin

    var clientPackages = generateClientPackages(communicationPattern)

    var serverPackage = generateServerPackage()
    var serverEventsAdministratorPackage =
generateAdministradorEventosServidorPackage()
    var serverAndServerEventsPackage =
generateServerAndServerEventsPackage()

    var generatedUIVisualComponents = []

    foreach object in communicationPattern->objects
    do
        if object->stereotype == 'boundary' and object->type ==
'ExternalApp' then
            boundaryObject2ExternalAppInterface()
        fi

        if object->stereotype == 'boundary' and object->type == 'UI' then
            boundaryObjects2UiVisualComponents(object, clientPackages,
generatedUIVisualComponents)

        elseif object->stereotype == 'boundary' and object->type ==
'Timer' then
            boundaryObject2TimerComponent(object, clientPackages)
        elseif object->stereotype == 'boundary' and object->type ==
'ConectorProtocoloRed' then
            boundaryObject2ProtocoloRedComponent(object, serverPackage)
        elseif object->stereotype == 'control' then
controlObject2ComponentsForClientServerClientsPattern(communicationPattern
, clientPackages, serverPackage, serverEventsAdministratorPackage,
serverComposedEvents)
        elseif object->stereotype == 'entity' then
            if storageTypes[entity] == 'persistent' or
storageTypes[entity] == 'both' then

generateEntityObject2PesistentStorageComponentComponent(serverPackage,
object)
            fi
            if storageTypes[entity] == 'volatil' or storageTypes[entity]
== 'both' then

entityObject2VolatilStorageComponent(storageTypeActor[entity],
clientPackages, object)
            fi
        fi
        markAsVisited(object)
    od
    generateClientUseRelationships(communicationPattern, clientPackages)
    generateServerUseRelationships(serverPackage)

generateServerEventAdministratorUseRelationships(serverEventAdministratorP
ackage)

generateServerAndServerEventsUseRelationships(serverAndServerEventsPackage
, serverPackage,  serverEventAdministratorPackage)
end
```

Tabla 7-3: Función de transformación de diagrama de comunicaciones de modelo de análisis a diagrama de componentes de modelo de arquitectura intependiente de plataforma para el patrón de comunicaciones Cliente-Servidor-Clientes.

7.3.5 Transformación de modelos de análisis respetando el patrón Client-Server-ExternalApps a arquitectura de componentes

Las Figuras 7-5 y 7-6 ilustran el mapeo para este patrón.

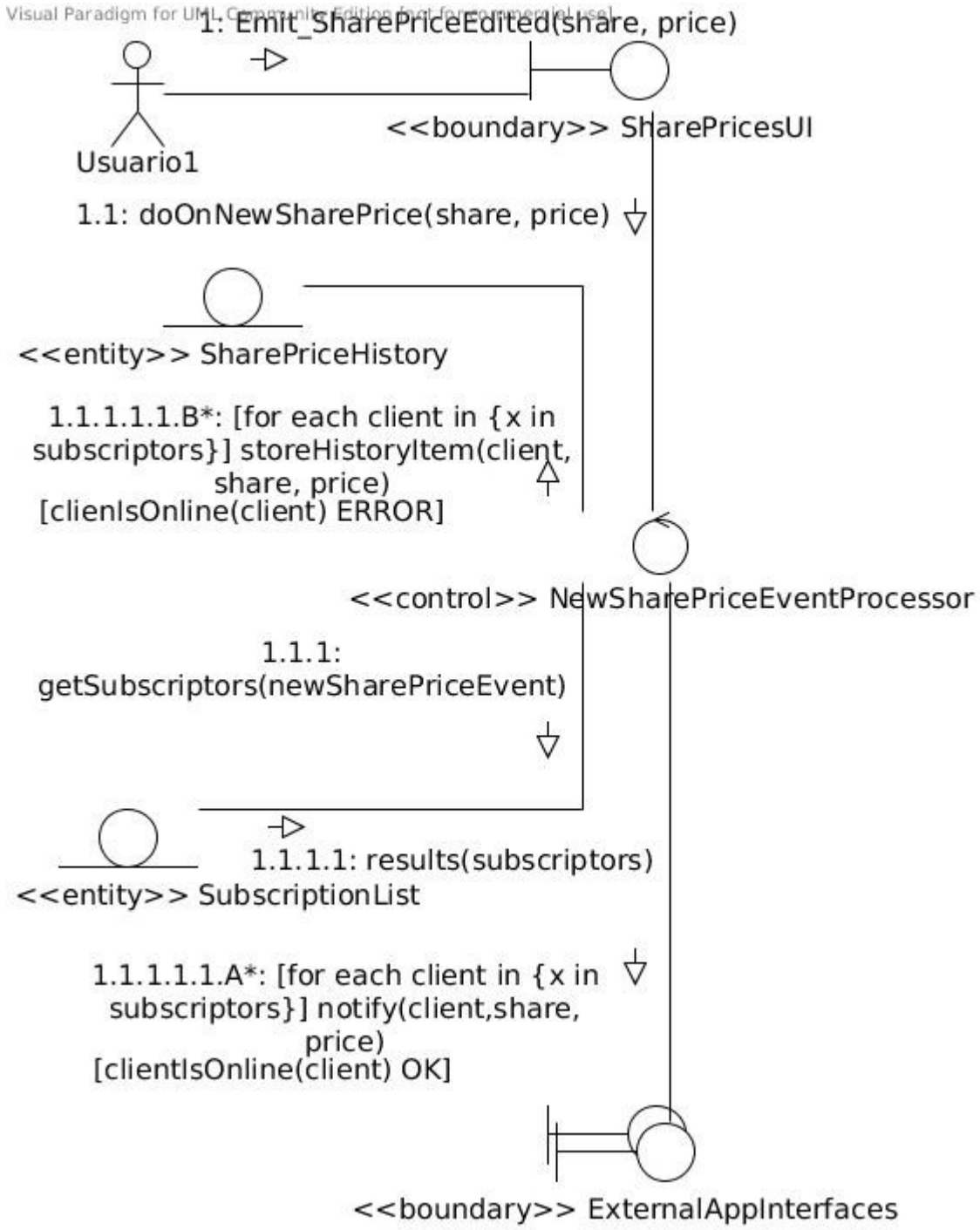

Figura 7-5: Diagrama de comunicaciones del modelo de análisis de un caso de uso con patrón de comunicación client-server-externalapps.

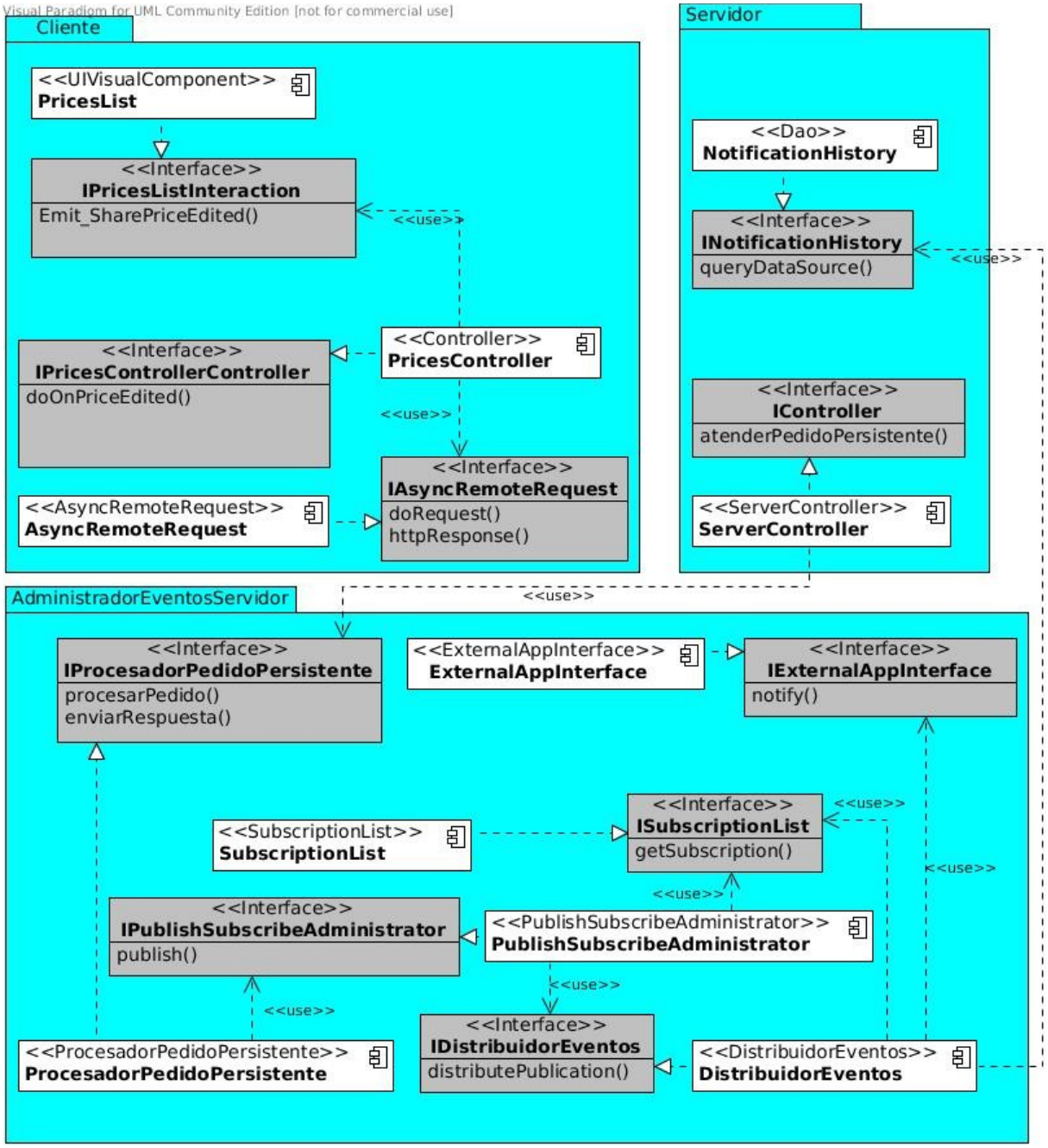

Figura 7-6: Resultado de la transformación del diagrama de comunicaciones de la Figura 7-5 a diagrama de componentes de modelo de arquitectura.

Aplicando la regla de transformación boundaryObjects2UIVisualComponents al objeto <<boundary>> SharePricesUI se transforma en <<UIVisualComponent>> PricesList. Luego, aplicando controlObject2ComponentsForClientServerClients-Pattern al objeto <<control>> NewSharePriceEventProcessor se generan las componentes <<ClientController>> PricesController, <<ProcesadorPedidPersistente>> ProcesadorPedidoPersistente, <<PublishSubscribeAdministrator>> Publish-SubscribeAdministrator y <<DistribuidorEventos>> DistribuidorEventos. Después, aplicando la transformación generateEntityObject2PesistentStorageComponent, el objeto <<entity>> SubscriptionList se transforma en <<SubscriptionList>> SubscriptionList y el objeto <<entity>> NotificationsHistoryEntity se transforma en

<<DAO>>NotificationsHistory. Finalmente, aplicando la transformación boundaryObject2ExternalAppInterface al objeto <<boundary>> ExternalAppInterface se transforma en <<ExternalAppInterface>> ExternalAppInterface.

```
function
clientServerExternalAppsCommPattern2ComponentDiagram(communicationPattern,
storageTypes, storageTypeActor, serverComposedEvents,
clientComposedEvents)
begin
    var clientPackages = generateClientPackages(communicationPattern)

    var serverPackage = generateServerPackage()
    var serverEventsAdministratorPackage =
generateAdministradorEventosServidorPackage()
    var serverAndServerEventsPackage =
generateServerAndServerEventsPackage()

    var generatedUIVisualComponents = []
    foreach object in communicationPattern->objects
    do
        if object->stereotype == 'boundary' and object->type == 'UI' then
            boundaryObjects2UiVisualComponents(object, clientController,
clientPackages, generatedUIVisualComponents)
        elseif object->stereotype == 'boundary' and object->type ==
'ExternalApp' then
            boundaryObject2ExternalAppInterface()
        elseif object->stereotype == 'boundary' and object->type ==
'Timer' then
            boundaryObject2TimerComponent(object, clientPackages)
        elseif object->stereotype == 'boundary' and object->type ==
'ConectorProtocoloRed' then
            boundaryObject2ProtocoloRedComponent(object, serverPackage)
        elseif object->stereotype == 'control' then
controlObject2ComponentsForClientServerExternalAppsPattern(communicationPa
ttern, clientPackages, serverPackage, serverComposedEvents,
clientComposedEvents, serverEventsAdministratorPackage)
        elseif object->stereotype == 'entity' then
            if storageTypes[entity] == 'persistent' or
storageTypes[entity] == 'both' then

generateentityObject2PesistentStorageComponentComponent(serverPackage,
object)
            fi
            if storageTypes[entity] == 'volatil' or storageTypes[entity]
== 'both' then

entityObject2VolatilStorageComponent(storageTypeActor[entity],
clientPackages, object)
            fi
        fi
    od
    generateClientUseRelationships(communicationPattern, clientPackages)
    generateServerUseRelationships(serverPackage)

generateServerEventAdministratorUseRelationships(serverEventAdministratorP
ackage)

generateServerAndServerEventsUseRelationships(serverAndServerEventsPackage
, serverPackage,  serverEventAdministratorPackage)
end
```

Tabla 7-4: Función de transformación de diagrama de comunicaciones de modelo de análisis a diagrama de componentes de modelo de arquitectura intependiente de plataforma para el patrón de comunicaciones Cliente-Servidor-ExternalApps.

7.3.6 Transformación de modelos de análisis respetando el patrón ExternalApps-Server-Clients a arquitectura de componentes

La Figura 7-7 y 7-8 muestran un ejemplo de este mapeo. El escenario de los diagramas es de una aplicación externa que notifica a la aplicación principal sobre un cambio en el precio de una acción de la bolsa de valores. Este evento se notifica a los clientes suscritos, actualizando el precio en sus interfaces de usuario.

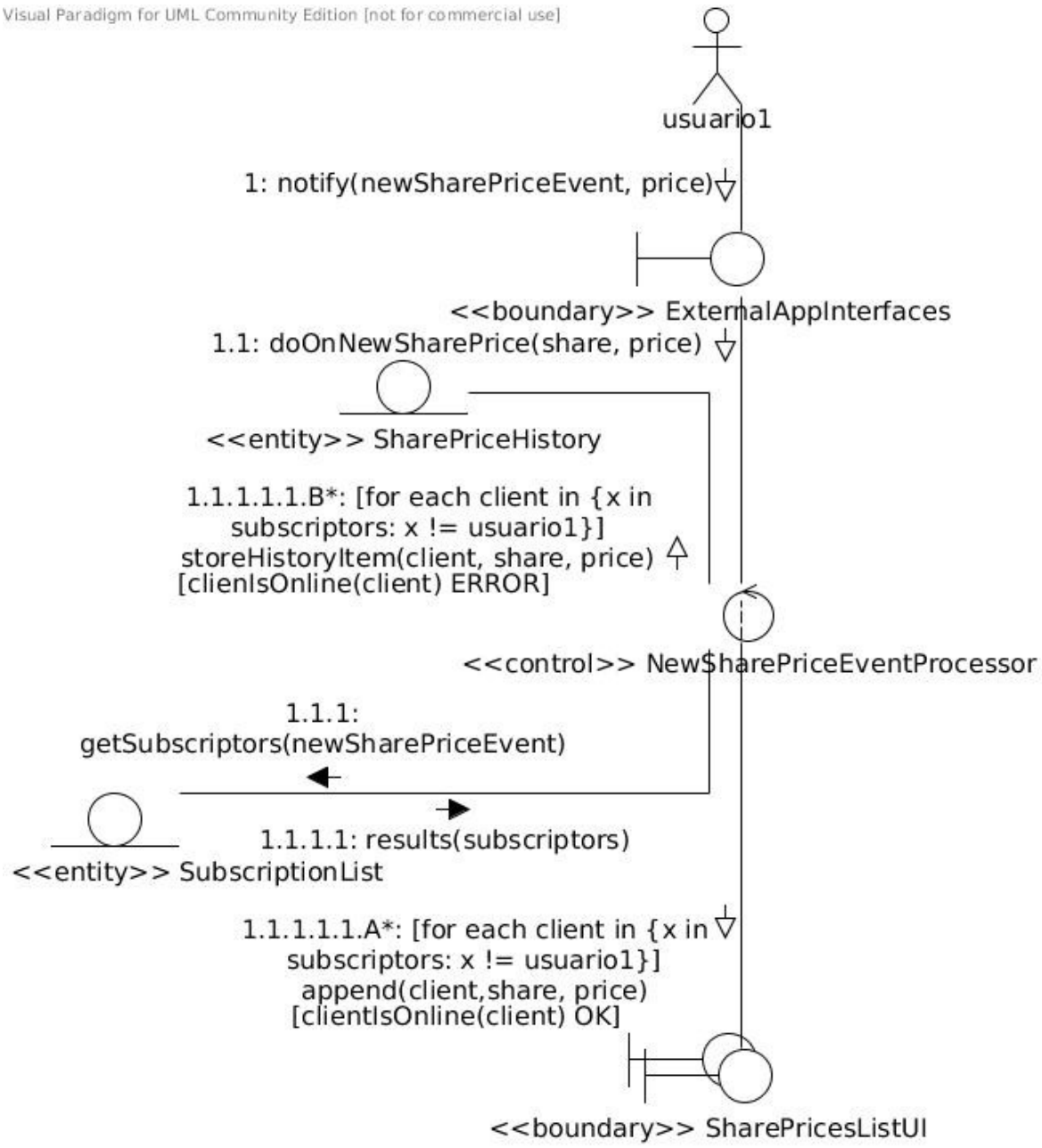

Figura 7-7: Diagrama de comunicación del modelo de análisis que respeta el patrón externalapp-server-clients.

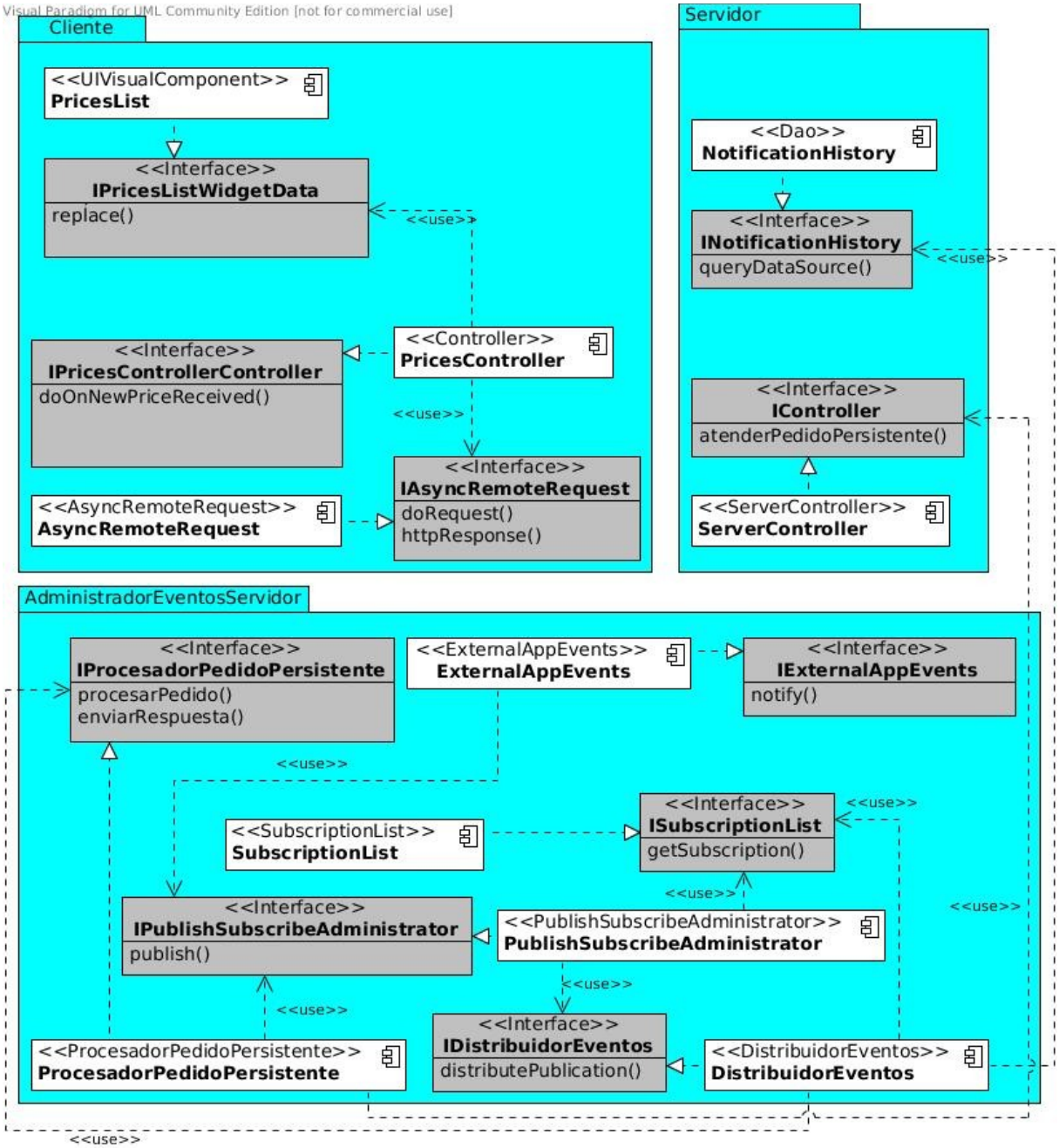

Figura 7-8: Diagrama de componentes que ilustra el modelo de arquitectura resultado del mapeo del modelo de análisis de la Figura 7-7.

Aplicando la regla de transformación boundaryObjects2UIVisualComponents al objeto <<boundary>> SharePricesUI se transforma en <<UIVisualComponent>> PricesList. Luego, aplicando controlObject2ComponentsForClientServerClients-Pattern al objeto <<control>> NewSharePriceEventProcessor se generan las componentes <<ClientController>> PricesController, <<ProcesadorPedidPersistente>> ProcesadorPedidoPersistente, <<PublishSubscribeAdministrator>> PublishSubscribeAdministrator y <<DistribuidorEventos>> DistribuidorEventos. Después, aplicando la transformación generateEntityObject2PesistentStorage-Component, el objeto <<entity>> SubscriptionList se transforma en <<SubscriptionList>> SubscriptionList y el objeto <<entity>>

NotificationsHistoryEntity se transforma en <<DAO>>NotificationsHistory. Finalmente, aplicando la transformación boundaryObject2ExternalAppInterface al objeto <<boundary>> ExternalAppInterface se transforma en <<ExternalAppInterface>> ExternalAppInterface.

```
function
exteralAppServerClientsCommPattern2ComponentDiagram(communicationPattern,
storageTypes, storageTypeActor, serverComposedEvents,
clientComposedEvents)
begin

    var clientPackages = generateClientPackages(communicationPattern)
    var serverPackage = generateServerPackage()
    var serverEventsAdministratorPackage =
generateAdministradorEventosServidorPackage()
    var serverAndServerEventsPackage =
generateServerAndServerEventsPackage()

    var generatedUIVisualComponents = []

    foreach object in communicationPattern->objects
    do
        if object->stereotype == 'boundary' and object->type ==
'ExternalApp' then
            boundaryObject2ExternalAppInterface()
        fi

        if object->stereotype == 'boundary' and object->type == 'UI' then
            boundaryObjects2UiVisualComponents(object, clientPackages,
generatedUIVisualComponents)

        elseif object->stereotype == 'boundary' and object->type ==
'Timer' then
             boundaryObject2TimerComponent(object, clientPackages)
        elseif object->stereotype == 'boundary' and object->type ==
'ConectorProtocoloRed' then
            boundaryObject2ProtocoloRedComponent(object, serverPackage)
        elseif object->stereotype == 'control' then

        elseif object->stereotype == 'entity' then
            if storageTypes[entity] == 'persistent' or
storageTypes[entity] == 'both' then

generateEntityObject2PesistentStorageComponentComponent(serverPackage,
object)
            fi
            if storageTypes[entity] == 'volatil' or storageTypes[entity]
== 'both' then

entityObject2VolatilStorageComponent(storageTypeActor[entity],
clientPackages, object)
            fi
        fi
        markAsVisited(object)
    od
    generateClientUseRelationships(communicationPattern, clientPackages)
    generateServerUseRelationships(serverPackage)

generateServerEventAdministratorUseRelationships(serverEventAdministratorP
ackage)

generateServerAndServerEventsUseRelationships(serverAndServerEventsPackage
, serverPackage,  serverEventAdministratorPackage)
end
```

Tabla 7-5: Función de transformación de diagrama de comunicaciones de modelo de análisis a diagrama de componentes de modelo de arquitectura intependiente de plataforma para el patrón de comunicaciones Cliente-Servidor-ExternalApps.

7.3.7 Generación de paquetes cliente del modelo de arquitectura.

La función *generateClientPackages* es la encargada de generar los paquetes Cliente del modelo de arquitectura. Estos paquetes son los que contendrán los componentes del lado cliente generados para el modelo de arquitectura. Por cada actor encontrado en el modelo de análisis, generamos un paquete cliente.

```
function generateClientPackages(communicationPattern)
begin
    var clientPackages = []
    // Generamos un paquete cliente para cada actor encontrado en el
modelo
    // de análisis.
    foreach actor in communicationPattern->actors
    do
        clientPackages[actor] = generateClientPackage()
    od

    return clientPackages
end
```

Tabla 7-6: Función de generación de los paquetes cliente que contienen a las componentes del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.8 Transformación de Objetos <<boundary>> de tipo UI

7.3.8.1 Regla de transformación boundaryObjects2UiVisualComponents

La función *boundaryObjects2UiVisualComponents* genera los UIVisualComponent para el modelo de arquitectura a partir de los objetos <<boundary>> de tipo UI encontrados en el modelo de análisis. Aquí podemos dividir el procesamiento en dos casos:

- Obtener componentes visuales a partir de eventos recibidos de un Usuario o Aplicacion Externa:
    - Cuando ninguno de los eventos recibidos por un objeto <<boundary>> de tipo UI tiene un nombre de componente contenido dentro de su nombre y tampoco hay mensajes desde objetos de control sobre algun tipo especifico de estructura visual, el objeto <<boundary>> de tipo UI mapea directamente a un UIVisualComponent del modelo de arquitectura. En este caso, se genera un UIVisualComponent con el mismo nombre que objeto <<boundary>> de tipo UI.
    - Cuando alguno de los eventos recibidos por un objeto <<boundary>> de tipo UI tiene un nombre de componente contenido dentro de su nombre (p.e.: Emit_EnterPressed_on_TextInput), entonces ese objeto se descompone en más de un UIVisualComponent durante el mapeo a el modelo de arquitectura. En este caso, los componentes a generar se obtienen mediante el parseo del nombre de los eventos asociados al objeto <<boundary>> de tipo UI y recorriendo los mensajes que

provienen del objeto <<control>>.

- Obtener componentes visuales a partir de mensajes de objetos de control.:
- Si el mensaje corresponde a un mensaje de alguno de los tipos de estructura visual (árbol, tabla, lista, etc., explicados más abajo en la función de generación de la interfaz de datos de un UIVisualComponent), entonces se genera una componente UIVisualComponent de ese tipo.

Luego se crean tanto la interfaz de interacción como la interfaz de datos, para cada uno de los UIVisualComponent y las respectivas relaciones <<inherit>> para conectar las componentes con las interfaces.

En la Figura 7-1 y Figura 7-2 se puede ver el resultado de aplicar esta transformación.

```
function boundaryObjects2UiVisualComponents (object, clientPackages,
generatedUIVisualComponents)
begin
    // Iteramos todos los eventos sobre el objeto <<boundary>> de tipo UI
    foreach event in object->messages->events
    do
        // Primero chequeamos que el componente no haya sido generado aún
        if not findGeneratedComponentByEvent(event,
generatedVisualComponents) then
            // Si el nombre del evento indica que se debe generar más de
un
            // UIVisualComponent, parseamos el nombre y hacemos la
generación.
            // Caso contrario, generamos un solo UIVisualComponent
            if hasTargetInName(event) then
                var targetName = getTargetFromName_helper(event->name)
                var uiVisualComponent =
generateUIVisualComponent(targetName)
            else
                var uiVisualComponent = generateUIVisualComponent(object-
>name)
            fi
            // Generamos la interfaz de datos y de interacción para cada
            // UIVisualComponent generado.
            var uiVisualComponentDataInterface =
generateUIVisualComponentDataInterface(object)
           var uiVisualComponentInteractionInterface =
generateUIVisualComponentInteractionInterface(object)
            clientPackages[object->actor]-
>addCompInterfRel(uiVisualComponent, uiVisualComponentDataInterface,
inheritArrow)
            clientPackages[object->actor]-
>addCompInterfRel(uiVisualComponent,
uiVisualComponentInteractionInterface, inheritArrow)
        fi
        // Agregamos el UIVisualComponent generado al paquete cliente
        // correspondiente.
        clientPackages[object->actor]->addCompInterfRel(uiVisualComponent,
uiVisualComponentDataInterface, inheritArrow)
        clientPackages[object->actor]->addCompInterfRel(uiVisualComponent,
uiVisualComponentInteractionInterface, inheritArrow)

        // Agregamos el UIVisualComponent generado a la lista de
componentes
        // generados, para evitar generar la misma componente 2 veces.
        generatedVisualComponents.push(uiVisualComponent)
    od
    // Ahora iteramos los mensajes que recibe el objeto <<boundary>> de
tipo UI
```

```
    // desde un objeto <<control>>.
    foreach message in object->messages
    do
        // Chequeamos que el componente no haya sido generado aún.
        if not findGeneratedComponentByMessage(message,
generatedVisualComponents) then
            // De acuerdo con el nombre del mensaje, generamos el
            // UIVisualComponent correspondiente.
            var componentName = generateComponentNameFromMessage(message)
            var uiVisualComponent =
generateUIVisualComponent(componentName)
            var uiVisualComponentDataInterface =
generateUIVisualComponentDataInterface(object)
            var uiVisualComponentInteractionInterface =
generateUIVisualComponentInteractionInterface(object)
            clientPackages[object->actor]-
>addCompInterfRel(uiVisualComponent, uiVisualComponentDataInterface,
inheritArrow)
            clientPackages[object->actor]-
>addCompInterfRel(uiVisualComponent,
uiVisualComponentInteractionInterface, inheritArrow)

            generatedVisualComponents.push(uiVisualComponent)
        fi
    od
end
```

Tabla 7-7: Función de transformación de objetos de frontera del diagrama de comunicaciones del modelo de análisis a componentes UIVisualComponent del diagrama de componentes del modelo de arquitectura.

Esta función auxiliar, utilizada en *boundaryObjects2UiVisualComponents*, se encarga de obtener el nombre del componente a generar a partir del nombre de un evento. Esto responde a una decisión de diseño que consiste en incluir los nombres de los UIVisualComponent del modelo de arquitectura en los que se descompone un objeto <<boundary>> de tipo UI, dentro de los nombres de los métodos de la interfaz de interacción del mismo.

En la Figura 7-1 y el nombre del mensaje de objeto <<boundary>> a objeto <<control>> contiene el nombre del target que indica a qué componente del modelo de arquitectura va dirigido el mensaje.

```
function getTargetFromName_helper (eventName)
begin
    return getToken(eventName, eventSource)
end
```

Tabla 7-8: Esta función permite obtener el nombre de la componente a generar en el diagrama de comunicaciones del modelo de arquitectura a partir del nombre de un mensaje del modelo de análisis.

7.3.8.2 Regla de transformación boundaryObject2ExternalAppInterface

A partir de un objeto <<boundary>> de tipo ExternalAppInterface, genera una componente ExternalAppEvents, su interfaz y la relación de tipo <<inherit>> en el modelo de arquitectura.

En la Figura 7-5 y Figura 7-6 se puede ver el resultado de aplicar esta transformación.

```
function boundaryObject2ExternalAppInterface(object, serverPackage)
begin
    var externalAppEvents = boundaryObject2ExternalAppInterface()
    var externalAppEventsInterface =
boundaryObject2ExternalAppInterfaceInterface()
    administradorEventosServidorPackage-
>addCompInterfRel(externalAppEvents, externalAppEventsInterface,
inheritArrow)
end
```

Tabla 7-9: Función de transformación de objetos de frontera del diagrama de comunicaciones del modelo de arquitectura a componentes ExternalAppInterface del diagarama de componentes del modelo de arquitectura independiente de plataforma.

7.3.8.3 Regla de transformación boundaryObject2TimerComponent

A partir de un objeto <<boundary>> de tipo Timer, genera una componente Timer, su interfaz, y la relación de tipo <<inherit>> en el modelo de arquitectura.

```
function boundaryObject2TimerComponent(clientPackages)
begin
    var timer = generateTimer()
    var timerInterface = generateTimerInterface()
    clientPackages[object->actor]->addCompInterfRel(timer, timerInterface,
inheritArrow)
end
```

Tabla 7-10: Función de transformación de objeto de frontera de diagrama de comunicación de análisis a componente Timer de modelo de arquitectura.

7.3.8.4 Regla de transformación boundaryObject2ProtocoloRedComponent

A partir de un objeto <<boundary>> de tipo ConectorProtocoloRed, genera una componente ConectorProtocoloRed, su interfaz y la relación de tipo <<inherit>> en el modelo de arquitectura.

```
function boundaryObject2ProtocoloRedComponent(object, serverPackage)
begin
    var conectorProtocoloRed = generateConectorProtocoloRed(object-
>protocol)
    var conectorProtocoloRedInterface =
generateConectorProtocoloRedInterface(object->protocol)
    serverPackage->addCompInterfRel(conectorProtocoloRed,
conectorProtocoloRedInterface, inheritArrow)
end
```

Tabla 7-11: Transformación de objeto de frontera del diagrama de comunicaciones del modelo de análisis a componente ConectorProtocoloRed del diagrama de componentes del modelo de arquitectura.

7.3.9 Reglas de transformación de objetos <<control>>

7.3.9.1 Regla de transformación controlObject2ComponentsForClientServerPattern

A partir de un objeto de <<control>>, genera las componentes de modelo de arquitectura ClientControllers y ServerController, sus interfaces y los agrega en los paquetes cliente y servidor correspondientes. Se utiliza cuando el patrón de comuniación detectado en el modelo de análisis es Client-Server.

En la Figura 7-1 y Figura 7-2 se puede ver el resultado de aplicar esta transformación en un caso de uso que respeta el patrón Cliente-Servidor.

```
function
controlObject2ComponentsForClientServerPattern(communicationPattern,
clientPackages, serverPackage, clientComposedEvents)
begin
    generateClientControllers(communicationPattern, clientPackages)
    generateServerController(serverPackage)
    generateSerializador(serverPackage)

    generateClientEventCoordinators(communicationPattern,
clientComposedEvents, clientPackages)
end
```

Tabla 7-12: Función de transformación de objetos de control del diagrama de componentes del modelo de análisis a las componentes correspondientes del diagrama de componentes del modelo de arquitectura para el patrón cliente-servidor.

7.3.9.2 Regla de transformación controlObject2ComponentsForClientServerClientsPattern

A partir de un objeto de <<control>>, genera las componentes de modelo de arquitectura ClientControllers y ServerController, sus interfaces y los agrega en los paquetes cliente y servidor correspondientes. Además, genera las componentes de manejo de eventos en el servidor: SubscriptionList, ProcesadorPedidoPersistente, PublishSubscribeAdministrator, DistribuidorEventos y EventCoordinator (si hay eventos compuestos). Se utiliza cuando el patrón de comuniación detectado en el modelo de análisis es Client-Server-Clients.

En la Figura 7-3 y Figura 7-4 se puede ver el resultado de aplicar esta transformación en un caso de uso que respeta el patrón Client-Server-Clients.

```
function
controlObject2ComponentsForClientServerClientsPattern(communicationPattern
, clientPackages, serverPackage, serverEventsAdministratorPackage,
serverComposedEvents)
begin
    generateClientControllers(communicationPattern, clientPackages)
    generateServerController(serverPackage)
    generateSerializador(serverPackage)
    generateSubscriptionList(serverEventsAdministratorPackage)
    generateProcesadorPedidoPersistente(serverEventsAdministratorPackage)

generatePublishSubscribeAdministrator(serverEventsAdministratorPackage)
    generateDistribuidorEventos(serverEventsAdministratorPackage)

    if serverComposedEvents == true then
        generateServerEventCoordinator(serverEventsAdministratorPackage)
    fi
    generateClientEventCoordinators(communicationPattern,
clientComposedEvents, clientPackages)
end
```

Tabla 7-13: Función de transformación de objetos de control del diagrama de componentes del modelo de análisis a las componentes correspondientes del diagrama de componentes del modelo de arquitectura para el patrón cliente-servidor-clients.

7.3.9.3 Regla de transformación controlObject2ComponentsForClientServerExternalAppsPattern

A partir de un objeto de <<control>>, genera las componentes de modelo de arquitectura ClientControllers y ServerController, sus interfaces y los agrega en los paquetes cliente y servidor correspondientes. Además, genera las componentes de manejo de eventos en el servidor: SubscriptionList, ProcesadorPedidoPersistente, PublishSubscribeAdministrator, DistribuidorEventos y EventCoordinator (si hay eventos compuestos). Se utiliza cuando el patrón de comuniación detectado en el modelo de análisis es Client-Server-ExternalApps.

En la Figura 7-5 y Figura 7-6 se puede ver el resultado de aplicar esta transformación.

```
function
controlObject2ComponentsForClientServerExternalAppsPattern(communicationPa
ttern, clientPackages, serverPackage, serverComposedEvents,
clientComposedEvents, serverEventsAdministratorPackage)
begin
    var clientController = generateClientControllers(communicationPattern,
clientPackages)
    generateServerController(serverPackage)
    generateSerializador(serverPackage)
    generateSubscriptionList(serverEventsAdministratorPackage)

    generateProcesadorPedidoPersistente(serverEventsAdministratorPackage)


generatePublishSubscribeAdministrator(serverEventsAdministratorPackage)
    generateDistribuidorEventos(serverEventsAdministratorPackage)

    if serverComposedEvents == true then
        generateServerEventCoordinator(serverEventsAdministratorPackage)
    fi
    if clientComposedEvents == true then
        generateClientEventCoordinators(communicationPattern,
clientComposedEvents, clientPackages)
    fi
end
```

Tabla 7-14: Función de transformación de objetos de control del diagrama de componentes del modelo de análisis a las componentes correspondientes del diagrama de componentes del modelo de arquitectura para el patrón cliente-servidor-externalapps.

7.3.9.4 Regla de transformación controlObject2ComponentsForExternalAppServerClientsPattern

A partir de un objeto de <<control>>, genera las componentes de modelo de arquitectura ClientControllers y ServerController, sus interfaces y los agrega en los paquetes cliente y servidor correspondientes. Además, genera las componentes de manejo de eventos en el servidor: SubscriptionList, ProcesadorPedidoPersistente, PublishSubscribeAdministrator, DistribuidorEventos y EventCoordinator (si hay eventos compuestos). Se utiliza cuando el patrón de comuniación detectado en el modelo de análisis es ExternalApps-Server-Clients.

En la Figura 7-7 y Figura 7-8 se puede ver el resultado de aplicar esta transformación.

```
function
controlObject2ComponentsForExternalAppServerClientsPattern(communicationPa
ttern, clientPackages, serverPackage, clientComposedEvents,
serverEventsAdministratorPackage, serverComposedEvents)
begin
    generateClientControllers(communicationPattern, clientPackages)
    generateServerController(serverPackage)
    generateSerializador(serverPackage)
    generateSubscriptionList(serverEventsAdministratorPackage)

    generateProcesadorPedidoPersistente(serverEventsAdministratorPackage)


generatePublishSubscribeAdministrator(serverEventsAdministratorPackage)
    generateDistribuidorEventos(serverEventsAdministratorPackage)

    if serverComposedEvents == true then
        generateServerEventCoordinator(serverEventsAdministratorPackage)
    fi
    if clientComposedEvents == true then
        generateClientEventCoordinators(communicationPattern,
clientComposedEvents, clientPackages)
    fi
end
```

Tabla 7-15 Función de transformación de objetos de control del diagrama de componentes del modelo de análisis a las componentes correspondientes del diagrama de componentes del modelo de arquitectura para el patrón externalapp-servidor-clientes.

7.3.10 Funciones de generación de componentes a partir de objetos <<control>>

7.3.10.1 Función de generación generateClientControllers

Esta función se encarga de generar los controladores del lado del cliente para cada uno de los actores hallados en el patrón de comunicación. Luego de generar el componente y su interfaz, se crean las relaciones de herencia. A continuación se crean los AsynRemoteRequest necesarios para que el cliente pueda lanzar pedidos asincrónicos al servidor. Finalmente se generan las relaciones inherit entre las componentes y las interfaces.

```
function generateClientControllers(communicationPattern, clientPackages)
begin
    foreach actor in communicationPattern->actors
    do
        var clientController = generateClientController(actor)
        var clientControllerInterface =
generateClientControllerInterface(actor)
        clientPackages[actor]->addCompInterfRel(clientController,
clientControllerInterface, inheritArrow)

        var asyncRemoteRequest = generatAsyncRemoteRequest(actor)
        var asyncRemoteRequestInterface =
generateAsyncRemoteRequestInterface(actor)
        clientPackages[actor]->addCompInterfRel(asyncRemoteRequest,
asyncRemoteRequestInterface, inheritArrow)
        clientPackages[actor]->addCompInterfRel(clientController,
asyncRemoteRequestInterface, useArrow)
        clientPackages[actor]->addCompInterfRel(asyncRemoteRequest,
clientControllerInterface, useArrow)
    od
end
```

Tabla 7-16: Función de generación de las componentes Controller del parquete cliente del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.10.2 Función de generación generateServerController

Esta función se engarga de generar el controlador del lado del servidor.

```
function generateServerController(serverPackage)
begin
    var serverController = generateServerController()
    var serverControllerInterface = generateServerControllerInterface()
    serverPackage->addCompInterfRel(serverController,
serverControllerInterface, inheritArrow)
end
```

Tabla 7-17: Función de generación de las componentes ServerController del parquete servidor del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.10.3 Función de generación generateSerializador

Esta función se encarga de generar el componente serializador de datos.

```
function generateSerializador(serverPackage)
begin
    var serializador = generateSerializador()
    var serializadorInterface = generateSerializadorInterface()
    serverPackage->addCompInterfRel(serializador, serializadorInterface,
inheritArrow)
end
```

Tabla 7-18: Función de generación de la componente Serializador.

7.3.10.4 Función de generación generateProcesadorPedidoPersistente

Esta función se engarga de generar, para cada actor hallado en el patrón de comunicación que requiera conexión persistente con el servidor (por estar subscripto a un evento del servidor), un procesador pedido persistente.

```
function generateProcesadorPedidoPersistente(communicationPattern,
serverEventsAdministratorPackage)
begin
    foreach communicationPattern->actors as actor
    do
        var procesadorPedidoPersistente =
generateProcesadorPedidoPersistente(actor)
        var procesadorPedidoPersistenteInterface =
generateProcesadorPedidoPersistenteInterface(actor)
        serverEventsAdministrationPackage-
>addCompInterfRel(procesadorPedidoPersistente,
procesadorPedidoPersistenteInterface, inheritArrow)
    od
end
```

Tabla 7-19: Función de generación de la componente ProcesadorPedidoPersistente para el diagrama de componentes de la arquitectura independiente de plataforma.

7.3.10.5 Función de generación generatePublishSubscribeAdministrator

Esta función es la encargada de generar el componente PublishSubscribeAdministrator, si interfaz, la relación de tipo <<inherit>> entre estas dos y agregarlo al paquete de administración de eventos en el servidor.

```
function
generatePublishSubscribeAdministrator(serverEventsAdministratorPackage)
begin
    var publishSubscribeAdministrator =
generatePublicSubscribeAdministrator()
    var publishsubscribeadministratorInterface =
generatePublishSubscribeAdministratorInterface()
    serverEventsAdministratorPackage-
>addCompInterfRel(publishSubscribeAdministrator,
publishSubscribeAdministratorInterface, inheritArrow)
end
```

Tabla 7-20: Función de generación de la componente PublishSubscribeAdministrator para el diagrama de componentes de la arquitectura independiente de plataforma.

7.3.10.6 Función de generación generateDistribuidorEventos

Esta función es la encargada de generar el componente DistribuidorEventos, su interfaz, la relación de tipo <<inherit>> entre estas dos y agregarlo al paquete de administración de eventos en el servidor.

```
function generateDistribuidorEventos(serverEventsAdministratorPackage)
begin
    var distribuidorEventos = ()
    var distribuidorEventosInterface = Interface()
    serverEventsAdministratorPackage-
>addCompInterfRel(distribuidorEventos, distribuidorEventosInterface,
inheritArrow)
end
```

Tabla 7-21: Función de generación de la componente DistribuidorEventos para el diagrama de componentes de la arquitectura independiente de plataforma.

7.3.10.7 Función de generación generateSubscriptionList

Esta función es la encargada de generar el componente SubscriptionList, su interfaz, la relación de tipo <<inherit>> entre estas dos y agregarlo al paquete de administración de eventos en el servidor.

```
function generateSubscriptionList(serverEventsAdministratorPackage)
begin
    var subscriptionList = ()
    var subscriptionListInterface = Interface()
    serverEventsAdministratorPackage->addCompInterfRel(subscriptionList,
subscriptionListInterface, inheritArrow)
end
```

Tabla 7-22: Función de generación de la componente SubscriptionList del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.10.8 Función de generación generateServerEventCoordinator

Esta función es la encargada de generar el componente ServerEventCoordinator, su interfaz, la relación de tipo <<inherit>> entre estas dos y agregarlo al paquete de administración de eventos en el servidor.

```
function generateServerEventCoordinator(serverEventsAdministratorPackage)
begin
    var serverEventCoordinator = generateServerEventCoordinator()
    var serverEventCoordinatorInterface =
generateServerEventCoordinatorInterface()
    serverEventsAdministratorPackage-
>addCompInterfRel(serverEventCoordinator, serverEventCoordinatorInterface,
inheritArrow)
end
```

Tabla 7-23: Función de generación de la componente EventCoordinator del paquete servidor para el diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.10.9 Función de generación generateClientEventCoordinators

Esta función es la encargada de generar el componente ClientEventCoordinator, su interfaz, la relación de tipo <<inherit>> entre estas dos y agregarlo al paquete de administración de eventos en el servidor.

```
function generateClientEventCoordinators(communicationPattern,
clientComposedEvents, clientPackages)
begin
    foreach actor in communicationPattern->actors
    do
        if clientComposedEvents[actor] == true then
            var eventCoordinator = generateEventCoordinator(actor)
            var eventCoordinatorInterface =
generateEventCoordinatorInterface(actor)
            clientPackages[actor]->addCompInterfRel(eventCoordinator,
eventCoordinator Interface, inheritArrow)
        fi
    od
end
```

Tabla 7-24: Función de generación de la componente EventCoodinator para el paquete Cliente del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.11 Reglas de transformación de objetos <<entity>>

7.3.11.1 Regla de transformación entityObject2PesistentStorageComponent

Esta función es la encargada de generar el componente entityObject2PesistentStorageComponent, su interfaz, la relación de tipo <<inherit>> entre estas dos y agregarlo al paquete de administración de eventos en el servidor

En la Figura 7-1 y Figura 7-2 se puede ver el resultado de aplicar esta transformación.

```
function entityObject2PesistentStorageComponent(serverPackage, entity)
begin
    var dao = generateDAO(entity)
    var daoInterface = generateDAOInterface(entity)
    serverPackage->addCompInterfRel(dao, daoInterface, inheritArrow)

    var dataSourceConnector = generateDataSourceConnector()
    var dataSourceConnectorInterface =
generateDataSourceConnectorInterface()
    serverPackage->addCompInterfRel(dataSourceConnector,
dataSourceConnectorInterface, inheritArrow)
end
```

Tabla 7-25: Función de generación de las componentes DAO y DataSourceConnector para el diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.11.2 Regla de transformación entityObject2VolatilStorageComponent

Esta función es la encargada de generar el componente VolatilStorage, su interfaz, la relación de tipo <<inherit>> entre estas dos y agregarlo al paquete de administración de eventos en el servidor.

```
function entityObject2VolatilStorageComponent(actor, clientPackages,
entity)
begin
    var localStore = generateLocalStore(actor, entity)
    var localStoreInterface = generateLocalStoreInterface(actor, entity)
    clientPackages[actor]->addCompInterfRel(localStore,
localStoreInterface, inheritArrow)
end
```

Tabla 7-26: Función de generación de la componente LocalStore para el paquete cliente del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.12 Funciones de generación de relaciones <<use>>

7.3.12.1 Función de generación generateClientUseRelationships

La función *generateClientUseRelationships* genera las relaciones de tipo <<use>> entre los componentes y las interfaces hallados en el paquete cliente. La función consta de un ciclo principal que a su vez contiene otros dos ciclos anidados. El ciclo principal itera la lista de actores hallados en el patrón. De cada actor obtiene las componentes del modelo de arquitectura ya generadas en el paquete cliente correspondiente a cada actor. En el primer ciclo anidado se crean todas las relaciones

<<use>> entre los componentes e interfaces de un paquete cliente determinado. En el segundo ciclo anidado se crea una relacion <<use>> entre cada LocalStore y UIVisualComponentDataInterface. Utiliza funciones auxiliares como *getController* y *getAsyncRemoteRequest* para obtener componetes que ya han sido generados.

```
function generateClientUseRelationships(communicationPattern,
clientPackages)
begin
  // iteramos todos los actores del patrón de comunicación
  foreach actor in communicationPattern->actors
      // obtenemos las componentes ya generadas
      var clientController = getClientController(clientPackages[actor])
      var clientControllerInterface =
getClientControllerInterface(clientPackages[actor])
      var asyncRemoteRequest =
getAsyncRemoteRequest(clientPackages[actor])
      var asyncRemoteRequestInterface =
getAsyncRemoteRequestInterface(clientPackages[actor])
      var clientEventCoordinator =
getClientEventCoordinator(clientPackages[actor])
      var clientEventCoordinatorInterface =
getClientEventCoordinatorInterface(clientPackages[actor])
      // almacenaremos cada UIVisualComponentDataInteface para luego poder
conectarla con los LocalStore hallados en el patrón de comunicación
      var visualComponentDataInterfaces = []

      // se crean las relaciones <<use>> entre las componentes e
interfaces que tienen sólo una instancia dentro del patrón de
comunicación.
      clientPackages[actor]->addCompInterfRel(clientController,
asyncRemoteRequestInterface, useArrow)
      clientPackages[actor]->addCompInterfRel(asyncRemoteRequest,
clientControllerInterface, useArrow)

      clientPackages[actor]->addCompInterfRel(clientEventCoordinator,
clientControllerInterface, useArrow)
      // iteramos las componentes generadas del paquete cliente de cada
actor del patrón de comunicación.
      foreach object in clientPackages[actor]
      do
          if object->stereotype == 'UIVisualComponentDataInterface'
          then
              clientPackages[actor]->addCompInterfRel(clientController,
object, useArrow)
              visualComponentDataInterfaces.add(object)
          elseif object->stereotype == 'LocalStoreInterface'
          then
              clientPackages[actor]->addCompInterfRel(clientController,
object, useArrow)
          elseif object->stereotype == 'UIVisualComponent'
          then
              clientPackages[actor]->addCompInterfRel(object,
clientControllerInterface, useArrow)
              clientPackages[actor]->addCompInterfRel(object,
clientEventCoordinatorInterface, useArrow)
          elseif object->stereotype == Timer
          then
              clientPackages[actor]->addCompInterfRel(object,
clientControllerInterface, useArrow)
          elseif object->stereotype == 'TimerInterface'
          then
              clientPackages[actor]->addCompInterfRel(clientController,
object, useArrow)
          fi
      od
      // Creamos una relación <<use>> entre cada LocalStore y cada
VisualComponentDataInterface.
      foreach object in clientPackages[actor]
      do
          if object->stereotype == 'LocalStore'
              foreach visualComponentDataInterface in
visualComponentDataInterfaces
              do
```

```
                    clientPackages[actor]->addCompInterfRel(object,
visualComponentDataInterface, useArrow)
                od
            fi
        od
    od
end
```

Tabla 7-27: Generación de las relaciones <<use>> entre las componentes e interfaces del paquete cliente del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.12.2 Función de generación generateServerUseRelationships

La función *generateServerUseRelationships* genera las relaciones de tipo <<use>> entre los componentes y las interfaces hallados en el paquete servidor. Primero se obtienen todas las componentes ya generadas para el paquete servidor del patrón de comunicación. Luego se generan las relaciones <<use>> entre las componentes y las interfaces que tienen una sola instancia dentro de patrón. A continuación se iteran las componentes del paquete servidor para generar las relaciones <<use>> entre las componentes e interfaces que tienen múltiples instancias dentro de patrón. Finalmente, se crean las relaciones <<use>> entre los DAOs y la interfaces de los DTOs. Utiliza funciones auxiliares de tipo *get** para obtener componentes que ya han sido generados.

```
function generateServerUseRelationships(serverPackage)
begin
    var serverController = getServerController(serverPackage)
    var serverControllerInterface =
getServerControllerInterface(serverPackage)
    var serializador = getSerializador(serverPackage)
    var serializadorInterface = getSerializadorInterface(serverPackage)
    var conectorProtocoloRed = getConectorProtocoloRed(serverPackage)
    var conectorProtocoloRedInterface =
getConectorProtocoloRedInterface(serverPackage)
    var dataSourceConnector = getDataSourceConnector(serverPackage)
    var dataSourceConnectorInterface =
getDataSourceConnectorInterface(serverPackage)

    serverPackage->addCompInterfRel(serverController,
conectorProtocoloRedInterface, useArrow)
    serverPackage->addCompInterfRel(serverController,
serializadorInterface, useArrow)

    var DAOs = []
    var DTOInterfaces = []
    foreach object in serverPackage
    do
        if object->stereotype == 'DAOInterface'
        then
            serverPackage->addCompInterfRel(serverController, object,
useArrow)
        elseif object->stereotype == 'DAO'
        then
            serverPackage->addCompInterfRel(object,
dataSourceConnectorInterface, useArrow)
            serverPackage->addCompInterfRel(distribuidorEventos, object,
useArrow)
            DAOs.add(object)
        elseif object->stereotype == 'DTOInterface'
        then
            serverPackage->addCompInterfRel(serverController, object,
useArrow)
            DTOInterfaces.add(object)
        fi
    od
    foreach DTOInterface in DTOInterfaces
    do
        var DAOObject = getDAOforDTO(DTOInterface)
        serverPackage->addCompInterfRel(DAOObject, DTOInterface, useArrow)
    od
end
```

Tabla 7-28: Generación de las relaciones <<use>> entre las componentes e interfaces del paquete servidor del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.12.3 Función de generación generateServerEventAdministratorUseRelationships

La función *generateServerEventAdministratorUseRelationships* genera las relaciones de tipo <<use>> entre los componentes y las interfaces hallados en el paquete ServerEventAdministrator. Primero se obtienen todas las componentes ya generadas para el paquete de adminstración de eventos del patrón de comunicación. Luego se generan las relaciones <<use>> entre las componentes y las interfaces que tienen una sola instancia dentro de patrón. Utiliza funciones auxiliares de tipo *get** para obtener componentes que ya han sido generados.

```
function
generateServerEventAdministratorUseRelationships(serverEventAdministratorP
ackage)
begin
    var procesadorPedidoPersistente =
getProcesadorPedidoPersistente(serverEventAdministratorPackage)
    var procesadorPedidoPersistenteInterface =
getProcesadorPedidoPersistenteInterface(serverEventAdministratorPackage)
    var publishSubscribeAdministrator =
getPublicSubscribeAdministrator(serverEventAdministratorPackage)
    var publishSubscribeAdministratorInterface =
getPublicSubscribeAdministratorInterface(serverEventAdministratorPackage)
    var serverEventCoordinator =
getServerEventCoordinator(serverEventAdministratorPackage)
    var serverEventCoordinatorInterface =
getServerEventCoordinatorInterface(serverEventAdministratorPackage)
    var subscriptionList =
getSubscriptionList(serverEventAdministratorPackage)
    var subscriptionListInterface =
getSubscriptionListInterface(serverEventAdministratorPackage)
    var distribuidorEventos =
getDistribuidorEventos(serverEventAdministratorPackage)
    var distribuidorEventosInterface =
getDistribuidorEventosInterface(serverEventAdministratorPackage)
    var externalAppEvents =
getExternalAppEvents(serverEventAdministratorPackage)
    var externalAppEventsInterface =
getExternalAppEventsInterface(serverEventAdministratorPackage)

    serverEventAdministatorPackage-
>addCompInterfRel(procesadorPedidoPersistente,
publishSubscribeAdministratorInterface, useArrow)
    serverEventAdministatorPackage-
>addCompInterfRel(publishSubscribeAdministrator,
externalAppEventsInterface, useArrow)
    serverEventAdministatorPackage-
>addCompInterfRel(publishSubscribeAdministrator,
serverEventCoordinatorInterface, useArrow)
    serverEventAdministatorPackage-
>addCompInterfRel(publishSubscribeAdministrator,
subscriptionListInterface, useArrow)
    serverEventAdministatorPackage-
>addCompInterfRel(serverEventCoordinator, distribuidorEventosInterface,
useArrow)
    serverEventAdministatorPackage->addCompInterfRel(distribuidoEventos,
subscriptionListInterface, useArrow)
    serverEventAdministatorPackage->addCompInterfRel(distribuidoEventos,
procesadorPedidoPersistenteInterface, useArrow)
    serverEventAdministatorPackage->addCompInterfRel(distribuidoEventos,
externalAppEventsInterface, useArrow)
    serverEventAdministatorPackage->addCompInterfRel(externalAppEvents,
publishSubscribeAdministratorInterface, useArrow)
end
```

Tabla 7-29: Generación de las relaciones <<use>> entre las componentes e interfaces del paquete administrador eventos servidor del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.12.4 Función de generación generateServerAndServerEventsUseRelationships

La función *generateServerAndServerEventsUseRelationships* genera las relaciones de tipo <<use>> entre los componentes del paquete servidor y las interfaces del paquete ServerEventsAdministrator. Primero se obtienen todas las componentes ya generadas para el paquete servidor y también del paquete de administración de eventos del servidor. Luego se generan las relaciones <<use>> entre el controlador

del paquete Server y la interfaz del procesador de pedidos persistente del paquete ServerEventsAdministrator y entre la componente del procesador de pedido perisistente y el serializador. A continuación se iteran las interfaces de los DAOs de tipo EventHistoryDAOInterface y se crea la relación <<use>> entre estos y los componentes distrubuidorEventos. Utiliza funciones auxiliares de tipo *get** para obtener componentes que ya han sido generados.

```
function
generateServerAndServerEventsUseRelationships(serverAndServerEventsPackage
, serverPackage, serverEventAdministratorPackage)
begin
    var serverController = getServerController(serverPackage)
    var serializadorInterface = getSerializadorInterface(serverPackage)
    var procesadorPedidoPersistenteInterface =
getProcesadorPedidoPersistenteInterface(serverEventAdministatorPackage)
    var distribuidorEventos =
getDistribuidorEventos(serverEventAdministatorPackage)
    var procesadorPedidoPersistente =
getProcesadorPedidoPersistente(serverEventAdministatorPackage)

    serverPackage->addCompInterfRel(serverController,
procesadorPedidoPersistenteInterface, useArrow)
    serverPackage->addCompInterfRel(procesadorPedidoPersistente,
serializadorInterface, useArrow)

    foreach object in serverPackage
    do
        if object->stereotype == 'DAOInterface'
        then
            if object->name = 'EventHistoryDAOInterface'
            then
                serverPackage->addCompInterfRel(distribuidorEventos,
object, useArrow)
            fi
        fi
    od
end
```

Tabla 7-30: Generación de las relaciones <<use>> entre las componentes e interfaces del paquete servidor y administrador eventos servidor del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.13 Funciones de generación de interfaces de componentes

7.3.13.1 Función de generación generateUIVisualComponentInteractionInterface

La función *generateUIVisualComponentInteractionInterface* genera la interfaz de eventos de un UIVisualComponent.

La función *generateUIVisualComponentDataInterface* genera la interfaz de datos de un UIVisualComponent. Para esto, itera las operaciones halladas en el objeto <<boundary>> de tipo UI, que son operaciones de estructuras de datos como pilas, colas, árboles, etc. y las mapea a operaciones sobre los UIVisualComponent relacionadas con la manera en que se visualiza el mismo.

```
function generateUIVisualComponentInteractionInterface(uiBoudaryObject)
begin
    var uiVisualComponentInteractionInterface =
createEmptyInterface(uiBoundaryObject)
    foreach event in uiBoundaryObject->messages->events
    do
        uiVisualComponentInteractionInterface->addMethod('Emit_' + event-
>name)
    od
    return uiVisualComponentInteractionInterface
end

function generateUIVisualComponentDataInterface(uiBoudaryObject)
begin
    var uiVisualComponentDataInterface =
createEmptyInterface(uiBoundaryObject)

    foreach message in uiBoundaryObject->messages
    do
        uiVisualComponentDataInterface-
>addMethod(getEntityMessageName(message))
    od
    return uiVisualComponentDataInterface
end
```

Tabla 7-31: Función de generación de interfaz de componente UIVisualComponent del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.13.2 Función de generación generateClientControllerInterface

Dado un actor hallado en el patrón de comunicación, genera la interfaz de un Controller del lado cliente.

```
function generateClientControllerInterface(actor)
begin
    var clientControllerInterface = createEmptyInterface(actor)
    clientControllerInterface->addMethod(bind)
    clientControllerInterface->addMethod(enableEvent)
    clientControllerInterface->addMethod(disableEvent)
    foreach uiBoundaryObject in uiBoundaryObjects
    do
        foreach event in uiBoundaryObject->messages->events
        do
            clientControllerInterface->addMethod(doOn(event))
            clientControllerInterface->addMethod(doOnDone(event))
        od
    od
    return clientControllerInterface
end
```

Tabla 7-32: Función de generación de interfaz de componente ClientController del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.13.3 Función de generación generateServerControllerInterface

Genera la interfaz del ServerController.

```
function generateServerControllerInterface()
begin
    var serverControllerInterface = createEmptyInterface()
    clientControllerInterface->addMethod(atenderPedido)
    clientControllerInterface->addMethod(atenderPedidoPersistente)
    return clientControllerInterface
end
```

Tabla 7-33: Función de generación de interfaz de componente ServerController del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.13.4 Función de generación generateDAOInterface

Dado un objeto <<entity>>, genera la interfaz de cada DAO.

```
function generateDAOInterface(entityObject)
begin
     var daoInterface = createEmptyInterface(entityObject)
     foreach message in entityObject->messages
     do
         daoInterface->addMethod(message)
     od
     return daoInterface
end
```

Tabla 7-34: Función de generación de interfaz de componente DAO del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.3.13.5 Función de generación generateLocalStoreInterface

Dado un actor hallado en el patrón de comunicación y un objeto <<entity>>, genera la interfaz de los LocalStore.

```
function generateLocalStoreInterface(actor, entityObject)
begin
    var localStoreInterface = createEmptyInterface(entityObject)
    foreach message in entityObject->messages
    do
         localStoreInterface->addMethod(message)
    od
    return localStoreInterface
end
```

Tabla 7-35: Función de generación de interfaz de componente LocalStore del diagrama de componentes del modelo de arquitectura independiente de plataforma.

7.4 Transformación de diagramas de comunicación de análisis a diagramas de comunicación de arquitectura

Debido a que la transformación a realizar tiene un grado elevado de complejidad, decidimos abstraerla utilizando el paradigma de objetos, ya que provee la flexibilidad necesaria para implementar cualquier orden o política para mapear los elementos de análisis en orden de numeración. De este modo, sabemos de antemano que el paradigma no será una limitación.
En la siguiente sección presentamos una clase que genera el diagrama de comunicaciones de la arquitectura de referencia a partir del diagrama de comunicaciones del modelo de análisis. Para esto, itera los mensajes del modelo de análisis en orden.

Es importante destacar que asumimos que el modelo de análisis recibido como parámetro puede contener sólo un patrón de comunicación.

7.4.1 Descripción del Algoritmo

En general la transformación considera:

- Identificar el patrón de comunicación.
- Iterar los mensajes del diagrama de comunicación en orden de numeración.

Para mapear un mensaje de análisis en general se siguen los siguientes pasos:

- Identificar el tipo de mensaje.
- Mapear el objeto source del mensaje a una instancia de la componente respectiva y agregarla al diagrama de comunicaciones de la arquitectura de referencia, siempre y cuando no haya sido creado aún.
- Crear la instancia del objeto destino o *destination* del mensaje y agregarla al diagrama de comunicaciones de la arquitectura de referencia, siempre y cuando no haya sido creada aún.
- Agregar el conector entre las instancias creadas.
- Agregar el mensaje al conector.
- Mapear las condiciones e iteraciones de un mensaje del modelo de análisis al mensaje del modelo de arquitectura correspondiente.
- Agregar la numeración al mensaje.

Como en el modelo de análisis no tenemos información sobre qué ocurre del lado del cliente y qué del lado del servidor, pero al momento de mapearlo a la arquitectura, debemos saber en que momento debemos generar los mensajes que vuelven del servidor al cliente, utilizamos una variable *inReferenceArchitecurePackage* que se actualiza de acuerdo a si las instancias generadas son en el cliente, en el servidor o en administrador de eventos del servidor. Puede tomar 3 valores: client, server o serverEventAdministrator. De esta manera, basados en ese valor podemos saber si tenemos que generar los mensajes que representan la transición de un paquete a otro.

Para mejorar la legibilidad del algoritmo, definimos una clase que contiene los métodos de mapeo que pueden reutilizarse en más de un patrón y una clase para cada patrón de comunicación, que contiene los métodos específicos para realizar las transformaciones de cada mensaje para ese patrón en particular. En estas clases, definimos un método de transformación para cada tipo de mensaje.
Ejemplos de métodos que puede reutilizarse:

- EmitMessage2ModArqEmitMessage: mapea mensajes de tipo Emit_*.
- ControlToBoundaryUIMessage2ModArqMessages: mapea mensajes de control a boundary de tipo UI.

Ejemplos de métodos que son específicos de un patrón:

- GetSaveMessage2ModArqMessages
- DoOnMessage2ModArqMessages

7.4.1.1 Convenciones de nombres para transformaciones y reglas de transformación

- Los nombres de las clases que realizan las transformaciones respetan el siguiente patrón:

  <nombre-del-patrón>CommunicationPattern2CommDiagram

- Los nombres de las reglas de transformacipon siguen los siguientes patrones:

  <tipo-de-mensaje-de-analisis>2ModArq<tipo-de-mensaje-de-arquitectura>-Message si mapea a un solo mensaje.

  <tipo-de-mensaje-de-analisis>2ModArqMessages si mapea a más de un mensaje.

7.4.2 Definición de clases que contienen funciones 'helper' para la transformación

7.4.2.1 La clase AnalysisModel

La clase AnalysisModel representa el modelo de análisis que se desea transformar. Dado que la transformación se realiza transformando un mensaje del diagrama de comunicación del modelo de análisis a la vez, decidimos iterar los mensajes según el orden de la numeración. Este diseño nos permite:

1. Asegurarnos que la transformación abarca a todos los mensajes.
2. Durante la transformación del mensaje podemos ir generando los componentes que se comunican mediante ese mensaje.
3. Evitar generar componentes que ya han sido generados.

Debido a que los números de los mensajes pueden tener varios niveles de anidamiento o concurrencia, dado un mensaje *n*, necesitamos un mecanismo para poder buscar cual es el número del siguiente mensaje *n+1*. Para esto definimos el método *getNextMessage* que devuelve el siguiente mensaje, tomando en cuenta todas los posibles anidamientos y/o concurrencia. Dado un número de mensaje *n*, la función busca dentro del diagrama de comunicación del modelo de análisis el siguiente número de mensaje y devuelve el primer mensaje que encuentre, según el siguiente orden de prioridad:

1. Busca el siguiente mensaje concurrente. Por ejemplo, si el mensaje actual es 1.1.a, la función intentará encontrar 1.1.b
2. Busca el siguiente mensaje anidado. Por ejemplo, si el mensaje actual es 1.1, la función intentará encontrar 1.1.1.
3. Busca el siguiente mensaje no anidado. Por ejemplo, si el mensaje actual es 1.1, la función intentará encontrar 1.2.
4. Busca el siguiente mensaje desanidado. Por ejemplo, si el mensaje actual es

1.2.1 la función intentará encontrar 1.3.

Este orden de prioridad responde al orden natural inherente de los mensajes en los diagramas de comunicación UML.

```
class AnalysisModel
begin
    /* Este método devuelve, dado un mensaje 1.2.3, el siguiente mensaje
según el     siguiente orden, si tal mensaje existe:
     * - 1.2.3.1
     * - 1.2.3.2
     * - 1.2.4
     * - 1.3
     * - 2
     */
    public function getNextMessage(analysisModel, currentMessageNumber)
    begin
        if analysisModel->messageExists(currentMessageNumber + '.1') then
            return analysisModel->getMessage(currentMessageNumber + '.1')
        else
            // convierte '1.2.3' a un array [1,2,3]
            var currentMessageNumberAsArray = currentMessageNumber-
>toArray()
            var i = currentMessageNumberAsArray.length() - 1
            while i >= 0
            do
                // verifica si el último carácter es un número 1.1.2
                if isNumber(currentMessageNumberAsArray[i]) then
                    currentMessageNumberAsArray[i] =
currentMessageNumberAsArray[i] + 1
                // verifica si el último carácter es una letra 1.1.2.a
                elseif isCharacter(currentMessageNumberAsArray[i]) then
                    // getNextCharacter devuelve el siguiente caracter en
el orden alfabetico
                    currentMessageNumberAsArray[i] =
getNextCharacter(currentMessageNumberAsArray[i])
                fi

                // convierte un array [1,2,3] a '1.2.3'
                currentMessageNumber = currentMessageNumberAsArray-
>toString()
                if analysisModel->messageExists(currentMessageNumber) then
                    return analysisModel->getMessage(currentMessageNumber)
                fi
                // si no encontré ningún mensaje en este nivel, vuelvo
para atrás un nivel.
                i = i - 1;
            od
            return false
        fi
    end
end
```

Tabla 7-36: La clase AnalysisModel representa al diagrama de comunicaciones del modelo de análisis.

7.4.2.2 La clase ArchitectureModel

Representa la arquitectura de referencia a generar y oficia de contenedor para todos los paquetes, objetos y mensajes generados durante la transformación.

```
class ArchitectureModel
begin
    var Array instances
    var Array connectors
    var Array messages

    public function addInstance(refArchObjectInstance)
    begin
        this->instances->push(refArchObjectInstance)
    end

    public function addConnector(refArchSourceInstance,
referenceArchitecturaDestinationInstance)
    begin
        this->connectors->push(new Array(refArchSourceInstance,
referenceArchitecturaDestinationInstance))
    end

    public function addMessage(refArchMessage, refArchSourceInstance,
referenceArchitecturaDestinationInstance)
    begin
        this->messages->push(new Array(refArchMessage,
refArchSourceInstance, referenceArchitecturaDestinationInstance))
    end

    public function createInstance(type)
    begin
        return new ArquitectureModelObjectInstance(type)
    end

    public function exists(instanceName)
    begin
        return this->instances->exists(instanceName)
    end
end
```

Tabla 7-37: La clase ArchitectureModel representa el digrama de comunicaciones del modelo de arquitectura independiente de plataforma.

7.4.2.3 La clase StorageType

Esta clase almacena información que el algoritmo necesita para saber si un objeto <<entity>> del modelo de análisis representa almacenamiento de información volátil, persistente o ambos. Esto es necesario debido a que no contamos con tal información en el modelo de análisis, con lo cual debe ser provisto como un input separado del algoritmo.

```
class StorageTypes
begin
    var Array storageTypeByEntity

    public constructor (storageTypeByEntity)
    begin
        this->storageTypeByEntity = storageTypeByEntity
    end
    // devuelve 'volatil', 'persistent' o 'both'
    public function getStorageType(entityObject)
    begin
        return this->storageTypeByEntity[entityObject->name];
    end
end
```

Tabla 7-38: La clase StorageTypes permite sabe, para cada objeto de entidad del modelo de análisis, qué tipo componentes de almacenamiento (persistente o volátil) se deben generar en el diagrama de comunicaciones del modelo de arquitectura.

7.4.2.4 La clase Message

Cada mensaje está representado por un objeto que tiene un nombre, un número, una fuente o *source* y un destinatario o *destination*. Además, definimos un método *getNestedMessage* que nos permite obtener por nombre un mensaje anidado.

```
class Message
begin
    var sources
    var destination
    var name
    var number
    var nestedMessages

    public function getNumber()
    begin
        return number
    end

    public function getNestedMessage(messageName)
    begin
       return nestedMessages[messageName]
    end

    public function setNumber(messageNumber)
    begin
        this->number = messageNumber
    end
end
```

Tabla 7-39: La clase Message representa un mensaje del diagrama de comunicaciones del modelo de análisis.

7.4.2.5 La clase ObjectMap

Esta clase es la encargada de almacenar qué objetos de la arquitectura de referencia fueron generados por cada objeto del modelo de análisis. Permite obtener un objeto de la arquitectura de referencia a partir de un objeto del modelo de análisis más el nombre del objeto de la arquitectura de referencia deseado. Para esto asumimos la existencia de una estructura de datos comúnmente utilizada llamada ArrayMap, que representa un Array indexado por Strings, en este caso, indexado por el nombre del objeto del modelo de análisis. Como necesitamos un Array de 3 dimensiones, utilizamos ArrayMaps anidados, de manera que objectmMap1.get(<nombre-del-objeto-del-modelo-de-analisis>) devuelve otro ArrayMap objectMap2 tal que objectMap2.get(<nombre-del-objeto-de-la-arq-de-ref>) devuelve el objeto de la arquitectura de referencia deseado.

```
class ObjectMap
begin
    var ArrayMap objectMap

    // devuelve un objeto de la arquitectura de referencia. El primer
parámetro indica el objeto del modelo de análisis a partir del cuál se
generó el objeto de la arquitectura de referencia. El segundo indica el
nombre del objeto que se quiere obtener.
    public function getInstance(analysisModelObject, archRefObjectName)
    begin
        // como el ArrayMap es multidimensional, primero utilizamos el
nombre del objeto del modelo de análisis como clave. El resultado es otro
ArrayMap indexado por nombre de objeto de la arquitectura de referencia.
(Ej.: this->objectMap[analysisModelObjName][archRefObjectName])
        var analysisModelMap = this->objectMap->get(analysisModelObject-
>name)
        return analysisModelMap->get(archRefObjectName)
    end

    public function push(analysisModelObject, archRefObject)
    begin
        // primero verificamos si ya existe el ArrayMap para el objeto del
modelo de análisis
        var analysisModelMap = objectMap->get(analysisModelObject->name)
        if (analysisModelMap == null)
        begin
            // si no existe lo creamos
            analysisModelMap = new ArrayMap()
        end
        // guardamos el array de dos dimensiones (Ej.: this-
>objectMap[analysisModelObjName][archRefObjectName])
        analysisModelMap->push(archRefObject->name, archRefObject)
        this->objectMap->push(analysisModelObject->name, analysisModelMap)
    end
end
```

Tabla 7-40: La clase ObjectMap permite almacenar por cada objeto del diagrama de comunicaciones del modelo de análisis, cuáles son las componentes generadas en el diagrama de comunicaciones del modelo de arquitectura.

7.4.2.6 La clase VisitedInstances

Esta clase es la encargada de almacenar qué objetos del modelo de análisis ya han sido visitados, de manera de no volver a generar los objetos de la arquitectura de referencia. Internamente, almacena en un Array los nombres de los objetos del modelo de análisis ya visitados y permite consultar a través del método *exists* si un objeto está en el Array o no. Asume la existencia de una estructura de datos de nombre Array.

```
class VisitedInstances
begin
    var Array visitedInstances

    // verifica si el objeto de análisis ya fue visitado o no
    public function exists(analysisModelObject)
    begin
        if this->visitedInstances->exists(analysisModelObject->name) then
            return true
        else
            return false
        end
    end

    public function push(analysisModelObject)
    begin
        this->visitedInstances->push(analysisModelObject)
    end
end
```

Tabla 7-41: La clase VisitedInstances permite ir almacenando, a medida que progresa la transformación, cuáles son los objetos del diagrama de comunicaciones del modelo de análisis que ya han sido transformados.

7.4.2.7 Numeración de los mensajes generados en el modelo de arquitectura

Consideramos que dos mensajes consecutivos son anidados si coinciden con alguna de las siguientes secuencias de mensajes:

- uiVisualComponent → clientController → asyncRemoteRequest
- clientController → asyncRemoteRequest → webserver
- serverController → dao → dataSourceConnector
- distribuidorEventos → subscriptionList
- distribuidorEventos → notificationsHistoryDAO
- distribuidorEventos → procesadorPedidoPersistente

Para controlar la generación de mensajes anidados utilizamos dos variables y un método. Las variables refArchCurrentMessage y refArchPreviousMessage almacenan el último mensaje generado para la arquitectura de referencia y el anterior (si lo hay). La función getRefArchNextMessageNumber toma como argumentos los valores de estas dos variables y a partir del número del mensaje previo genera el número para el último mensaje generado. También, a partir de los estereotipos del mensaje previo y el actual, detecta si el número del nuevo mensaje debe ser anidado o no.

```
class CommunicationPattern2CommunicationDiagram
begin
    [...]
    public function setRefArchMessageNumber()
    begin
        var nextNumber = 0
        // la siguiente tupla constituye un candidato a mensaje anidado
        var nestedCandidateTuple = [getStereoType(this-
>refArchPreviousMessage->destination), getStereotype(this-
>refArchCurrentMessage->source), getStereotype(this-
>refArchCurrentMessage->destination)]
        if not this->refArchPreviousMessage then
            this->message->setNumber(1)
        else
            if nestedCandidateTuple in {[uiVisualComponent,
clientController, asyncRemoteRequest]}
              or nestedCandidateTuple in {[clientController,
asyncRemoteRequest, webserver]}
              or nestedCandidateTuple in {[serverController, dao,
dataSourceConnector]} then
                nextNumber = getNextUnnestedNumber(nextNumber)
            elseif nestedCandidateTuple in
{[asyncRemoteRequest,clientController , uiVisualComponent]}
              or nestedCandidateTuple in {[webserver, asyncRemoteRequest,
clientController]}
              or nestedCandidateTuple in {[dataSourceConnector, dao,
serverController]} then
                nextNumber = getNextUnnestedNumber(this-
>refArchPreviousMessage->getNumber())
            else
                nextNumber = getNextNumber(this->refArchPreviousMessage-
>getNumber())
            fi
            this->refArchCurrentMessage->setNumber(nextNumber)
        fi
    end
    [...]
end
```

Tabla 7-42: El método setRefArchMessageNumber de la clase CommunicationPattern2CommunicationDiagram se encarga de setear el número de mensaje al último mensaje generado para el diagrama de comunicaciones del modelo de arquitectura.

7.4.2.8 Generación de secuencia de mensajes de retorno del servidor al cliente en el modelo de arquitectura

La transformación desde el modelo de análisis al modelo de arquitectura independiente de plataforma presenta una particularidad introducida por los mensajes de objeto de *<<control>>* a objeto *<<boundary>>*. Estos mensajes mapean a varios mensajes en el modelo de arquitectura, algunos de los cuales están en el paquete cliente y otros en el paquete servidor. Por lo tanto, en el modelo de arquitectura tenemos una secuencia de mensajes que refleja el retorno desde el servidor al cliente. Sin embargo, el modelo de análisis no provee información sobre qué es lo que ocurre en el cliente y qué en el servidor. Esto quiere decir que la secuencia de mensajes de retorno del servidor al cliente en el modelo de arquitectura no resulta del mapeo de un mensaje del modelo de análisis, sino que es fija y siempre igual. Para generar esta secuencia fija de mensajes, definimos la función *returnFromServer*. Por otro lado, debemos también definir un mecanismo para detectar en qué momento ocurre el retorno desde el servidor al cliente, para generar la secuencia de mensajes antes de proceder a generar el siguiente mensaje. Para esto definimos la variable de instancia

currentPackage la cual se va actualizando con los valores *'server'* o *'client'* y refleja en qué paquete se generó la última componente del modelo de referencia. Como el mapeo de los objetos *<<entity>>* también puede generar componentes en el paquete cliente (almacenamiento volátil) y en el servidor (almacenamiento persistente), en este caso deberemos utilizar la información provista por el modelo de marcas para actualizar la variable *currentPackage*. De este modo, cuando durante la transformación encontramos un mensaje de objeto de control a objeto boundary de tipo UI, llamamos a la función *returnFromServer* la cual genera la secuencia de mensajes de retorno del servidor, sólo en el caso en que el paquete en el que se generó la última componente del modelo de arquitectura sea el paquete servidor.

```
class CommunicationPattern2CommunicationDiagram
begin
    // Este método genera los mensajes de vuelta del servidor cuando es
necesario.
    public function returnFromServer()
    begin
        // Si estamos en el servidor generamos la secuencia de mensajes de
vuelta al cliente
        if this->currentPackage == 'server' then
            // Primero obtenemos todos los objetos ya generados que están
implicados en la secuencia de mensajes. Asumimos que ya existen dado que
si no no estaríamos volviendo del servidor.
            var serverControllerInstance = this->objectMap-
>getInstance(this->message->source, 'ServerController')
            var browser = this->objectMap->getInstance(this->message-
>source, 'browser')
            var asyncRemoteRequestInstance = this->objectMap-
>getInstance(this->message->source, 'AsyncRemoteRequest')
            var webserver = this->objectMap->getInstance(message->source,
'webserver')
            var clientControllerInstance = this->objectMap-
>getInstance(this->message->source, 'ClientController')

            // Cada vez que agregamos un mensaje lo guardamos en la
variable que almacena el mensaje que se está generando:
refArchCurrentMessage. Luego lo necesitaremos para calcular el número de
mensaje.
            this->refArchCurrentMessage = this->archModel-
>addMessage('httpResponse', serverControllerInstance, browser)
            this->setRefArchMessageNumber()
            // Luego de setear el número de mensaje, guardamos el mensaje
en la variable que almacena el último mensaje generado. Este será
utilizado por  setRefArchMessageNumber para generar el número del próximo
mensaje.
            this->refArchPreviousMessage = this->refArchCurrentMessage

            this->refArchCurrentMessage = this->archModel-
>addMessage('httpResponse', browser, asyncRemoteRequestInstance)
            this->setRefArchMessageNumber()
            this->refArchPreviousMessage =  this->refArchCurrentMessage

            this->refArchCurrentMessage = this->archModel-
>addMessage(this->message->name + '_Done', asyncRemoteRequestInstance,
clientControllerInstance)
            this->setRefArchMessageNumber()
            this->refArchPreviousMessage =  this->refArchCurrentMessage
        fi
    end
end
```

Tabla 7-43: El método returnFromServer de la clase CommunicationPattern2CommDiagram es el encargado de detectar las condiciones necesarias bajo las cuáles se debe generar la secuencia de mensajes de vuelta del servidor en el diagrama de comunicaciones del modelo de arquitectura.

7.4.3 Definición de la transformación contemplando los diferentes patrones de comunicación

7.4.3.1 Punto de entrada de la transformación: CommunicationPattern2CommunicationDiagramMain

CommunicationPattern2CommunicationDiagramMain es la clase principal, donde comienza la transformación. Identifica el patrón de comunicación a partir del modelo de análisis e inicializa la transformación. Una vez que sabe cuál es el patrón, delega el procesamiento del mapeo al CommunicationPattern2CommunicationDiagram apropiado.

```
class CommunicationPattern2CommunicationDiagramMain
begin
    var AnalysisModel analysisModel
    var StorageTypes storageTypes
    var Boolean serverHasComposedEvents
    var Boolean clientHasComposedEvents
    var CommunicationPattern2CommunicationDiagram
CommunicationPattern2CommunicationDiagram

    public function constructor(analysisModel, storageTypes,
serverComposedEvents, clientComposedEvents)
    begin
        this->analysisModel = analysisModel
        this->storageTypes = storageTypes
        this->serverComposedEvents = serverComposedEvents
        this->clientComposedEvents = clientComponsedEvents
    end

    public function parseCommunicationPattern(analysisModel)
    begin
        // asumimos que esta función identifica el patrón de comunicación
subyacente del modelo de análisis
        return getCommunicationPattern(analysisModel)
    end

    public function mapCommunicationPatternMessages()
    begin
        var communicationPattern = this->parseCommunicationPattern(this-
>analysisModel)

        // De acuerdo al patrón iterado, llamamos a la función de mapeo
        // correspondiente.
        case communicationPattern->type of 'Client-Server'
        do
            var ClientServerCommPattern2CommDiagram = new
ClientServerCommPattern2CommDiagram('client')
            ClientServerCommPattern2CommDiagram-
>transformMessages(communicationPattern, this)
        od
        case communicationPattern->type of 'Client-Server-Clients'
        do
            var clientServerClientsMapper = new
ClientServerClientsCommPattern2CommDiagram('client')
            clientServerClientsMapper-
>transformMessages(communicationPattern, this)
        od
        case communicationPattern->type of 'Client-Server-ExternalApps'
        do
            var clientServerExternalAppsMapper = new
ClientServerExternalAppsCommPattern2CommDiagram('server')
            clientServerExternalAppsMapper-
>transformMessages(communicationPattern, this)
        od
        case communicationPattern->type of 'ExternalApp-Server-Clients'
        do
            var externalAppServerClientsExternalAppsMapper = new
ExternalAppServerClientsCommPattern2CommDiagram('server')
externalAppsServerClientsMapper->transformMessages(communicationPattern,
this)
        od
    end
end
```

Tabla 7-44: La clase CommunicationPattern2CommunicationDiagramMain funciona como punto de entrada de la transformación.

7.4.3.2 Identificación del patrón de comunicación del modelo de análisis a transformar

La clase CommunicationPattern2CommunicationDiagram agrupa los métodos que serán utilizados para el mapeo de mensajes en todos los patrones de comunicación. Para los métodos específicos de cada patrón, crearemos nuevas clases que heredarán de esta.

```
class CommunicationPattern2CommunicationDiagram
begin
    var Message message
    var Integer currentMessageNumber
    var Boolean refArchPreviousMessage
    var Boolean refArchCurrentMessage
    var ArchitectureModel archModel
    var VisitedInstances visitedInstances
    var ObjectMap objectMap
    var Enum('client', 'server', 'serverEventsAdmin') currentPackage

    public function constructor(initialPackage)
    begin
        this->message = communicationPattern->getMessage(1)
        this->currentMessageNumber = 1
        this->refArchPreviousMessage = false
        this->refArchCurrentMessage = false
        // Creamos un contenedor para las instancias de objetos generadas
y los conectores.
        this->archModel = new ArchitectureModel()
        // Almacena todos los objetos ya visitados del modelo de análisis
        this->visitedInstances = new VisitedInstances
        // Almacena las instancias de objetos de la arquitectura de
referencia para cada objeto del modelo de análisis
        this->objectMap = new ObjectMap()

        // Este patrón inicia en el cliente, currentPackage indica si
estamos en el cliente, en el servidor o en el administrador de eventos del
servidor
        this->currentPackage = initialPackage
    end

    abstract public function transformMessages(communicationPattern,
CommunicationPattern2CommunicationDiagramMain)
    begin
        // definir en los descendientes el mapeo para cada patrón
    end

    abstract public function DoOnMessage2ModArqMessages()
    begin
        // definir en los descendientes el mapeo para cada patrón
    end

    abstract public function
GetSaveMessage2ModArqMessages(CommunicationPattern2CommunicationDiagramMai
n)
    begin
        // definir en los descendientes el mapeo para cada patrón
    end

    abstract public function ControlToBoundaryUIMessage2ModArqMessages()
    begin
        // definir en los descendientes el mapeo para cada patrón
    end
```

Tabla 7-45: La clase CommunicationPattern2CommunicationDiagram funciona como padre para las clases que realizan la transformación específica para cada patrón de comunicación. En esta clase encontraremos la implementación de los métodos que son comunes a todos los patrones.

7.4.3.3 La regla de transformación EmitMessage2ModArqEmitMessage

EmitMessage2ModArqEmitMessage es la encargada del mapeo de los mensajes de tipo *Emit_ **, que comunican a los actores con los objetos boundary de tipo UI y representan los eventos generados por el usuario. Una primera idea que surge para realizar esta transformación es mapear los objetos <<boundary>> de tipo UI del modelo de análisis uno a uno con componentes UIVisualComponent del modelo de arquitectura, como se muestra en la Figura 7-9. En el ejemplo se ilustra como sería esta transformación para el ejemplo de un usuario que selecciona un E-mail de una lista de E-mails.

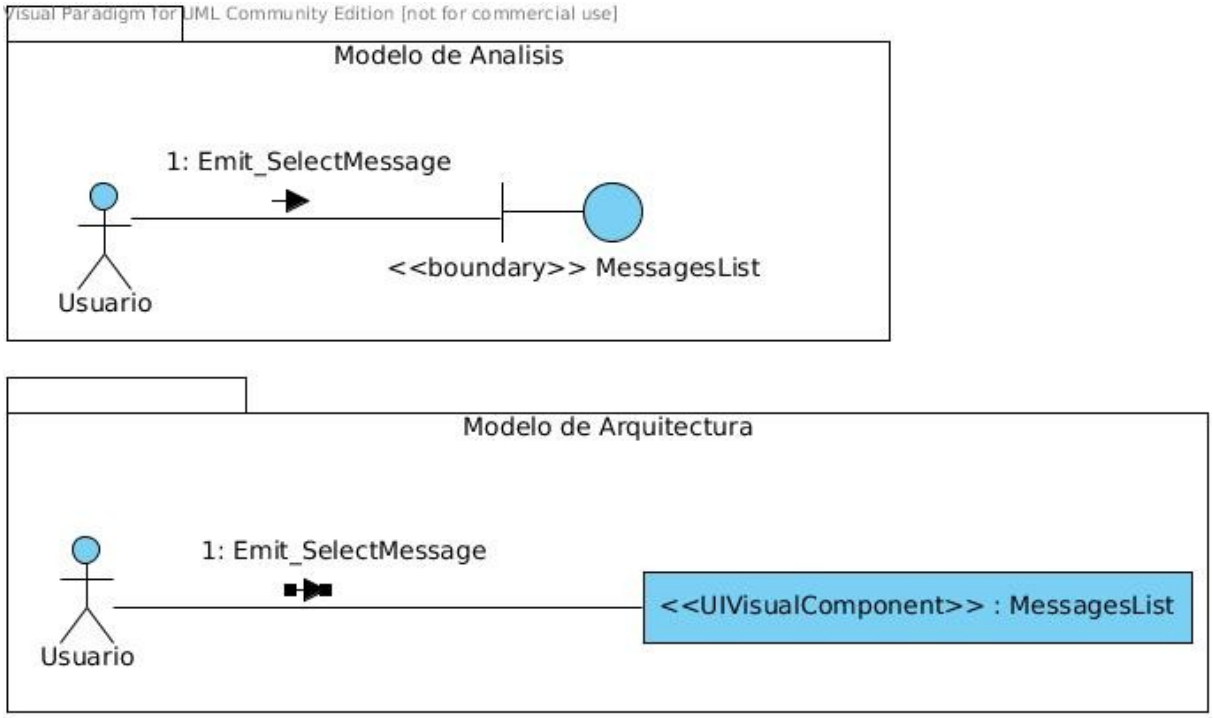

Figura 7-9: Resultado del mapeo de un mensaje Emit_* con generación de un solo destinatario en el modelo de arquitectura, descrito a través de un diagrama de comunicaciones UML

Sin embargo, en ocasiones, las interfaces con las que interactúa el usuario, representadas a través de objetos <<boundary>> de tipo UI en el modelo de análisis, son complejas. Y si bien no es necesario dar los detalles de como está conformada tal interfaz en el modelo de análisis, sí lo es en el modelo de arquitectura, ya que es a partir de este que generaremos las instancias para las arquitecturas específicas de cada tecnología, por lo cuál necesitaremos ese nivel de detalle. Cuando una interfaz está compuesta por más de un control visual, debemos descomponer el objeto <<boundary>> de tipo UI en varias componentes UIVisualComponente en el modelo de arquitectura. Para ilustrar esto, pensemos en una ventana de chat simple. La ventana está compuesta por un área de texto donde aparecen los mensajes que recibimos, otra área de texto para escribir mensajes y un botón para enviar los mensajes. Todo esto puede representarse con un solo objeto <<boundary>> en el modelo de análisis, pero debe descomponerse en cada uno de los controles visuales mencionados durante la transformación. La Figura 7-10 ilustra el mapeo de un objeto

<<boundary>> de tipo UI a más de un UIVisualComponent.

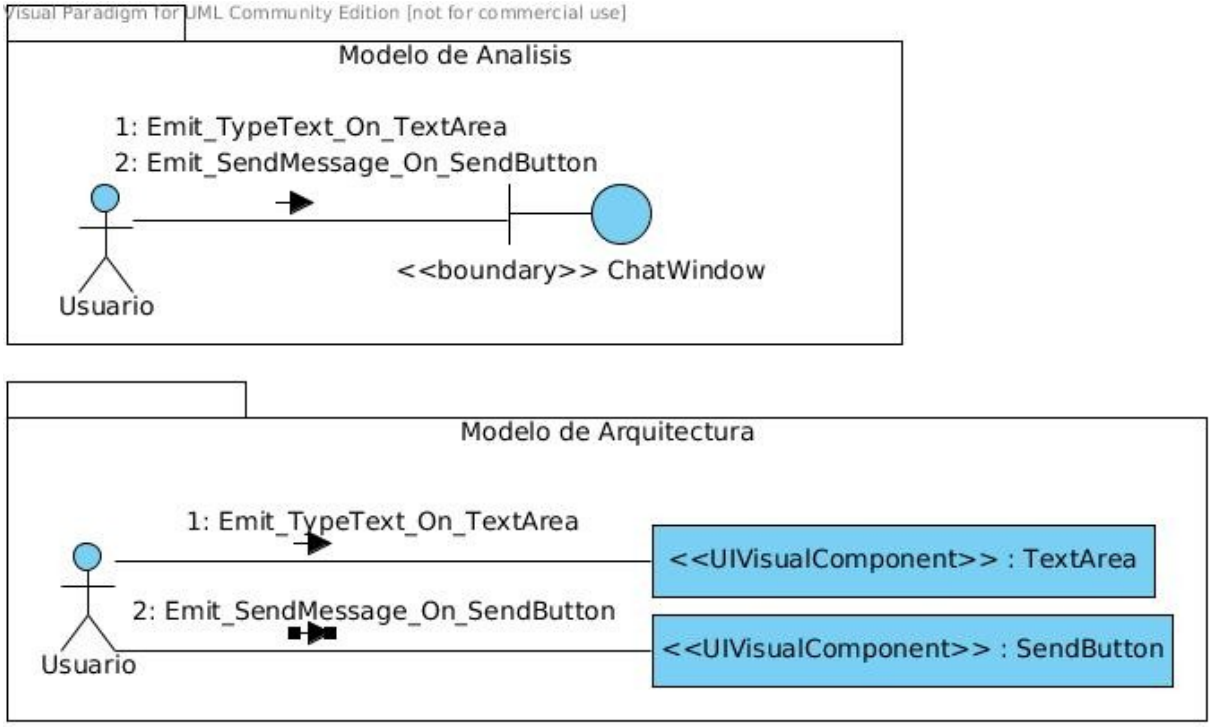

Figura 7-10: Resultado del mapeo de un mensaje Emit_* con generación de múltiples destinatarios en el modelo de arquitectura, descrito a través de un diagrama de comunicaciones UML

Surge entonces un problema que debemos resolver: ¿Como sabemos si un objeto <<boundary>> de tipo UI mapea a uno o más UIVisualComponents? En el caso en que el destinatario mapee a varias componentes, los mensajes *Emit_ *** contienen el nombre de la componente destinatario a generar en el modelo de arquitectura. En este caso, podemos extraer el nombre de la componente a generar parseando el nombre del mensaje. En el caso en que el destinatario mapee a una sola componente, lo mensajes Emit_* NO contienen el nombre del componente destinatario. En este caso, el nombre de la componente a generar lo tomamos del destinatario del mensaje en el modelo de análisis.

Para detectar el tipo de mensaje *Emit_ ** definimos la función *hasTargetInName.* Para los casos en los que sea necesario extraer el nombre de la componente a generar en el modelo de arquitectura, definimos la función *getTargetFromName*, que realiza el parseo y devuelve el nombre que debemos asignarle.

Por lo tanto las reglas en general para el mapeo uno a uno tienen la forma:

[Actor] A Emit_<event> <<boundary>> N

se mapea a

[Actor] A Emit_<event> [UIVisualComponent] N

Y para el mapeo a múltiples UIVisualComponent tiene la forma:

[Actor] A Emit_<event>_On_M <<boundary>> N

se mapea a

[Actor] A Emit_<event>_On_M [UIVisualComponent] M

```
class CommunicationPattern2CommunicationDiagram
begin
    [...]
    public function EmitMessage2ModArqEmitMessage()
    begin
        // sabemos que el source en este caso es un Actor, ya que son los
únicos que pueden generar eventos de tipo Emit_*
        if not this->visitedInstances->exists(this->message->source) then
            var actorInstance = this->archModel->createInstance('Actor')
            this->archModel->addInstance(actorInstance)
            this->visitedInstances->push(this->message->source)
        fi
        // Verificamos que todavía no se haya visitado este objeto del
modelo de análisis
        if not this->visitedInstances->exists(this->message->destination)
then
            // Verificamos si el nombre del mensaje Emit contiene el
nombre del objeto de destino o no
            if hasTargetInName(this->message) then
                // Si el nombre de la componente a generar está dentro del
mensaje Emit_, lo extraemos
                var targetName = getTargetFromName(this->message->name)
                var uiVisualComponentInstance = this->archModel-
>createInstance('UIVisualComponent')
                uiVisualComponentInstance->setName(targetName)
                this->objectMap->push(this->message->destination,
uiVisualComponentInstance);
            else
                // Si el nombre de la componente a generar NO está dentro
del mensaje Emit_, utilizamos el mismo nombre que tiene el objeto del
modelo de análisis que se está mapeando
                var uiVisualComponentInstance = this->modelArchitecture-
>createInstance('UIVisualComponent')
                uiVisualComponentInstance->setName(this->message-
>destination->name)
                this->objectMap->push(this->message->destination,
uiVisualComponentInstance);
            fi
            this->archModel->addInstance(uiVisualComponentInstance)
            this->visitedInstances->push(this->message->destination)
        fi
        // Agregamos el conector y el mensaje a la arquitectura de
referencia.
        this->archModel->addConnector(actorInstance,
uiVisualComponentInstance)
        refArchCurrentMessage = this->archModel->addMessage(actorInstance,
uiVisualComponentInstance, this->message->name)
        this->setRefArchMessageNumber()
        refArchitecturePreviousMessage = refArchCurrentMessage
    end
    [...]
end
```

Tabla 7-46: El método EmitMessage2ModArqEmitMessage realiza la transformación de mensajes entre actores y objetos de frontera del diagrama de comunicaciones del modelo de análisis a mensajes entre actores y componentes UIVisualComponent del diagrama de comunicaciones del modelo de arquitectura.

7.4.3.4 Transformación de modelos de análisis usando el patrón Client-Server a arquitectura

La clase ClientServerCommPattern2CommDiagram extiende a CommunicationPattern2CommunicationDiagram y tiene como responsabilidad el

mapeo de los mensajes del patrón Client-Server. Para esto, define los métodos requeridos específicamente para este patrón.

```
class ClientServerCommPattern2CommDiagram extends
CommunicationPattern2CommunicationDiagram
begin
    public function transformMessages(communicationPattern,
CommunicationPattern2CommunicationDiagramMain)
    begin
        // Iteramos hasta que
CommunicationPattern2CommunicationDiagramMain->analyisisModel-
>getMessage() devuelva false, es decir, cuando ya se han iterado todos los
mensajes.
        while this->message != false
        do
            if this->message->name matches 'Emit_*' then
                this->EmitMessage2ModArqEmitMessage()
            elseif this->message->name matches 'doOn_*' then
                this->DoOnMessage2ModArqMessages()
            elseif this->message->name matches 'save*' or this->message-
>name matches 'get*' then
                this-
>saveMessage2ModArqMessages(CommunicationPattern2CommunicationDiagramMain)
            fi
            elseif this->message->name matches ['appendTo_*', 'pushTo_*',
'enqueueTo_*'] or
              this->message->name matches ['removeFrom_*', 'popFrom_*',
'dequeueFrom_*'] or
              this->message->name matches 'get_*' or
              this->message->name matches 'replaceIn_*' or
              this->message->name matches 'addContentTo_*' or
              this->message->name matches 'addSubtreeTo_*' or
              this->message->name matches 'removeSubtreeFrom_*' or
              this->message->name matches 'findSubtreeIn_*' or
              this->message->name matches 'removeNodeFrom_*' or
              this->message->name matches 'findNodeIn_*' or
              this->message->name matches 'updateNodeIn_*' or
              this->message->name matches 'addRegisterTo_*' or
              this->message->name matches 'removeRegisterFrom_*' or
              this->message->name matches 'cancelEditIn_*' or
              this->message->name matches 'getRegisterFrom_*' or
              this->message->name matches 'updateRegisterIn_*' or
              this->message->name matches 'setTextTo_*' or
              this->message->name matches 'getTextFrom_*' then
                this->ControlToBoundaryUIMessage2ModArqMessages()
            fi
            this->message = CommunicationPattern2CommunicationDiagramMain-
>analysisModel->getNextMessage(this->currentMessageNumber)
        od
    end
end
```

Tabla 7-47: La clase ClientServerCommPattern2CommDiagram contiene los métodos para realizar la transformación para el patrón cliente-servidor. El método transformMessage itera los mensajes del modelo de análisis y llama al método correspondiente al cada tipo de mensaje encontrado.

7.4.3.5 La regla de transformación DoOnMessage2ModArqMessages

Los mensajes DoOn* son los mensajes de objetos <<boundary>> de tipo UI a objetos <<control>>. La finalidad de los mensajes DoOn* es responder a los eventos del usuario sobre la interfaz de usuario. El patrón que aplica a los nombres de este tipo de mensajes es DoOn<nombre-del-evento> o DoOn<nombre-del-evento>_on_M, donde M es el nombre del target donde ocurrió el evento. Más adelante explicaremos cuando se utiliza cada patrón.

Durante esta transformación podemos reutilizar las componentes que han sido generadas en otros mapeos. Para este caso, usaremos algunas de las *UIVisualComponent* ya generadas en la transformación *EmitMessage2ModArqEmitMessage*. Podemos asumir que estas componentes ya existen porque diseñamos la transformación para el patrón client-server de modo que recorra los mensajes de los modelos de entrada (que son de comunicación de análisis) en orden. Un mensaje DoOn_* puede tener como origen un objeto <<boundary>> de tipo UI o un objeto <<boundary>> de tipo ExternalApp. Como estamos aplicando la transformación al patrón Client-Server, podemos descartar el segundo caso. Por lo tanto, se puede asumir que DoOn_* tiene como origen un objeto <<boundary>> de tipo UI, y que las componentes visuales asociadas a este <<boundary>> ya han sido generadas durante el mapeo de un mensaje Emit_*. Sin embargo, surge un nuevo problema a resolver: ¿Cómo sabemos cuál es el UIVisualComponent que corresponde al mensaje generado? Recordemos que un solo objeto <<boundary>> puede generar más de una UIVisualComponent. Para poder reutilizarlas en *DoOn2ModArqMessages* debemos poder identificar cuál de ellas es la emisora del mensaje. Nuevamente, como en la transformación de los mensajes Emit_*, debemos hacer uso de la función *hasTargetInName* para saber si el mensaje, en este caso de tipo DoOn*, contiene el nombre del emisor o si debemos utilizar directamente el nombre del objeto <<boundary>> y la función *getTargetFromName* para obtener el nombre de la componente generada, en caso de ser necesario. Luego, para recuperar las instancias de componente a partir de sus nombres, utilizamos la función getInstance definida sobre el objeto ObjectMap, que contiene todas las instancias de componente que ya han sido generadas.

En la Figura 7-11 puede verse el resultado de un mapeo uno a uno. En la Figura 7-12 puede verse un ejemplo de mapeo que involucra a dos UIVisualComponentes. Se basan en el caso de uso de un usuario escribiendo en una ventana de chat y luego presionando el boton "send" para enviar dicho mensaje.

Finalmente, debemos indicar en qué paquete, cliente o servidor, se generó la última componente, de modo que método *returnFromServer* funcione correctamente cuando sea utilizado.

Las reglas según cada caso tienen la forma:

<<boundary>> N DoOn<event> <<control>> C

se mapea a

<<UIVisualComponent>> N *DoOn<event>* <<ClientController>> clientController *doRequest* <<AsyncRemoteRequest>> AsyncRemoteRequest *httpRequest* <<Webserver>> webserver *processRequest* <<ServerController>> serverController

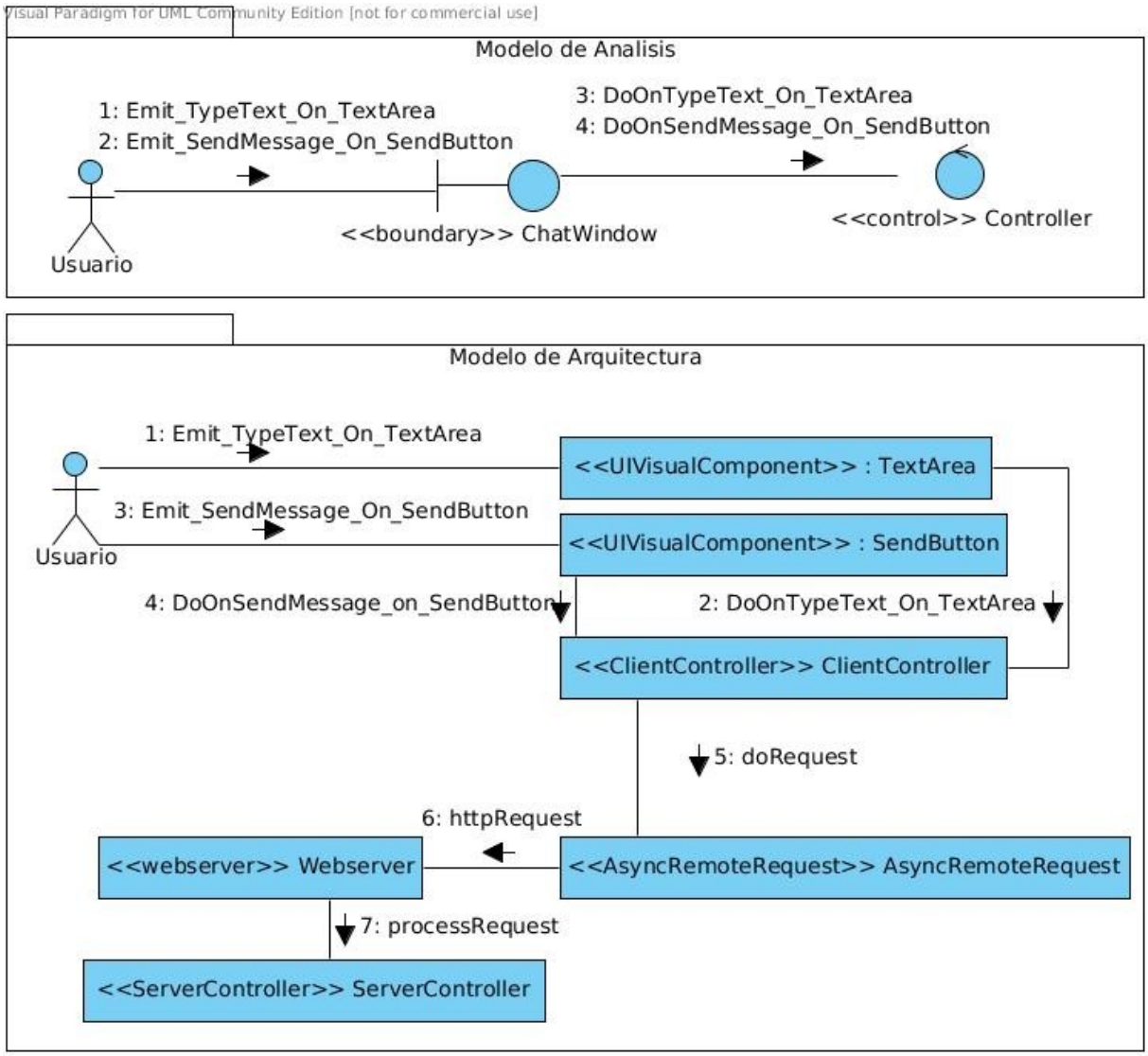

Figura 7-12: Mapeo de mensaje DoOn en el que están involucrados dos UIVisualComponents*

o bien

<<boundary>> N *DoOn<event>_ On_ <targetName>* <<control>> C

se mapea a<<UIVisualComponent>> *objectMap->getInstance((getTargetFromName(DoOn<event>_ On_ <targetName>)))* *DoOn<event>_ On_ <targetName>* <<ClientController>> clientController *doRequest* <<AsyncRemoteRequest>> AsyncRemoteRequest *httpRequest* <<Webserver>> webserver *processRequest* <<ServerController>> serverController

```
class ClientServerCommPattern2CommDiagram extends
CommunicationPattern2CommunicationDiagram
begin
    [...]
    public function DoOnMessage2ModArqMessages()
    begin
        // Sabemos que los mensajes de tipo doOn_* vienen de
UIVisualComponent o de un ExternalApp, los cuales ya han sido generados
dado que los mensajes se iteran en orden. Por lo tanto no hace falta crear
las instancias, solo conectarlas.
        if not this->visitedInstances->exists(this->message->destination)
then
            var clientControllerInstance = this-
>generateComponent('ClientController')
            var asyncRemoteRequestInstance = this-
>generateComponent('AsyncRemoteRequest')
            var webserver = this->generateComponent('WebServer')
            var serverControllerInstance = this-
>generateComponent('ServerConotroller')
            var browser = this->generateComponent('Browser')
        fi
        // obtenemos la instancia de UIVisualComponent ya generada
        if hasTargetInName(this->message)
            var uiVisualComponentInstanceName =  getTargetFromName(this-
>message->getName())
        else
            var uiVisualComponentInstanceName =  getTargetFromName(this-
>message->source->getName())
        end
        var uiVisualComponentInstance = this->objectMap->getInstance(this-
>message->source, uiVisualComponentInstanceName)
        this->archModel->addConnector(uiVisualComponentInstance,
clientControllerInstance)
        this->archModel->addConnector(clientControllerInstance,
asyncRemoteRequestInstance)
        this->archModel->addConnector(asyncRemoteRequestInstance,
webserver)
        this->archModel->addConnector(webserver, serverControllerInstance)
        this->archModel->addConnector(asyncRemoteRequestInstance,
clientControllerInstance)
        this->archModel->addConnector(serverControllerInstance, browser)
        this->archModel->addConnector(browser, asyncRemoteRequestInstance)
        this->archModel->addConnector(asyncRemoteRequestInstance,
clientControllerInstance)

        this->refArchCurrentMessage = this->archModel->addMessage(message-
>name, uiVisualComponentInstance, clientControllerInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('doRequest', clientControllerInstance,
asyncRemoteRequestInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  refArchCurrentMessage
```

```
        this->refArchCurrentMessage = this->archModel-
>addMessage('httpRequest', asyncRemoteRequestInstance, webserver)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('processRequest', webserver, serverControllerInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  refArchCurrentMessage

        this->currentPackage = 'server'
    end
    [...]
end
```

Tabla 7-48: Método de transformación para mensajes de tipo DoOn para el patrón de comunicación cliente-servidor.

7.4.3.6 Generación de componentes del modelo de arquitectura a partir de objetos del modelo de análisis

Todos las componentes en el modelo de arquitectura se generan de la misma manera. Se genera la instancia, se le da un nombre, se agrega al modelo de arquitectura y por último se guarda una referencia de la relación entre el objeto del modelo de análisis y el objeto generado en el modelo de arquitectura. Para evitar repetir esta secuencia una y otra vez en la diferentes transformaciones, definimos el método *generateComponent* que dado un estereotipo, se encarga de aplicar cada uno de estos pasos de generación de una componente y la devuelve. El parámetro *name* se utiliza en los casos en que el nombre de la componente a generar es diferente del nombre del estereotipo. Por ejemplo, los UIVisualComponente pueden tener nombres diferentes, pero los ServerController siempre se llaman ServerController, entonces podemos usar el estereotipo como nombre. El segundo parámetro del método es opcional. Esto quiere decir que si está ausente, se utilizará el nombre del estereotipo como nombre para la componente.

```
class ClientServerCommPattern2CommDiagram extends
CommunicationPattern2CommunicationDiagram
begin
    [...]
    public function generateComponent(stereotype, name = '')
    begin
        objectInstance = this->archModel->createInstance(stereotype)
        if name != ''
            objectInstance->setName(name)
        else
            objectInstance->setName(stereotype)
        end
        this->archModel->addInstance(objectInstance)
        this->objectMap->push(this->message->destination, objectInstance);

        return objectInstance
    end
end
```

Tabla 7-49: Este método auxiliar encapsula el código necesario para generar un objeto del diagrama de comunicaciones de destino a partir de un objeto del diagrama de comunicaciones de origen.

7.4.3.7 La regla de transformación getSaveMessage2ModArqMessages

Para el patrón client-server, los mensajes de tipo get* y save* son enviados por un objeto <<control>> a un objeto <<entity>>. Este tipo de mensajes representan la lectura y el almacenamiento de información de la aplicación tanto de manera persistente como de manera volátil. Esto presenta algunos desafíos en el diseño del mapeo. Una primera idea consiste en mapear un objeto <<entity>> a una componente en el servidor. Sin embargo, luego de analizar varios casos de usos, notamos que los objetos <<entity>> pueden generar componentes tanto en el servidor como en el cliente. Esta información puede obtenerse del modelo de marcas. Este proporciona información sobre que tipo de componentes es necesario generar en el modelo de referencia por cada objeto <<entity>> del modelo de análisis. Utilizaremos método *getStorageType* el cuál nos indicará que tipo de almacenamiento debemos generar dado un objeto <<entity>> del modelo de análisis. En el caso de que el objeto <<entity>> genere una componente de almacenamiento volátil en el cliente, debemos generar los mensajes de vuelta del servidor. Para esto utilizamos la función *returnFromServer.*

La secuencia de mensajes y componentes generadas por esta transformación es:

<<control>> C save* | get* <<entity>> E

se mapea a

<<ServerController>> serverController save* | get* <<DAO>> E
queryDataSource <<DataSourceConnector>> E

o bien

<<ServerController>> serverController *save** | *get** <<DAO>> E
queryDataSource <<DataSourceConnector>> E
{returnFromServer} <<ClientController>> clientController
*save** | *get** <<LocalStore>> E

Donde *returnFromServer*

<<ServerController>> S *respuestaHttp* <<Browser>> B *httpResponse*
<<AsyncRemoteRequest>> A *onEvenDone* <<ClientController>> C

```
class ClientServerCommPattern2CommDiagram extends
CommunicationPattern2CommunicationDiagram
begin
    [...]
    public function
getSaveMessage2ModArqMessages(CommunicationPattern2CommunicationDiagramMai
n)
    begin
        var serverControllerInstance = this->objectMap-
>getInstance('ServerController')
        var daoInstance = this->generateComponent('DAO')
        var dataSourceConnectorInstance = this-
>generateComponent('DataSourceConnector')

        this->archModel->addConnector(serverControllerInstance,
daoInstance)
        this->refArchCurrentMessage = this->archModel->addMessage(this-
>message->getName(), serverControllerInstance, daoInstance)
        this->setRefArchMessageNumber()

        this->archModel->addConnector(daoInstance,
dataSourceConnectorInstance)
        this->refArchCurrentMessage = this->archModel-
>addMessage('queryDataSource', daoInstance, dataSourceConnectorInstance)
        this->setRefArchMessageNumber()

        this->refArchPreviousMessage =  this->refArchCurrentMessage
        this->currentPackage = 'server'

        if this->getStorageType(this->message->destination) == 'volatil'
then
            this->returnFromServer()
            var clientControllerInstance = this->objectMap-
>getInstance('ClientController')
            var localStoreInstance = this->generateComponent('LocalStore')

            this->archModel->addConnector(clientControllerInstance,
localStoreInstance)
            this->refArchCurrentMessage = this->archModel-
>addMessage(this->message->getName(), clientControllerInstance,
localStoreInstance)
            this->setRefArchMessageNumber()

            this->refArchPreviousMessage =  this->refArchCurrentMessage
        end
    end
end
```

Tabla 7-50: Método de transformación de mensajes de tipo GetSave para el patrón cliente-servidor.

7.4.3.8 La regla de transformación ControlToBoundaryUIMessage2ModArqMessages

Este tipo de mensajes representa la actualización de la interfaz de usuario como respuesta a un evento generado por el usuario anteriormente.

El receptor del mensaje en el modelo de arquitectura debe ser un UIVisualComponent. Aquí podemos encontrarnos con dos escenarios. O bien el UIVisualComponent receptor del mensaje ya ha sido generado en una transformación anterior, o bien el destinatario del mensaje es una nueva componente UIVisualComponente. El método *exists* nos indicará si la generación es necesaria o no.

En caso de ser necesario utilizar un UIVisualComponent ya generado, nuevamente hacemos uso de la función *getTargetFromName* para, a partir del nombre del mensaje, obtener el nombre de la componente a generar o a obtener (si ya

ha sido generada anteriormente mediante la aplicación de una regla de transformación).

Otro problema que surge viene dado por el hecho de que el emisor de este mensaje es un objeto <<control>>. Esto quiere decir que el último objeto generado en la transformación anterior al mensaje de <<control>> a <<boundary>> puede haber sido generada en el cliente o en servidor. Es necesario entonces llamar a la función *returnFromServer,* la cuál verificará en que paquete se generó la última componente y generará la secuencia de mensajes de vuelta del servidor en caso de ser necesario.

La secuencia de mensajes generada durante esta transformación es:

Applicability condition: last component generated on client

<<control>> C <*controlToBoundaryUIMessage*> <<boundary>> B

se mapea a:

<<clientController>> clientController <controlToBoundaryUIMessage>
<<UIVisualComponent>> *getTargetFromName (controlToBoundaryUIMessage)*

el otro caso:

Applicability condition: last component generated on server

<<control>> C <*controlToBoundaryUIMessage*> <<boundary>> B

se mapea a:

{returnFromServer}<<clientController>> clientController
<*controlToBoundaryUIMessage*> <<UIVisualComponent>> *getTargetFromName (controlToBoundaryUIMessage)*

Donde *returnFromServer*

<<ServerController>> S *respuestaHttp* <<Browser>> B *httpResponse*
<<AsyncRemoteRequest>> A *onEvenDone* <<ClientController>> C

```
class ClientServerCommPattern2CommDiagram extends
CommunicationPattern2CommunicationDiagram
begin

    public function ControlToBoundaryUIMessage2ModArqMessages()
    begin
        // generamos los mensajes de retorno del servidor si es necesario.
        this->returnFromServer()

        // devuelve el nombre del mensaje en la arquitectura de referencia
        var messageName = getEntityMessageName(this->message->getName())
        // el nombre del mensaje de control a boundary en el modelo de
análisis contiene el nombre del destinatario del mensaje en la
arquitectura de referencia.
        var uiVisualComponentName = getTargetFromName(this->message-
>getName())
        // intentamos obtener la instancia del uiVisualComponent, si no
existe, la generamos.
        var uiVisualComponent = this->objectMap->getInstance(this-
>message->destination, uiVisualComponentName)
        if not this->archModel->exists(uiVisualComponent)
        then
            uiVisualComponent = this-
>generateComponent('UIVisualComponent')
        fi
        // Agregamos el mensaje a la arquitectura de referencia
        this->refArchCurrentMessage = this->archModel-
>addMessage(messageName, clientControllerInstance, uiVisualComponent)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        // indicamos que el último mensaje fue generado en el paquete
cliente.
        this->currentPackage = 'client'
    end
end
```

Tabla 7-51: Método de transformación de mensajes entre objetos de control y objetos boundary de tipo UI para el patrón cliente-servidor.

7.4.3.9 Transformación de modelos de análisis usando el patrón Client-Server-Clients a arquitectura

La clase ClientServerClientsMapper extiende a CommunicationPattern2CommunicationDiagram y tiene como responsabilidad el mapeo de los mensajes del patrón Client-Server-Clients. Para esto, define los métodos requeridos específicamente para este patrón.

```
class ClientServerClientsMapper extends
CommunicationPattern2CommunicationDiagram
begin
    public function transformMessages(communicationPattern,
CommunicationPattern2CommunicationDiagramMain)
    begin
        // Iteramos hasta que
CommunicationPattern2CommunicationDiagramMain->analyisisModel-
>getMessage() devuelva false, es decir, cuando ya se han iterado todos los
mensajes.
        while this->message != false
        do
            if this->message->name matches 'Emit_*' then
                // usa el método declarado en el padre
                this->EmitMessage2ModArqEmitMessage()
            elseif this->message->name matches 'doOn_*' then
                this-
>DoOnMessage2ModArqMessages(CommunicationPattern2CommunicationDiagramMain)
            elseif this->message->name matches 'save*' or this->message-
>name matches 'get*' then
                this-
>mapClientServerClientsGetSaveMessage(CommunicationPattern2CommunicationDi
agramMain)
            elseif this->message->name matches ['appendTo_*', 'pushTo_*',
'enqueueTo_*'] or
              this->message->name matches ['removeFrom_*', 'popFrom_*',
'dequeueFrom_*'] or
              this->message->name matches 'get_*' or
              this->message->name matches 'replaceIn_*' or
              this->message->name matches 'addContentTo_*' or
              this->message->name matches 'addSubtreeTo_*' or
              this->message->name matches 'removeSubtreeFrom_*' or
              this->message->name matches 'findSubtreeIn_*' or
              this->message->name matches 'removeNodeFrom_*' or
              this->message->name matches 'findNodeIn_*' or
              this->message->name matches 'updateNodeIn_*' or
              this->message->name matches 'addRegisterTo_*' or
              this->message->name matches 'removeRegisterFrom_*' or
              this->message->name matches 'cancelEditIn_*' or
              this->message->name matches 'getRegisterFrom_*' or
              this->message->name matches 'updateRegisterIn_*' or
              this->message->name matches 'setTextTo_*' or
              this->message->name matches 'getTextFrom_*' then
                this->ControlToBoundaryUIMessage2ModArqMessages()
            fi
            this->message = CommunicationPattern2CommunicationDiagramMain-
>analysisModel->getNextMessage(this->currentMessageNumber)
        od
    end
end
```

Tabla 7-52: Método de transformación de mensajes para el patrón cliente-servidor-clientes. Itera los mensajes del diagrama de comunicaciones del modelo de análisis y llama al método correspondiente de transformación para cada tipo diferente de mensaje encontrado.

7.4.3.10 La regla de transformación DoOnMessage2ModArqMessages

Los mensajes DoOn* para el patrón Client-Server-Clients representan la notificación a un conjunto de subscriptores sobre la ocurrencia de un evento generado por un cliente emisor. Esto significa que hay más de un rol de cliente envuelto en la comunicación: un rol de cliente emisor del evento y un rol de cliente subscriptor al evento. Para entender mejor esta transformación, podemos dividirla en dos etapas:

1. Mapeo de mensajes y objetos que representan la generación del evento.

2. Mapeo de mensajes y objetos que representan la obtención de los subscriptores

a los que se debe notificar.

Las primeras diferencias las encontramos en la generación de las componentes del paquete servidor. El ServerController delega el procesamiento del evento a una nueva componente, ProcesadorPedidoPersistente, la cual representa un canal de comunicación constante con los clientes suscritos a un evento del servidor.

A partir de aquí comienza la segunda etapa. Obtener los subscriptores al evento. Básicamente, debemos encontrar a aquellos clientes que se han suscrito al evento y notificarles de la ocurrencia del mismo.

Esto introduce varios nuevos problemas a resolver.

El primero esta dado por el hecho de que el mensaje *getSubscriptors*, utilizado para obtener los subscriptores al evento, es un mensaje de <<control>> a <<entity>> cuyo mapeo no es competencia de la transformación de los mensajes DoOn*, sino de la transformación de los mensajes getSave*. Como el mapeo asume que los mensajes se genera en orden ¿cómo hacemos para mapear *getSubscriptors* en medio del mapeo de los DoOn*? La respuesta son los mapeos anidados. Utilizando la función *getNestedMessage(messageName)*, donde messageName es 'getSubscriptors', indicamos que dejamos de mapear por un momento el mensaje DoOn* para mapear getSave* llamando a getSave2ModArqMessage. Una vez finalizado el mapeo de getSave*, continuamos con el mapeo del mensaje DoOn*.

El segundo problema es que debemos representar la iteración de subscriptores en el modelo de análisis y luego mapearla al modelo de arquitectura. Para resolverlo, utilizamos la notación para iteraciones que provee UML.

Por último, debemos decir qué sucede si un cliente no está en línea en el momento en el que ocurre el evento. Para resolver este caso, primero debemos poder expresar una condición que indique si el cliente subscriptor está en línea o no. Para esto utilizamos la notación de guardas provista por UML. Tanto las guardas como las iteraciones pueden extraerse del modelo de análisis, utilizando el método *getConditions.* El patrón nos indica que hay una condición y una iteración. La condición indica si un cliente subscriptor del evento está en línea o no. La iteración recorre los subscriptores. Si están en línea, son notificados en el acto. Si no lo están, la notificación es almacenada en el servidor mediante el mecanismo de persistencia utilizado, para notificar al cliente cuando vuelva a estar online. Por lo tanto las condiciones e iteraciones deben asignarse a los mensajes consecutivos al mensaje que obtiene los subscriptores al evento, en este caso, al mensaje que va de *distribuidorEvento* a *procesadorPedidoPersistente2* y de *distribuidorEventos* a *notificationsHistoryDAO.*

Como el lector ya habrá notado, el mensaje de *distribuidorEventos* a *notificationsHistoryDAO* no pertenece a este mapeo, porque es de objeto <<control>> a objeto <<entity>>. Nuevamente, hacemos uso de la función *getNestedMessage* y anidamos este el mapeo al mapeo de los DoOn*.

En la Figura 7-13 podemos ver el modelo de análisis de un caso de uso que respeta el patrón de comunicación client-server-clients. La Figura 7-14 muestra modelo de

arquitectura resultado de la transformación del modelo de análisis de la Figura 7-13. A modo ilustrativo, dejamos el mensaje *saveHistoryItem*, aunque el lector debe tener en cuenta que en realidad no es generado por esta transformación, sino por *GetSaveMessage2ModArqMessages,* explicada en la siguiente sección.

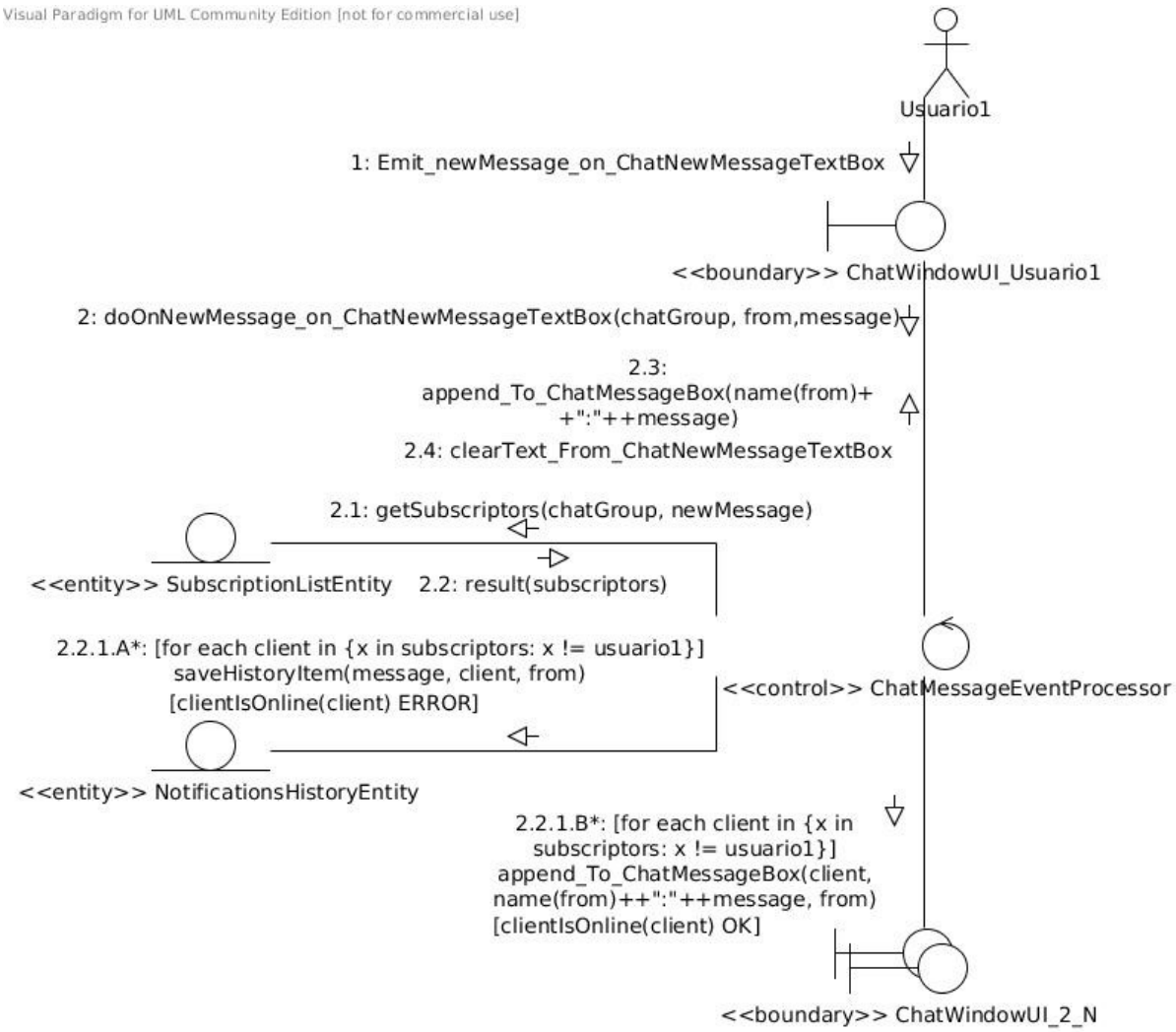

Figura 7-13: Modelo de análisis de un caso de uso cuyo patrón de comunicación es Client-server-clients.

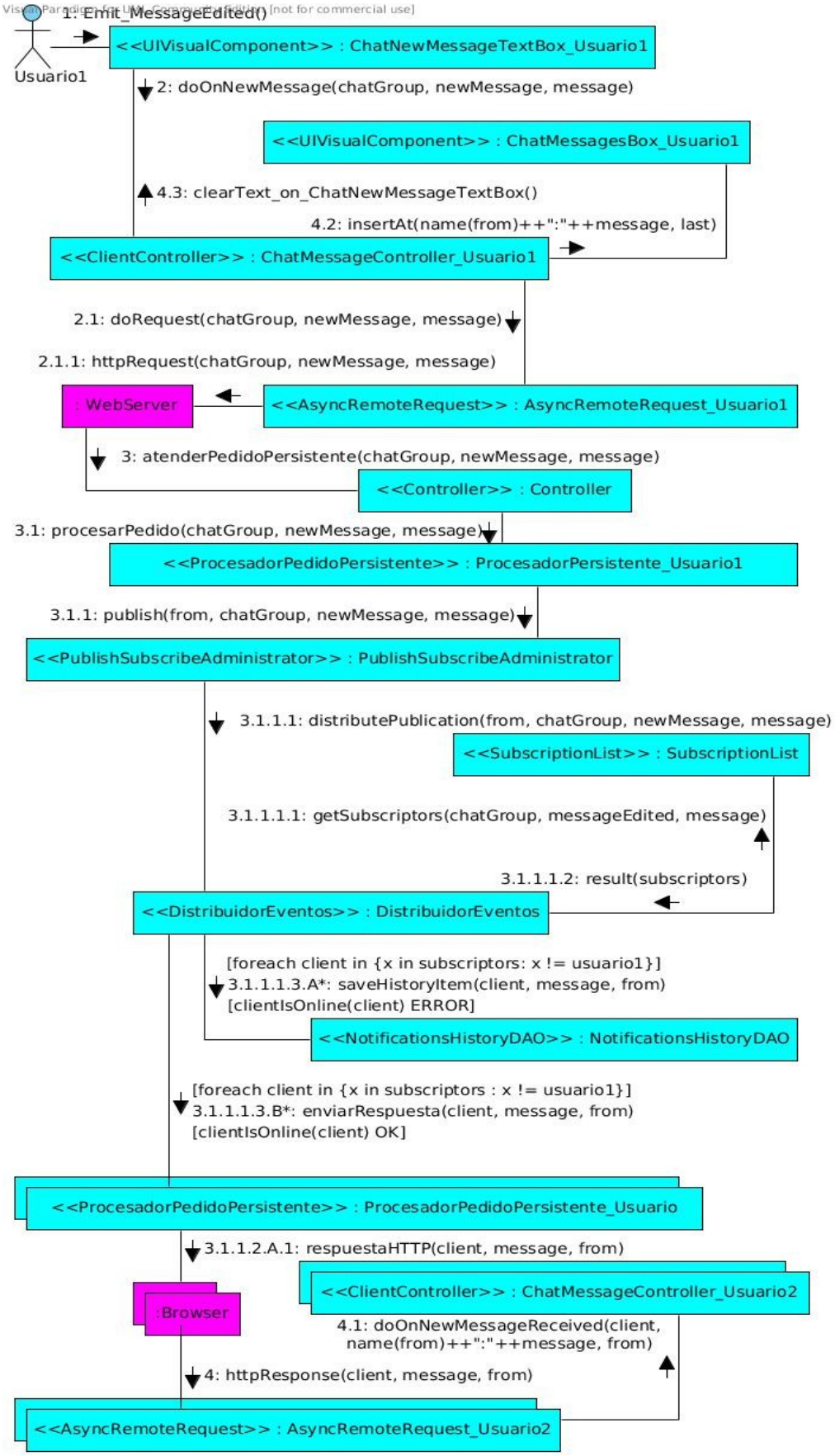

Figura 7-14: Mapeo de mensaje DoOn para el patrón Client-Server-Clients*

La secuencia de mensajes generada durante esta transformación es:

<<boundary>> N DoOn<event> <<control>> C

se mapea a

<<UIVisualComponent>> N *DoOn<event>* <<ClientController>> clientController1 *doRequest* <<AsyncRemoteRequest>> asyncRemoteRequest *httpRequest* <<Webserver>> webserver *processRequest* <<ServerController>> serverController *procesarPedido* <<ProcesadorPedidoPersistente>> procesadorPedidoPersistente1 *publish* <<PublishSubcribeAdministrator>> publishSubscribeAdministrator *distributePublication* <<DistribuidorEventos>> distribuidorEventos *getNestedMessage('getSubscriptors')* GetSaveMessage2ModArqMessages *getNestedMessage('saveHistoryItem')* GetSaveMessage2ModArqMessages

```
class ClientServerClientsMapper extends
CommunicationPattern2CommunicationDiagram
begin
    [...]
    public function
DoOnMessage2ModArqMessages(CommunicationPattern2CommunicationDiagramMain)
    begin
        // Sabemos que los mensajes de tipo doOn_* vienen de
UIVisualComponent o de un ExternalApp, los cuales ya han sido generados
dado que los mensajes se iteran en orden. Por lo tanto no hace falta crear
las instancias, solo conectarlas.
        if not this->visitedInstances->exists(this->message->destination)
then
            var clientControllerInstance1 = this-
>generateComponent('ClientController', 'ClientController1')
            var asyncRemoteRequestInstance1 = this-
>generateComponent('AsyncRemoteRequest', 'AsyncRemoteRequest1')
            var webserver = this->generateComponent('WebServer')
            var serverControllerInstance = this-
>generateComponent('ServerConotroller')

            var procesadorPedidoPersistenteInstance1 = this-
>generateComponent('ProcesadorPedidoPersistente',
'ProcesadorPedidoPersistente1')
            var publishSubscribeAdministratorInstance = this-
>generateComponent('PublishSubscribeAdministrator')
createPublishSubscribeAdministratorInstance()
            var distriubuidorEventosInstance = this-
>generateComponent('DistribuidorEventos')

        fi


        // obtenemos la instancia de UIVisualComponent ya generada
        var uiVisualComponentInstanceName =  getTargetFromName(this-
>message->getName())
        var uiVisualComponentInstance = this->objectMap->getInstance(this-
>message->source, uiVisualComponentInstanceName)
        this->refArchCurrentMessage = this->archModel->addMessage(message-
>name, uiVisualComponentInstance, clientControllerInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('doRequest', clientControllerInstance,
asyncRemoteRequestInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage
```

```
        this->refArchCurrentMessage = this->archModel-
>addMessage('httpRequest', asyncRemoteRequestInstance, webserver)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('processRequest', webserver, serverControllerInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('procesarPedido', serverControllerInstance,
      procesadorPedidoPersistenteInstance1)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage = this->refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('publish', procesadorPedidoPersistenteInstance,
publishSubscribeAdministratorInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('distributePublication',
publishSubscribeAdministratorInstance, distribuidorEventosInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage = this->refArchCurrentMessage

        this->parentMessage = this->message
        this->message = this->message->getNestedMessage('getSubscriptors')
        this->GetSaveMessage2ModArqMessages()
        this->message = this->parentMessage

        this->parentMessage = this->message
        this->message = this->message->getNestedMessage('saveHistoryItem')
        this->GetSaveMessage2ModArqMessages()
        this->message = this->parentMessage

        this->currentPackage = 'client'
    end
    [...]
end
```

Tabla 7-53: Método de transformación de mensajes de tipo DoOn para el patrón de comunicación cliente-server-clients.

7.4.3.11 La regla de transformación GetSaveMessage2ModArqMessages

El método *GetSaveMessage2ModArqMessages* se encarga de la transformación de los mensajes get* y save* de <<control>> a <<entity>>. Los mensajes de esto tipo, en el caso de este patrón, son utilizados para obtener los subscriptores a un evento y para almacenar las notificaciones de eventos que no pudieron ser enviados a los clientes por no estar estos en línea en el momento en que el evento ocurre. Para el patrón Client-Server-Clients, encontramos algunas particularidades. Como explicamos durante el mapeo de los mensajes DoOn* para este patrón, la numeración del mensaje de <<control>> a <<entity>> a generar debe intercalarse con mensajes ya generados y numerados durante el mapeo de los mensajes DoOn*. Es por este motivo, que esta función de mapeo no será llamada desde el ciclo principal de iteración de mensajes, sino desde la función de mapeo de los mensajes DoOn*.

Por otro lado, en el caso del patrón client-server, el emisor del mensaje en el modelo

de arquitectura es un objeto ServerController. En cambio para el patrón Client-Server-Clients, el emisor es un objeto DistribuidorEventos.

También debemos representar la iteración de subscriptores en el modelo de análisis y luego mapearla al modelo de arquitectura. Para resolverlo, utilizamos la notación para iteraciones que provee UML.

La Figura 7-15 y y la Figura 7-16 ilustran el mapeo de este tipo de mensajes para el patrón Client-Server-Clients.

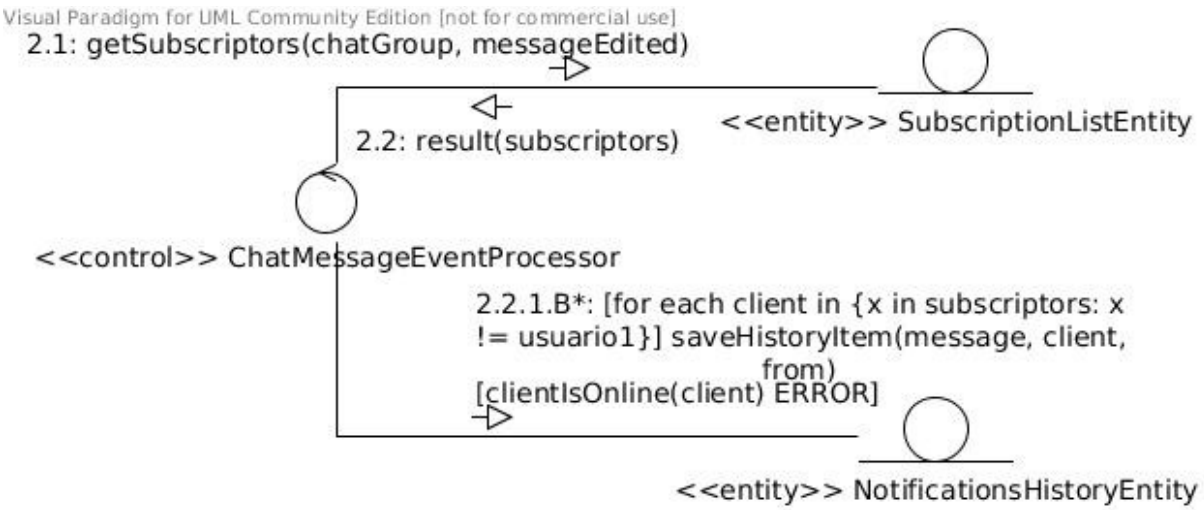

Figura 7-15: En este modelo de análisis se pueden ver las condiciones bajo las cuales se envía el mensaje saveHistoryItem de <<control>> a <<entity>>.

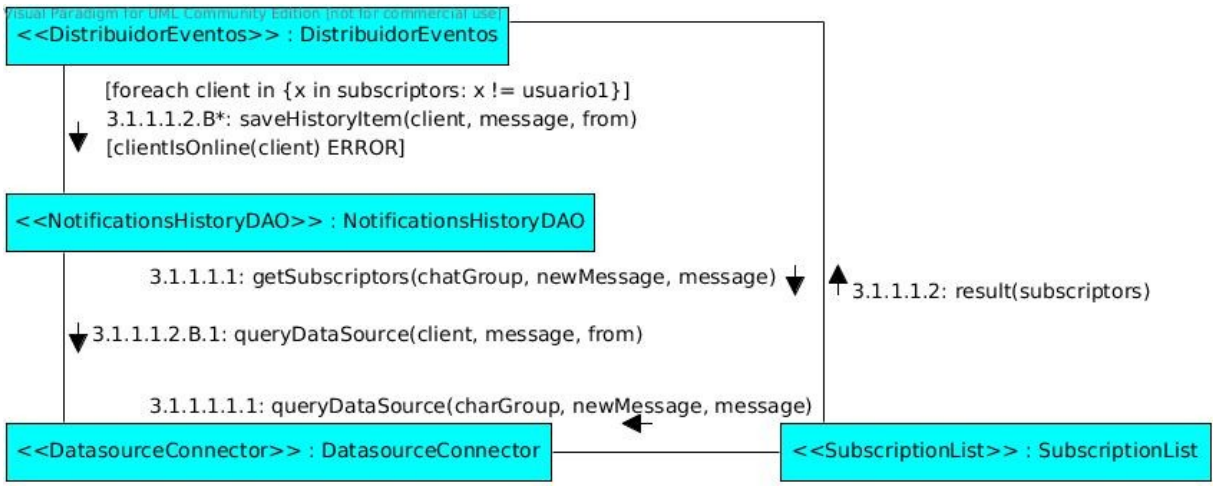

Figura 7-16: Esta figura muestra el resultado del mapeo de los mensajes getSubscriptors y saveHistoryItem.

La secuencia de mensajes y componentes generadas por esta transformación es:

<<control>> C *save** <<entity>> E
se mapea a

<<DistribuidorEventos>> [foreach client in {x in subscriptors: x != usuario1}] distribuidorEventos *save** [clientIsOnline(client) ERROR] <<DAO>> notificationsHistoryDAO

queryDataSource <<DataSourceConnector>> dataSourceConnector

<<control>> C *get** <<entity>> E
se mapea a

<<DistribuidorEventos>> distribuidorEventos get *<<SubsciptionList>>
subscriptionList
queryDataSource <<DataSourceConnector>> dataSourceConnector <<DAO>> dao
result(subscriptors) <<DistribuidorEventos>> distribuidorEventos

```
class ClientServerClientsMapper extends
CommunicationPattern2CommunicationDiagram
begin
    [...]
    public function
GetSaveMessage2ModArqMessages(CommunicationPattern2CommunicationDiagramMai
n)
    begin
        if not this->visitedInstances->exists(this->message->destination)
then
            // verifica si el tipo de localStorage a generar es
persistente (deberia serlo)
            if CommunicationPattern2CommunicationDiagramMain-
>storageTypes->getStorageType(this->message->destination->name) ==
'persistent' then
                var daoInstance = this->generateComponent('DAO')
                var dataSourceConnectorInstance = this-
>generateComponent('DataSourceConnector')

                var distribuidorEventosInstance = this->objectMap-
>getInstance(message->source, 'DistribuidorEventos')

                this->archModel->addConnector(distribuidorEventosInstance,
daoInstance)
                this->refArchCurrentMessage = this->archModel-
>addMessage(message->name, distribuidorEventosInstance, daoInstance)
this->refArchCurrentMessage->addConditions(this->message->getConditions())
                this->refArchPreviousMessage =  this-
>refArchCurrentMessage

                this->archModel->addConnector(daoInstance,
dataSourceConnectorInstance)
                this->refArchCurrentMessage = this->archModel-
>addMessage('queryDataSource', daoInstance, dataSourceConnectorInstance)
                this->refArchPreviousMessage =  this-
>refArchCurrentMessage

                if this->message->name matches 'get*' then
                    this->refArchCurrentMessage = this->archModel-
>addMessage('results(subscriptors)', daoInstance,
distribuidorEventosInstance)
                    this->refArchPreviousMessage =  this-
>refArchCurrentMessage
                fi
            fi
        fi
    end
end
```

Tabla 7-54: Método de transformación de mensajes GetSave para el patrón cliente-servidor-clientes.

7.4.3.12 La regla de transformación ControlToBoundaryUIMessage2ModArqMessages

Este método *ControlToBoundaryUIMessage2ModArqMessages* define el mapeo de los mensajes que van de objetos de control a objetos boundary de tipo UI para el patrón

Client-Server-Clients en particular. En este caso, hay dos tipos de mensaje. Por un lado, tenemos los mensajes de <<control>> a <<boundary>> de tipo UI que representan la notificación a los clientes subscriptores sobre la ocurrencia de un evento. Por otro lado, en ocasiones es preciso procesar el evento del emisor en el mismo emisor.

Primero trataremos los mensajes para notificar a los clientes subscriptores que se encuentran en línea. Podemos identificar varios problemas que necesitan solución. Por una parte, deberemos ser cuidadosos con los nombres que le damos a las instancias de las componentes generadas debido a que hay al menos dos tipos de clientes involucrados en esta transformación, pero ambos utilizan los mismos tipos de componentes.
Por otro lado, es claro que puede haber varias instancias de cliente de rol receptor del evento; pero como no estamos a nivel de ejecución no podemos hablar de cuantos clientes de un cierto rol hay en determinado momento. A lo más, podemos decir que tenemos varios clientes de un cierto rol. La forma correcta de expresar esto en UML son los multiobjetos. Para marcar las componentes como multiobjetos, definimos el método *setMultiobject.* Las componentes representadas por medio de multiobjects son: UIVisualComponent, ClientController, AsyncRemoteRequest, ProcesadorPedidoPersistente y LocalStore. Todos estos multiobjetos corresponden al paquete cliente para los roles de clientes subscriptores al evento, excepto ProcesadorPedidoPersistente que se incluye en el paquete servidor.

El lector puede notar que es necesario generar la secuencia de mensajes que expresan la vuelta del servidor. Sin embargo, en este caso no es necesario utilizar el método *returnFromServer* ya que todos los mensajes y componentes son generados en esta misma transformación.

Las Figuras 7-17 y 7-18 muestran el fragmento de modelo de análisis y los mensajes y objetos generados para este tipo de mensajes en el modelo de arquitectura. Extraídos del caso de uso de una comunicación de dos usuarios a través de un chat.

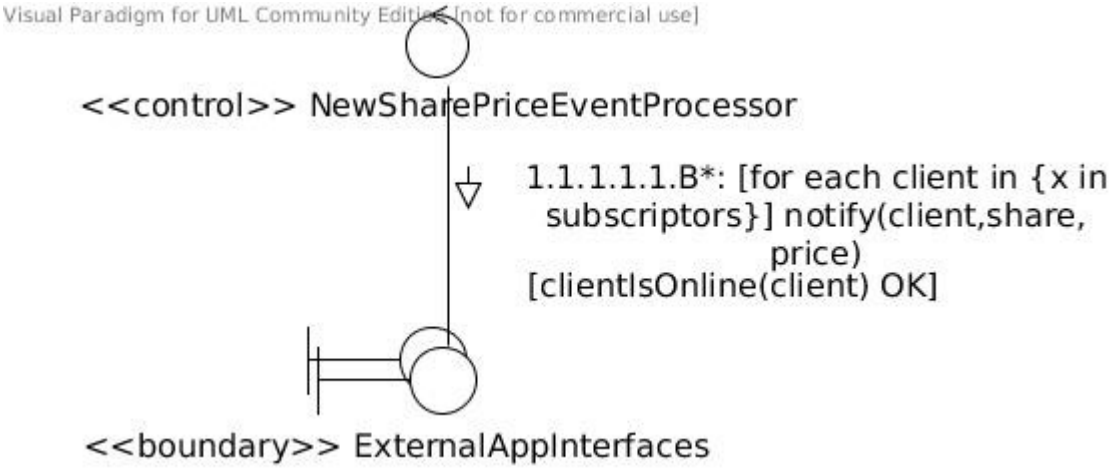

Figura 7-17: Mensaje de <<control>> a <<boundary>> de tipo UI en el modelo de análisis para el patrón Client-server-clients.

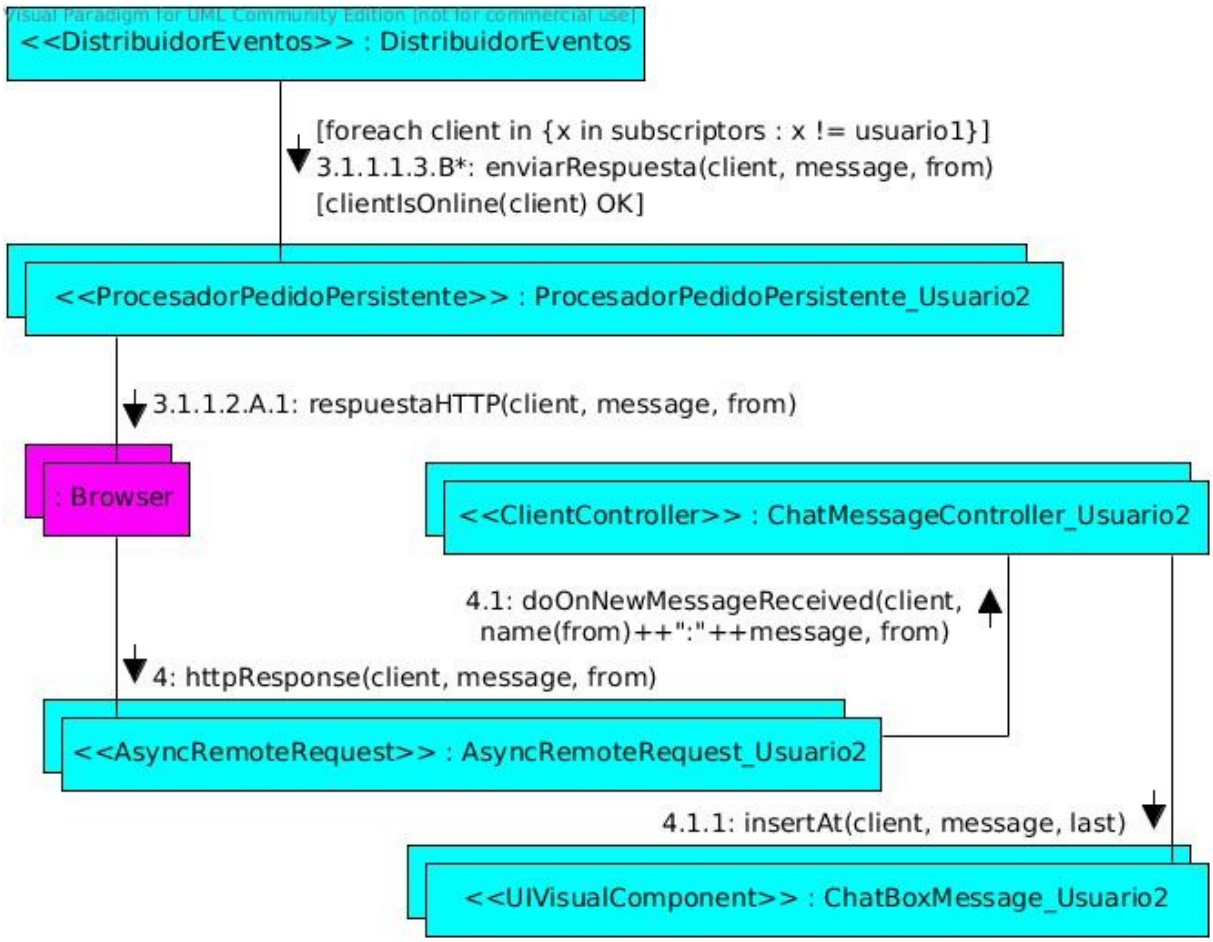

Figura 7-18: Resultado del mapeo de un mensaje de <<control>> a <<boundary>> de tipo UI en el modelo de arquitectura para el patrón Client-server-clients

La secuencia de mensajes y componentes generadas por esta transformación es:

<<control>> C *update* <<boundary>> B

en el caso de mensajes a clientes con rol receptor se mapea a

<<DistribuidorEventos>> distribuidorEventos [foreach client in { x in subcriptors: x != usuario1 }] *enviarRespuesta* [clientOnline(client) OK] <<ProcesadorPedidoPersistente>> *setMultiobject*(procesadorPedidoPersistente2) *respuestaHttp* <<Browser>> *setMultiobject*(browser) *httpResponse* <<AsyncRemoteRequest>> *setMultiobject*(asyncRemoteRequest2) *doOn<event>Notification* <<ClientController>> *setMultiobject*(clientController2)

<<ClientController>> clientController *update* <<UIVisualComponent>> *getTargetFromName(*B*)*

en el caso de mensajes a cliente con rol emisor, se mapea a

<<ServerController>> serverController *respuestaHttp* <<Browser>> browser *httpResponse* <<AsyncRemoteRequest>> asyncRemoteRequest1 *doOnNotificationDone* <<ClientController>> clientController1 *update* <<UIVisualComponent>> *getTargetFromName(*B*)*

```
class ClientServerClientsMapper extends
```

```
CommunicationPattern2CommunicationDiagram
begin

    public function ControlToBoundaryUIMessage2ModArqMessages()
    begin
        if this->message->source->stereotype == 'DistribuidorEventos' then
            this->ControlToBoundaryUIMessage2ModArqMessagesSubscribers()
        else
            this->ControlToBoundaryUIMessage2ModArqMessagesEmitters()
        fi
    end

    public function ControlToBoundaryUIMessage2ModArqMessagesEmitter()
    begin
        // generamos los mensajes de retorno del servidor si es necesario.
        this->returnFromServer()

        // devuelve el nombre del mensaje en la arquitectura de referencia
        var messageName = getEntityMessageName(this->message->getName())
        // el nombre del mensaje de control a boundary en el modelo de
análisis contiene el nombre del destinatario del mensaje en la
arquitectura de referencia.
        var uiVisualComponentName = getTargetFromName(this->message-
>getName())
        // intentamos obtener la instancia del uiVisualComponent, si no
existe, la generamos.
        var uiVisualComponent = this->objectMap->getInstance(this-
>message->destination, uiVisualComponentName)
        if not this->archModel->exists(uiVisualComponent)
        then
            uiVisualComponent = this-
>generateComponent('UIVisualComponent')
        fi
        // Agregamos el mensaje a la arquitectura de referencia
        this->refArchCurrentMessage = this->archModel-
>addMessage(messageName, clientControllerInstance, uiVisualComponent)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        // indicamos que el último mensaje fue generado en el paquete
cliente.
        this->currentPackage = 'client'
    end

    public function ControlToBoundaryUIMessage2ModArqMessagesSubscribers()
    begin
        var distribuidorEventosInstance = this->objectMap-
>getInstance('DistribuidorEventos')
        var procesadorPedidoPersistenteInstance2 = this-
>generateComponent('ProcesadorPedidoPersistente',
'ProcesadorPedidoPersistente2')
        var browser = this->archModel->createInstance('Browser')
        var clientControllerInstance2 = this-
>generateComponent('ClientController', 'ClientController2')
        var asyncRemoteRequestInstance2 = this-
>generateComponent('AsyncRemoteRequest', 'AsyncRemoteRequest2')

        // indicamos que el objeto procesadorPedidoPersistente y browser
son multiobjects
        procesadorPedidoPersistenteInstance2->setMultiobject(true)
        clientControllerInstance2->setMultiobject(true)
        asyncRemoteRequestInstance2->setMultiobject(true)
        browser->setMultiobject(true)

        this->refArchCurrentMessage = this->archModel-
>addMessage('enviarRespuesta', distribuidorEventosInstance,
procesadorPedidoPersistenteInstance2)
        this->setRefArchMessageNumber()
```

```
        this->refArchPreviousMessage = this->refArchCurrentMessage
        this->refArchPreviousMessage->addConditions(this->message-
>getConditions())

        this->refArchCurrentMessage = this->archModel-
>addMessage('respuestaHTTP', procesadorPedidoPersistenteInstance2,
browser)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('httpResponse', browser, asyncRemoteRequestInstance2)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('doOnNotificationDone', asyncRemoteRequestInstance2,
clientControllerInstance2)
        this->setRefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        // devuelve el nombre del mensaje en la arquitectura de referencia
        var messageName = getEntityMessageName(this->message->name)
        // el nombre del mensaje de control a boundary en el modelo de
análisis contiene el nombre del destinatario del mensaje en la
arquitectura de referencia.
        var uiVisualComponentName = getTargetFromName(this->message->name)
        // intentamos obtener la instancia del uiVisualComponent, si no
existe, la generamos.
        var uiVisualComponent = this->objectMap->getInstance(this-
>message->destination, uiVisualComponentName)
        if not this->archModel->exists(uiVisualComponent)
        then
            uiVisualComponent = this-
>generateComponent('UIVisualComponent')
            uiVisualComponent->setMultiobject(true)
        fi
        // Agregamos el mensaje a la arquitectura de referencia
        this->refArchCurrentMessage = this->archModel-
>addMessage(messageName, clientControllerInstance, uiVisualComponent)
        this->setRefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        // indicamos que el último mensaje fue generado en el paquete
cliente.
        this->currentPackage = 'client'
    end
end
```

Tabla 7-55: Método de transformación de mensajes de control a objetos de frontera de tipo UI para el patrón cliente-servidor-clientes.

7.4.3.13 Transformación de modelos de análisis usando el patrón ExternalApp-Server-Clients a arquitectura

La clase ExternalAppServerClientsMapper extiende a CommunicationPattern2CommunicationDiagram y tiene como responsabilidad el mapeo de los mensajes del patrón ExternalApp-Server-Clients. Para esto, define los métodos requeridos específicamente para este patrón. Dada la similitud entre ExternalApp-Server-Clients y Client-Server-Clients, hacemos que ExternalAppServerClientsMapper herede de ClientServerClientsMapper, para poder reutilizar los métodos que se comportan de igual manera en ambos patrones.

```
class ExternalAppServerClientsMapper extends
ClientServerClientsCommunicationPattern2CommunicationDiagram
begin
    public function transformMessages(communicationPattern,
CommunicationPattern2CommunicationDiagramMain)
    begin
        // Iteramos hasta que
CommunicationPattern2CommunicationDiagramMain->analyisisModel-
>getMessage() devuelva false, es decir, cuando ya se han iterado todos los
mensajes.
        while this->message != false
        do
            if this->message->name = 'notify' then
                this->transformNotifyMessage()
            elseif this->message->name matches 'doOn_*' then
                this->DoOnMessage2ModArqMessages()
            elseif this->message->name matches 'save*' or this->message-
>name matches 'get*' then
                this-
>mapClientServerClientsGetSaveMessage(CommunicationPattern2CommunicationDi
agramMain)
            elseif this->message->name matches ['appendTo_*', 'pushTo_*',
'enqueueTo_*'] or
              this->message->name matches ['removeFrom_*', 'popFrom_*',
'dequeueFrom_*'] or
              this->message->name matches 'get_*' or
              this->message->name matches 'replaceIn_*' or
              this->message->name matches 'addContentTo_*' or
              this->message->name matches 'addSubtreeTo_*' or
              this->message->name matches 'removeSubtreeFrom_*' or
              this->message->name matches 'findSubtreeIn_*' or
              this->message->name matches 'removeNodeFrom_*' or
              this->message->name matches 'findNodeIn_*' or
              this->message->name matches 'updateNodeIn_*' or
              this->message->name matches 'addRegisterTo_*' or
              this->message->name matches 'removeRegisterFrom_*' or
              this->message->name matches 'cancelEditIn_*' or
              this->message->name matches 'getRegisterFrom_*' or
              this->message->name matches 'updateRegisterIn_*' or
              this->message->name matches 'setTextTo_*' or
              this->message->name matches 'getTextFrom_*' then
                this->ControlToBoundaryUIMessage2ModArqMessages()
            fi
            this->message = CommunicationPattern2CommunicationDiagramMain-
>analysisModel->getNextMessage(this->currentMessageNumber)
        od
    end
end
```

Tabla 7-56: Método de transformación de mensajes para el patrón externalapp-servidor-clientes. Itera los mensajes del diagrama de comunicaciones del modelo de análisis y llama al método correspondiente de transformación para cada tipo diferente de mensaje encontrado.

7.4.3.14 La regla de transformación transformNotifyMessage

Este método mapea los mensajes de *notify*, generados por una aplicación externa para avisar a la aplicación principal la ocurrencia de un evento. Aquí tenemos un escenario que comienza de manera diferente a los analizados anteriormente. En primer lugar, el actor que genera el evento ya no es un usuario sino una aplicación externa. En segundo lugar, el primer mensaje no ocurre en el paquete cliente, sino en el paquete administrador eventos del servidor. Sin embargo, el diseño no presenta ninguna particularidad. Simplemente se generan los mensajes y las componentes si no han sido generadas aún.

La Figura 7-19 y 7-20 ilustran el resultado del mapeo de este tipo de mensajes, basado en un ejemplo en el que una aplicación externa informa a la aplicación principal sobre el cambio en el precio de una acción de la bolsa de valores.

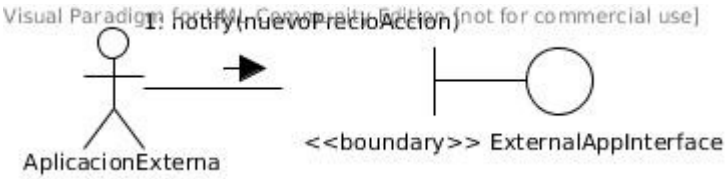

Figura 7-19: Mensaje notify en un modelo de análisis.

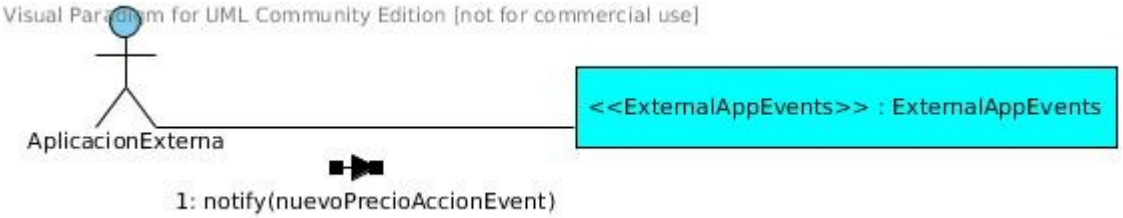

Figura 7-20: Resultado del mapeo del mensaje notify.

La secuencia de mensajes y componentes generadas por esta transformación es:

<<Actor>> aplicacionExterna *notify* <<boundary>> extenalAppInterface

mapea a

<<Actor>> aplicacionExterna *notify* <<ExternalAppEvents>> extenalAppInterface

```
class ExternalAppServerClientsMapper
begin
    [...]
    public function transformNotifyMessage()
    begin
    // sabemos que el source en este caso es un ExternalApp, ya que son
las únicas que pueden generar eventos de tipo 'notify'
        if not this->visitedInstances->exists(this->message->source) then
            var externalAppInstance = this-
>generateComponent('ExternalApp')
        fi
        // Verificamos que todavía no se haya visitado este objeto del
modelo de análisis
        if not this->visitedInstances->exists(this->message->destination)
then
            var externalAppInterfaceInstance = this-
>generateComponent('ExternalAppEvents')
        fi
        // Agregamos el conector y el mensaje a la arquitectura de
referencia.
        this->archModel->addConnector(externalAppInstance,
externalAppInterfaceInstance)
        refArchCurrentMessage = this->archModel-
>addMessage(externalAppInstance, externalAppInterfaceInstance, this-
>message->name)
        this->setRefArchMessageNumber()
        refArchitecturePreviousMessage = refArchCurrentMessage
    end
    [...]
end
```

Tabla 7-57: Método de transformación de mensajes Notify para el patrón externalapp-servidor-

clientes.

7.4.3.15 La regla de transformación DoOnMessage2ModArqMessages

La función DoOnMessage2ModArqMessages para el patrón ExternalApp-Server-Clients lo definimos de la misma manera que para el patrón Client-Server-Clients, ya que el flujo de la transformación es el mismo, excepto que no es necesario generar componentes para el cliente emisor del evento, por supuesto, ya que el emisor del evento es una aplicación externa. Los métodos GetSaveMessage2ModArqMessage y ControlToBoundaryUI2ModArqMessages pueden reutilizarse completamente los definidos para el patrón Client-Server-Clients.

La Figura 7-21 y 7-22 ilustran el mapeo de este tipo de mensajes. Seguimos utilizando el ejemplo de una aplicación externa que notifica a la aplicación principal sobre el cambio en un precio.

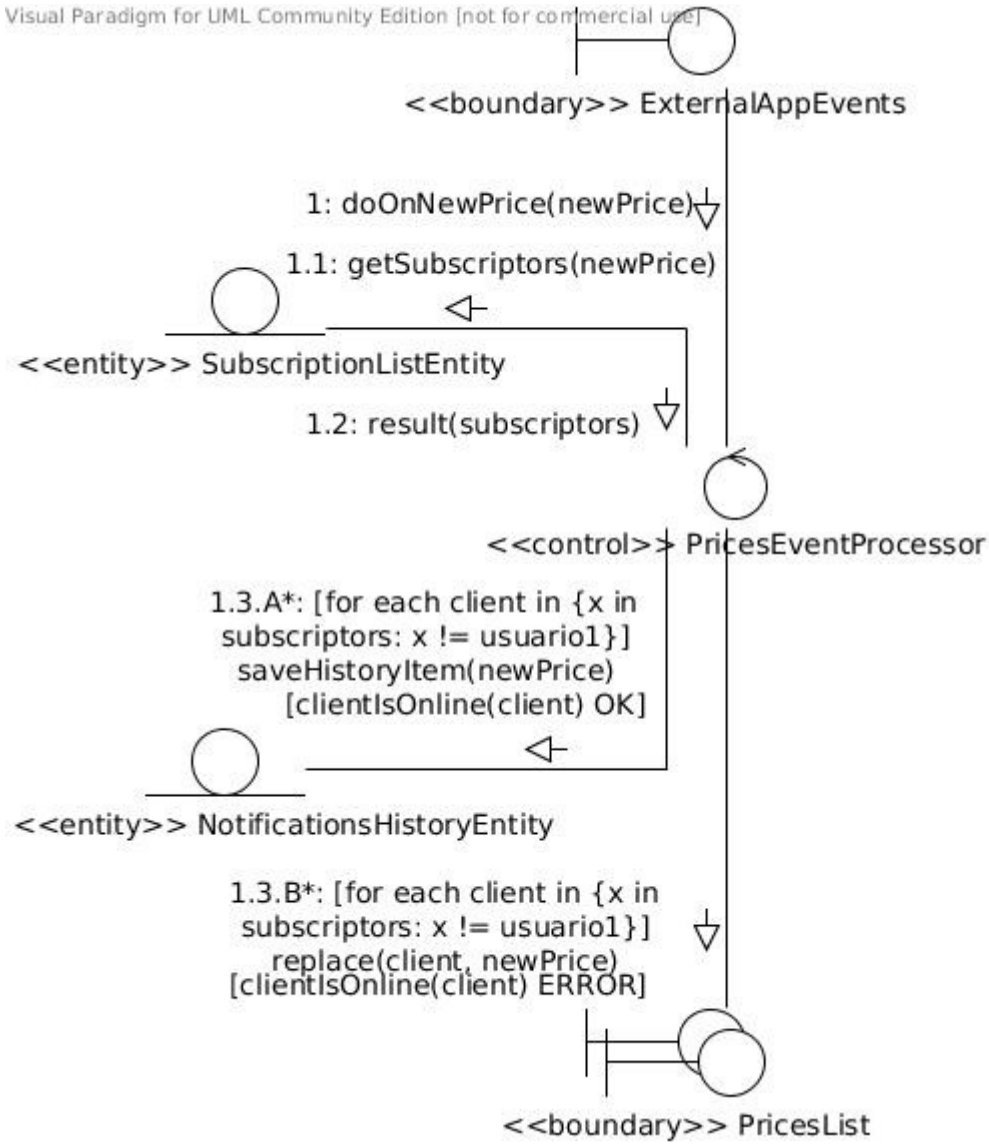

Figura 7-21: Mensaje DoOn en el modelo de análisis.

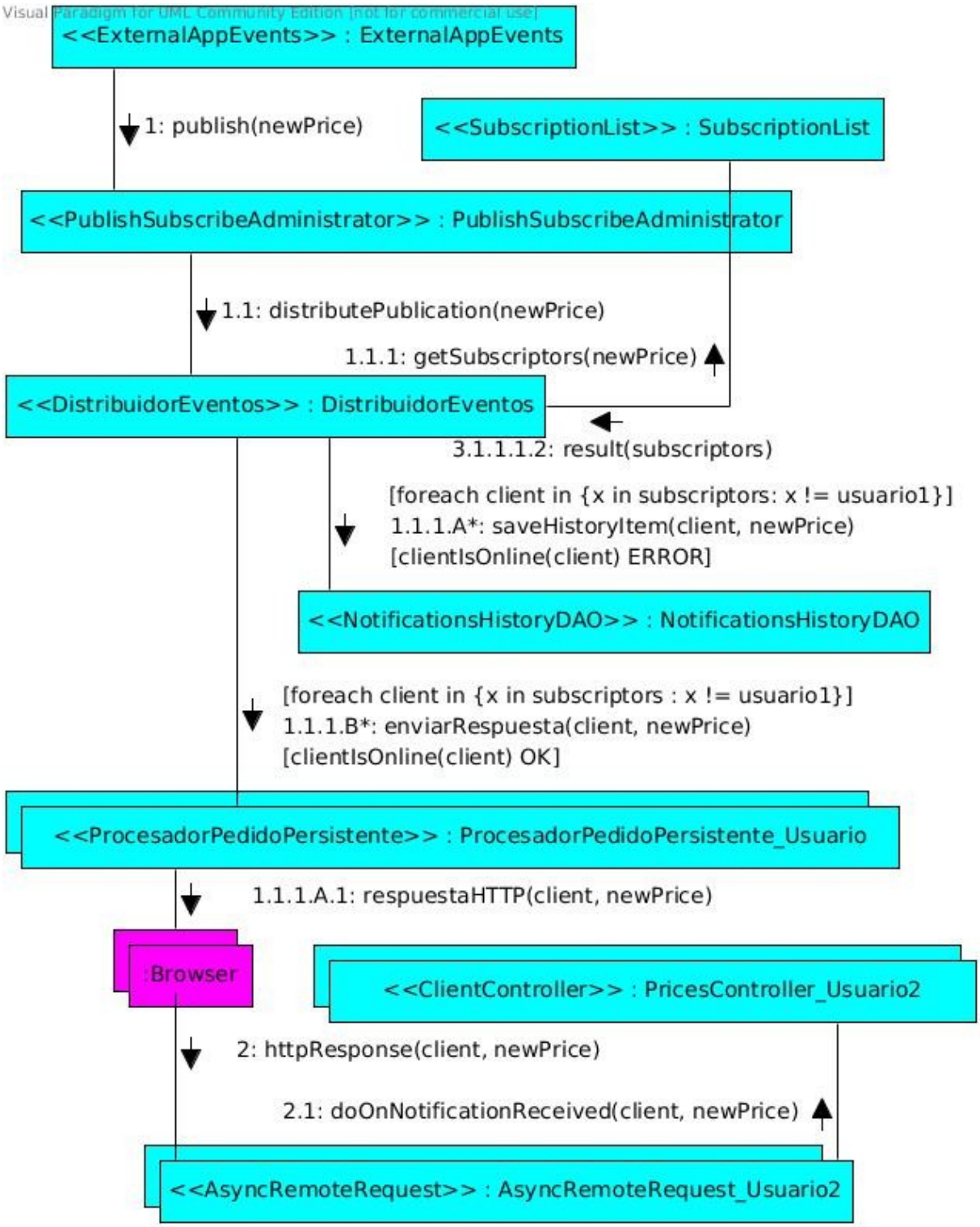

Figura 7-22: Resultado del mapeo del mensaje DoOn de la Figura 8-13.

La secuencia de mensajes generada durante esta transformación es:

<<boundary>> N DoOn<event> <<control>> C

se mapea a

<<ExternalAppEvents>> N *publish* <<PublishSubcribeAdministrator>>

publishSubscribeAdministrator *distributePublication* <<DistribuidorEventos>>
distribuidorEventos *getNestedMessage('getSubscriptors')*
GetSaveMessage2ModArqMessages *getNestedMessage('saveHistoryItem')*
GetSaveMessage2ModArqMessages

```
class ExternalAppServerClientsMapper extends ClientServerClientsMapper
begin
    [...]
    public function DoOnMessage2ModArqMessages()
    begin
        // Sabemos que los mensajes de tipo doOn_* vienen de
UIVisualComponent o de un ExternalApp, los cuales ya han sido generados
dado que los mensajes se iteran en orden. Por lo tanto no hace falta crear
las instancias, solo conectarlas.
        if not this->visitedInstances->exists(this->message->destination)
then
            var externalAppInterface = this->objectMap-
>getInstance('ExteralAppEvents')
            var publishSubscribeAdministratorInstance = this-
>generateComponent('PublishSubscribeAdministrator')
createPublishSubscribeAdministratorInstance()
            var distriubuidorEventosInstance = this-
>generateComponent('DistribuidorEventos')
        fi

this->archModel->addMessage('publish', externalAppInterface,
publishSubscribeAdministratorInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('distributePublication',
publishSubscribeAdministratorInstance, distribuidorEventosInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage = this->refArchCurrentMessage

        this->parentMessage = this->message
        this->message = this->message->getNestedMessage('getSubscriptors')
        this->GetSaveMessage2ModArqMessages()
        this->message = this->parentMessage

        this->parentMessage = this->message
        this->message = this->message->getNestedMessage('saveHistoryItem')
        this->GetSaveMessage2ModArqMessages()
        this->message = this->parentMessage

        this->currentPackage = 'client'
    end
    [...]
end
```

Tabla 7-58: Método de transformación de mensajes DoOn para el patrón externalapp-servidor-clientes.

7.4.3.16 Transformación de modelos de análisis usando el patrón Client-Server-ExternalApps a arquitectura

La clase ClientServerExternalAppsMapper extiende a ClientServerClientsMapper y tiene como responsabilidad el mapeo de los mensajes del patrón Client-Server-ExternalApps. Para esto, define los métodos requeridos específicamente para este patrón. Dado que el patrón presenta muchas similitudes con el patrón Client-Server-Clients, podemos reutilizar algunos de los métodos sin necesidad de reescribirlos. Por

ejemplo, podemos reutilizar completamente el método GetSaveMessage2ModArqMessages.

```
class ClientServerExternalAppsMapper extends ClientServerClientsMapper
begin
    public function transformMessages(communicationPattern,
CommunicationPattern2CommunicationDiagramMain)
    begin
        // Iteramos hasta que
CommunicationPattern2CommunicationDiagramMain->analyisisModel-
>getMessage() devuelva false, es decir, cuando ya se han iterado todos los
mensajes.
        while this->message != false
        do
            if this->message->name matches 'Emit_*' then
                // usa el método declarado en el padre
                this->EmitMessage2ModArqEmitMessage()
            elseif this->message->name matches 'doOn_*' then
                this->DoOnMessage2ModArqMessages()
            elseif this->message->name matches 'save*' or this->message-
>name matches 'get*' then
                this-
>GetSaveMessage2ModArqMessages(CommunicationPattern2CommunicationDiagramMa
in)
            elseif this->message->name matches 'notifyExternalApp'
                this->NotifyExternalAppMessage2ModArqMessages()
            fi
            this->message = CommunicationPattern2CommunicationDiagramMain-
>analysisModel->getNextMessage(this->currentMessageNumber)
        od
    end
end
```

Tabla 7-59: Método de transformación de mensajes para el patrón cliente-servidor-externalapps. Itera los mensajes del diagrama de comunicaciones del modelo de análisis y llama a los métodos correspondientes de transformación de acuerdo al tipo de mensaje encontrado.

7.4.3.17 La regla de transformación DoOnMessage2ModArqMessages

El diseño de esta transformación sigue los mismos lineamientos que para el patrón Client-Server-Clients, al punto que podemos reutilizar completamente el método definido para ese patrón.

La secuencia de mensajes generada durante esta transformación es:

<<boundary>> N DoOn<event> <<control>> C

se mapea a

<<UIVisualComponent>> N *DoOn<event>* <<ClientController>>
clientController1 *doRequest* <<AsyncRemoteRequest>> asyncRemoteRequest
httpRequest <<Webserver>> webserver *processRequest* <<ServerController>>
serverController *procesarPedido* <<ProcesadorPedidoPersistente>>
procesadorPedidoPersistente1 *publish* <<PublishSubcribeAdministrator>>
publishSubscribeAdministrator *distributePublication* <<DistribuidorEventos>>
distribuidorEventos *getNestedMessage('getSubscriptors')*
GetSaveMessage2ModArqMessages *getNestedMessage('saveHistoryItem')*
GetSaveMessage2ModArqMessages

```
class ClientServerExternalAppsMapper extends ClientServerClientsMapper
begin
    [...]
    public function DoOnMessage2ModArqMessages()
    begin
                // Sabemos que los mensajes de tipo doOn_* vienen de
UIVisualComponent o de un ExternalApp, los cuales ya han sido generados
dado que los mensajes se iteran en orden. Por lo tanto no hace falta crear
las instancias, solo conectarlas.
        if not this->visitedInstances->exists(this->message->destination)
then
            var clientControllerInstance1 = this-
>generateComponent('ClientController', 'ClientController1')
            var asyncRemoteRequestInstance1 = this-
>generateComponent('AsyncRemoteRequest', 'AsyncRemoteRequest1')
            var webserver = this->generateComponent('WebServer')
            var serverControllerInstance = this-
>generateComponent('ServerConotroller')

            var procesadorPedidoPersistenteInstance1 = this-
>generateComponent('ProcesadorPedidoPersistente',
'ProcesadorPedidoPersistente1')
            var publishSubscribeAdministratorInstance = this-
>generateComponent('PublishSubscribeAdministrator')
createPublishSubscribeAdministratorInstance()
            var distriubuidorEventosInstance = this-
>generateComponent('DistribuidorEventos')

        fi

        // obtenemos la instancia de UIVisualComponent ya generada
        var uiVisualComponentInstanceName =  getTargetFromName(this-
>message->getName())
        var uiVisualComponentInstance = this->objectMap->getInstance(this-
>message->source, uiVisualComponentInstanceName)
        this->refArchCurrentMessage = this->archModel->addMessage(message-
>name, uiVisualComponentInstance, clientControllerInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('doRequest', clientControllerInstance,
asyncRemoteRequestInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('httpRequest', asyncRemoteRequestInstance, webserver)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('processRequest', webserver, serverControllerInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('procesarPedido', serverControllerInstance,
procesadorPedidoPersistenteInstance1)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage = this->refArchCurrentMessage

        this->refArchCurrentMessage = this->archModel-
>addMessage('publish', procesadorPedidoPersistenteInstance,
publishSubscribeAdministratorInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage
```

```
        this->refArchCurrentMessage = this->archModel-
>addMessage('distributePublication',
publishSubscribeAdministratorInstance, distribuidorEventosInstance)
        this->setrefArchMessageNumber()
        this->refArchPreviousMessage = this->refArchCurrentMessage

        this->parentMessage = this->message
        this->message = this->message->getNestedMessage('getSubscriptors')
        this->GetSaveMessage2ModArqMessages()
        this->message = this->parentMessage

        this->parentMessage = this->message
        this->message = this->message->getNestedMessage('saveHistoryItem')
        this->GetSaveMessage2ModArqMessages()
        this->message = this->parentMessage

        this->currentPackage = 'client'
    end
    [...]
end
```

Tabla 7-60: Método de transformación de mensajes DoOn para el patrón cliente-servidor-externalapps.

7.4.3.18 La regla de transformación controlToBoundaryExternalAppx

Este método es el encargado de transformar los mensajes de <<control>> a <<boundary>> de tipo ExernalApp. El diseño no presenta particularidades. Simplemente se generan las componentes y los mensajes necesarios.

La Figura 7-23 y 7-24 ilustran el mapeo de este tipo de mensajes. El escenario del ejemplo es un usuario que actualiza el precio de una acción de la bolsa de valores, y esto dispara un evento al cual está suscrita una o más aplicaciones externas.

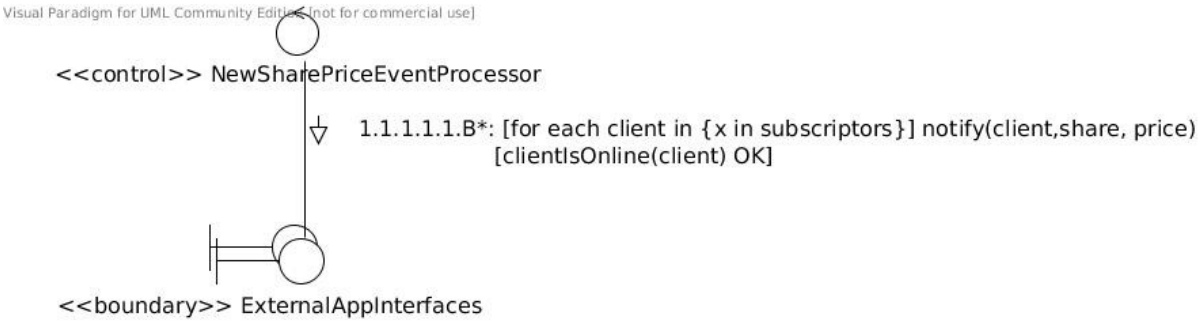

Figura 7-23: Notificación de eventos a aplicaciones externas en el modelo de análisis.

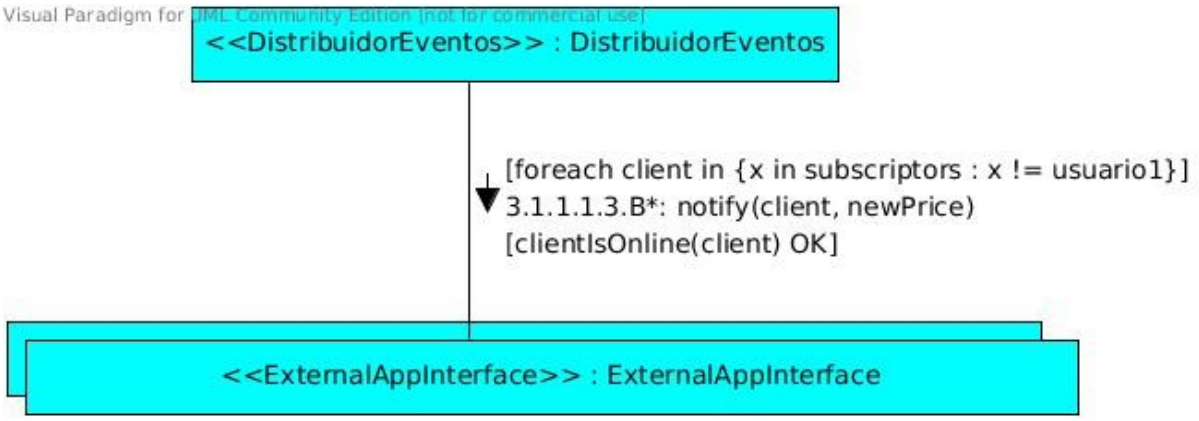

Figura 7-24: Resultado del mapeo del diagrama de la Figura 7-23.

El mapeo es el siguiente:

<<control>> C *notifyExternalApp* <<boundary>> ExternalApp

mapeo a

<<DistribuidorEventos>> distribuidorEventos [foreach client in {x in subscriptors: x != usuario1] *notify* [clientIsOnline(client) OK] <<ExternalAppInterface>> ExternalAppInterface

```
class ClientServerExternalAppsMapper extends ClientServerClientsMapper
begin

    public function ControlToBoundaryExternalAppMessage2ModArqMessages()
    begin
        var externalAppInterfaceInstance = this->archModel-
>createInstance(ExternalAppInterface)
        var distribuidorEventosInstance = this->objectMap-
>getInstance(this->message->source, 'DistribuidorEventos')

        externalAppInterfaceInstance->setName('ExternalAppInterface')
        externalAppInterfaceInstance->setMultiobject(true)
        this->archModel->addInstance(externalAppInterfaceInstance)

        // Agregamos el mensaje a la arquitectura de referencia
        this->refArchCurrentMessage = this->archModel->addMessage(this-
>message->getName(), distribuidorEventosInstance, externalAppInterface)
        this->setRefArchMessageNumber()
        this->refArchPreviousMessage =  this->refArchCurrentMessage
    end
end
```

Tabla 7-61: Método de transformación de mensajes de control a objetos de frontera de tipo ExternalApp para el patrón cliente-server-externalapps.

7.5 Trabajo Relacionado

No encontramos ninguna metodología RIA que transforme modelos de requisitos UML en diseño de arquitectura. En el mejor de los casos hay métodos donde se mapea un modelo de diseño (en un nivel de abstracción más bajo que análisis) a código respetando una arquitectura de acuerdo con algún framework y tecnología. En webML (ver [15]) se transforman modelos de hipertexto (son modelos de navegación

especiales) y de entidad-relación a código respetando arquitectura MVC 2 (ver [21]) especializada para tecnología OpenLazlo (ver [46]) y J2EE (ver [41]).

- En OOH4RIA (ver [3]) se usan de OO-H modelos conceptual y de navegación y además se agregan modelo de presentación estáticos (basados en elementos de interfaz de usuario de GWT) y de orquestación. En OOH4RIA los modelos conceptual, de navegación, de presentación y de orquestación se mapean a GWT (ver [43]) de acuerdo con una arquitectura del lado del cliente basada en patrón message broker. En el patrón "message broker", un administrador de mensajes central recibe notificaciones de eventos y determina los destinos correctos y envía los eventos a los destinatarias. Además, este administrador es único en el sistema (siguiendo los lineamientos del patrón Singleton) lo que elimina la necesidad de los suscriptores y publicadores de tener que guardar una referencia al administrador.

Pero en estos métodos no se produce modelo de arquitectura independiente de tecnología especializado para la aplicación siendo desarrollada como salida y se comienza con modelos a nivel de diseño en lugar de modelos que describen casos de uso.

Cabe observar que los modelos de análisis son más abstractos que los modelos de navegación porque se abstrae de detalles de organización/navegación del hipertexto como:

- clasificación en distintos tipos de elementos de modelado para el hipertexto
- se abstrae de distintos tipos de enlaces de navegación
- se abstrae de detalles sobre cómo está organizado el sitio a muy groso modo

(visiones del sitio, áreas, etc.)

- se abstrae de detalles sobre cómo está organizado el sitio en forma media (p.ej.

páginas, containers de nivel más alto).

Además los modelos de análisis son más abstractos que los modelos de presentación estática RIA (p.ej. RUX - ver [5] -, OOH4RIA - ver [3] -) en lo que se refiere a organización de la UI:

- se abstrae de distintos tipos de elementos de presentación
- se abstrae de detalles sobre cómo está organizado el sitio en forma media (p.ej. páginas, containers de más arriba).
- se abstrae de layout y estilos

Por lo tanto, nosotros al comenzar con modelos de análisis partimos de un nivel de abstracción mayor.

Capítulo 8 - Arquitecturas de Componentes para Algunas Tecnologías RIA actuales

Las aplicaciones RIA proponen una nueva arquitectura alternativa a la utilizada en las aplicaciones web tradicionales. La mayoría de la lógica de la aplicación se implementa en el cliente y el servidor funciona como proveedor de datos y como nexo en la comunicación entre clientes o entre clientes y aplicaciones externas. Los pedidos al servidor son más pequeños y se realizan de manera asincrónica, lo cual permite actualizar solo las partes de la interfaz afectadas por la respuesta al pedido al servidor. Esto también acelera significativamente los tiempos de respuesta.

El los últimos años han surgido nuevas tecnologías que implementan esta arquitectura. Hemos tomado a cuatro de ellas, las consideradas más populares, y hemos definido perfiles que extienden los diagramas de componentes para estas tecnologías. El lector podrá notar que las arquitecturas encontradas para estas tecnologías no están expresadas en la misma notación que la utilizada en este trabajo. A continuación, describimos la arquitectura de cada una de las tecnologías.

8.1 La tecnología GWT

8.1.1 Introducción

Google Web Toolkit (ver [47]) es un framework que permite desarrollar aplicaciones Web sobre el lenguaje Java. Cuando la aplicación esta lista y ya ha sido depurada en Java utilizando toda la maquinaria provista para el desarrollo en este lenguaje, un preprocesador traduce la aplicación a HTML y Javascript.

Las aplicaciones desarrolladas con GWT corren mayormente en el cliente, haciendo llamados al servidor solo cuando es necesario. Estos llamados se realizan mediante RPC. GWT se encarga de generar la mayoría de las clases que implementan los servicios remotos necesarios. Todos los llamados son realizados asincrónicamente. Tiene su propio sistema de clases para programar (no es escribir un código Java cualquiera).

8.1.2 Descripción de la Arquitectura

Para el desarrollo de aplicaciones medianas/grandes, GWT, uno de los enfoques más populares es el que se propone en [43]. Una arquitectura Model-View-Presenter conformada por los siguientes componentes:

- *Display:* Los displays son los componentes gráficos (widgets) con los que el usuario puede interactuar. Son la representación visual de los datos subyacentes de la aplicación. Los displays comunican las interacciones del usuario al sistema a través de eventos como onClick, onMouseDown, onKeyPressed, etc.
- *BusinessObject:* Este componente provee una representación del lado del cliente de los datos de la aplicación. Normalmente representan una estructura de datos bien definida, como una lista o un objeto con una propiedad por cada campo de un registro de la base de datos.
- *Presenter:* Este componente define los manejadores de evento para todos los eventos que ocurren en los displays. Como regla general, tendremos un Presenter por cada Display. Los manejadores de evento se encargan de gestionar el history del browser, de hacer pedidos asincrónicos al servidor cuando es necesario y de realizar la transición entre las deferentes vistas de la aplicación
- *AppController:* Este componente se encarga de manejar la lógica que no esta asociada a ningún display en particular, es decir, la lógica a nivel global de la aplicación.
- *Eventos:* Podemos identificar 2 clases de eventos en GWT. Eventos locales, es decir, que solo tienen relevancia para un Presenter/Display en particular y eventos globales o app-wide, que afectan a más de un Presenter/Display.

8.1.3 Estructura de una aplicación GWT

Para organizar aplicaciones en GWT en [47] se propone que una aplicación GWT tenga la siguiente estructura:

Lado Cliente:

- Interfaz de Usuario:
  - *Widgets:* Heredan de alguna de las clases provistas por la librería de widgets de GWT
  - *Panels:* Heredan de alguna de las clases provistas por la librería de contenedores de GWT.
  - *Host Page:* Un documento HTML que importa todos los módulos traducidos de Java a JavaScript necesarios para el funcionamiento de la aplicación. La importación se realiza mediante la etiqueta
- Estado del Cliente:
  - *Objetos Serializables:* Los objetos serializables heredan de la clase Java *IsSerializable* y son utilizados para representar el estado de la aplicación en el lado cliente. Normalmente proveen únicamente métodos setXXX y getXXX para manipular las propiedades del objeto. Hasta el momento, GWT no provee una forma de asociar el modelo con la interfaz gráfica.

- *Entry Point:* Los puntos de entrada heredan de la clase EntryPoint provista por GWT. Generalmente hay un solo punto de entrada por cada módulo GWT aunque puede haber más. Se encargan de inicializar el módulo. Es el primer código ejecutado por la aplicación, es aquí donde se accede el contenedor principal en el que se cargan los widgets. Se compone de un único método llamado onModuleLoad el cual contiene toda la funcionalidad de inicialización.
- *Interfaz RPC:* Los llamados asincrónicos al servidor respetan el patrón de diseño llamado Command Pattern. Cada comando es definido mediante dos interfaces:
  - *RemoteService:* Estas interfaces heredan de la interfaz provista por GWT llamada RemoteService. Declaran los parámetros y el tipo devuelto por el comando remoto pero no contiene detalles de la implementación del comando. Estas interfaces definen cuales son los comandos que pueden ser ejecutados en el servidor. Es básicamente, una descripción del comando remoto que funciona como nexo entre la función que realiza el llamado asincrónico desde el cliente al servidor y el servlet que implementa la funcionalidad del comando del lado del servidor.
  - *AsyncCallback:* Estas interfaces no heredan de ninguna otra y serán las que, luego de ser traducidas a JavaScript por GWT, implementen el llamado asincrónico al servidor con el objetivo de ejecutar el comando. Si bien no heredan de otra interfaz, están altamente relacionadas con las interfaces RemoteService mencionadas arriba pero agregan a la interfaz del comando una función de callback que será la encargada de manejar la respuesta enviada por el servidor luego de la ejecución del comando.
- *Event Bus:* El Event Bus normalmente es implementado heredando de la clase HandlerManager provista por GWT. Centraliza el control de eventos. Permite disparar eventos y registrar manejadores para responder a ellos. El objetivo es maximizar el desacoplamiento de los distintos elementos de la aplicación haciendo uso de la invocación implícita. Dada una determinada condición, un elemento puede disparar un evento determinado a través del Event Bus. Otros elementos pueden registrar un manejador para ese tipo de eventos, el cual será ejecutado cuando dicho evento sea disparado. Los manejadores de Evento deben implementar la interfaz EventHandler provista por GWT. Los eventos deben heredar de la clase GwtEvent. Finalmente, cabe aclarar que cuando nos referimos a un "elemento de la aplicación", estamos hablando de widgets, objetos serializados, objetos RPC y cualquier otro tipo de entidad que necesite comunicarse con otra.

Lado Servidor:

- *Comandos:* Del lado servidor, encontramos la implementación de cada uno de los comandos disponibles de la aplicación. Los comandos extienden a la clase RemoteServiceServlet provista por GWT que a su vez es descendiente de la

clase HttpServlet provista por Java. Además, deben implementar la interfaz definida por RemoteService correspondiente al comando que implementa. Estas clases están relacionadas directamente con las interfaces RemoteService y AsyncCallback del lado cliente. La clase RemoteServiceServlet es la encargada de serializar y deserializar los datos transmitidos entre el cliente y el servidor. El formato de la serialización es propio de GWT, no se utiliza JSON.

- DAOs: Los DAOs implementan el patrón de diseño DAO (Data Access Object). Proveen las operaciones básicas como insertar, modificar, buscar, eliminar, etc. para manipular la base de datos. Estos deben ser implementados enteramente, es decir, Java no provee ninguna clase base de la cual heredar. Los DAOs se conectan con las base de datos a través de los conectores JDBC provistos por Java. Almacenan los datos en objetos denominados DTOs (Data Transfer Object) cuya función es hacer el mapeo Objeto-Relacional. En resume, el DAO funciona como intermediario entre la base de datos y la aplicación.
- Web Services: A la fecha, GWT no incluye soporte nativo para llamar a servicios externos a la aplicación. Si bien esto puede lograrse, debe ser programado manualmente en JavaScript y luego debe ser inyectado en el documento HTML de la aplicación dinámicamente dentro de un tag SCRIPT.
- JAX-RPC: se encarga de encapsular los llamados a procedimientos remotos (RPC) basados en XML, como por ejemplo SOAP.

La Figura 8-1 describe la arquitectura de una aplicación GWT [44] utilizando un backend implementado utilizando el Framework MVC de Java llamado Spring.

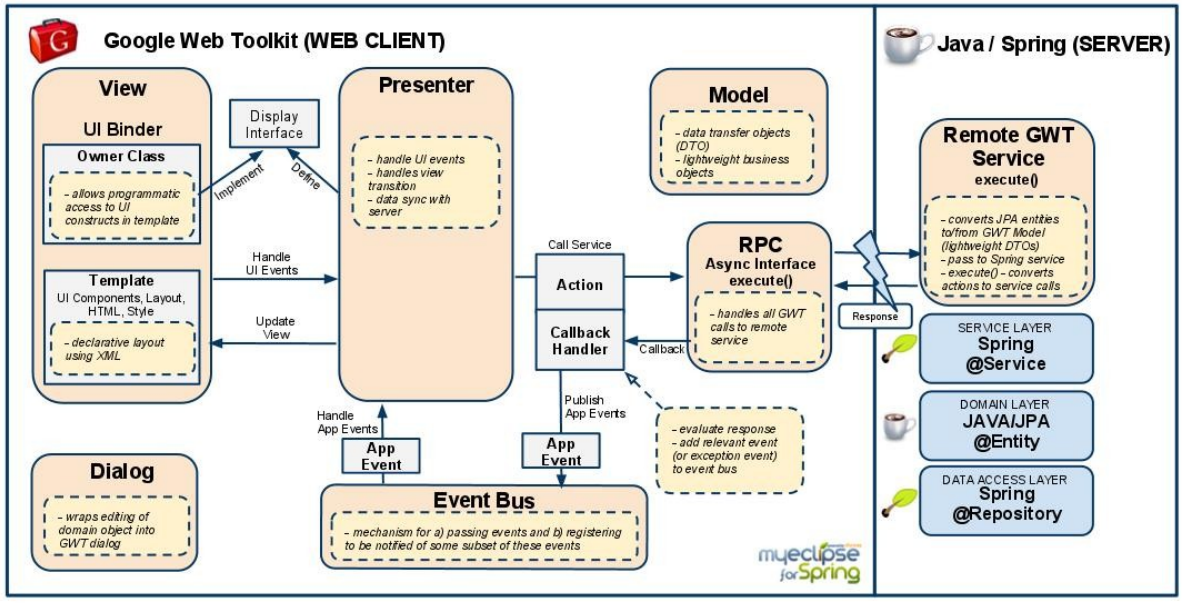

Figura 8-1: Arquitectura GWT

8.1.4 Representación UML de la arquitectura

Para acercar la arquitectura de GWT a este trabajo decidimos construir una

representación de la misma mediante un diagrama de componentes UML. El lector podrá notar que hay elementos que no fueron mencionados en la sección anterior. Estos elementos pertenecen a un conjunto de elementos que agregamos nosotros, en nuestra meta de poder lograr una arquitectura común a todas las tecnologías contempladas. Para empezar, las plataformas que analizamos dan soluciones mayormente para la parte cliente de la aplicación. Esto quiere decir que los paquetes Servidor y AdministradorEventosServidor fueron definidos en este trabajo. Respecto del paquete cliente, todas las componentes pertenecen al enfoque de GWT mencionado en la sección anterior, excepto las componentes EventCoordinator y sus dependencias, que también fueron definidas durante este trabajo para dar tratamiento a los eventos compuestos.

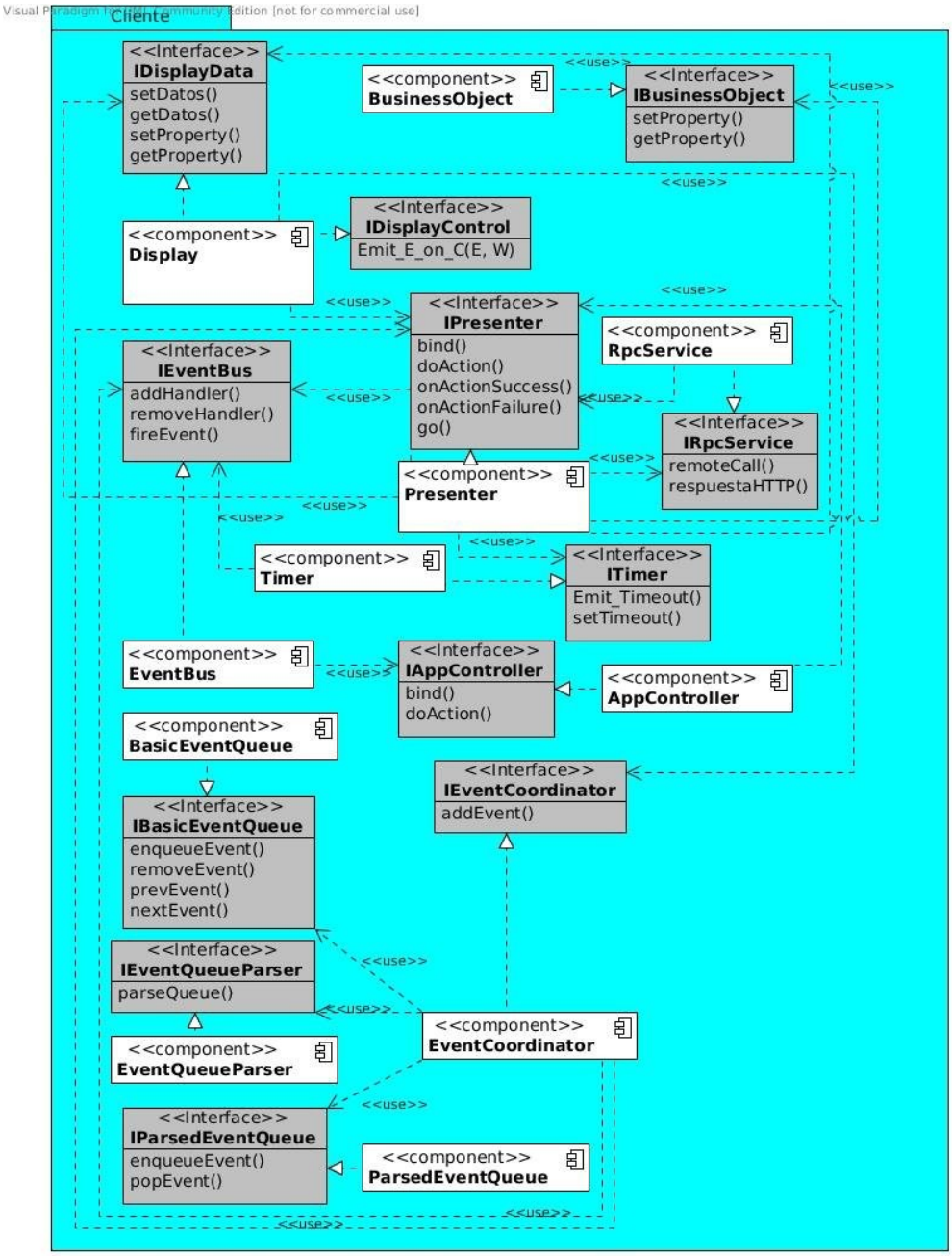

Figura 8-2: Paquete cliente de la Arquitectura GWT.

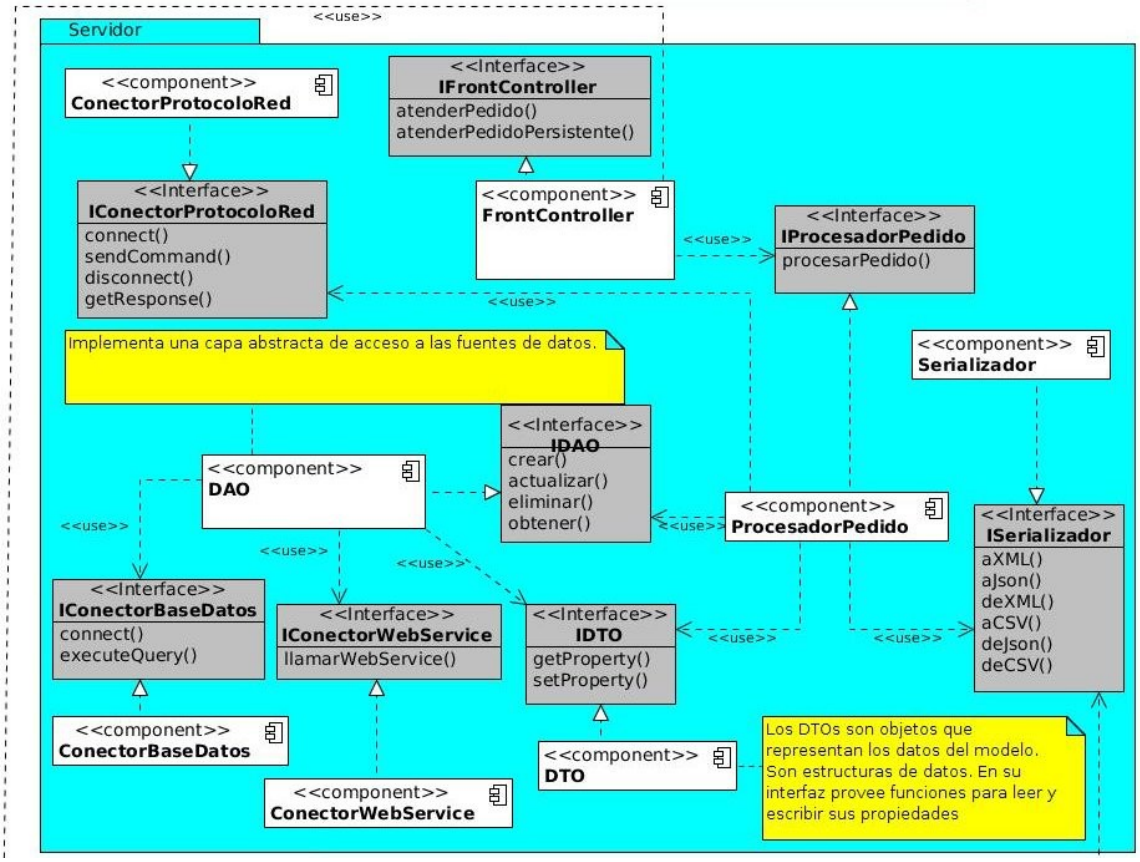

Figura 8-3: Paquete Servidor de la arquitecutra de GWT

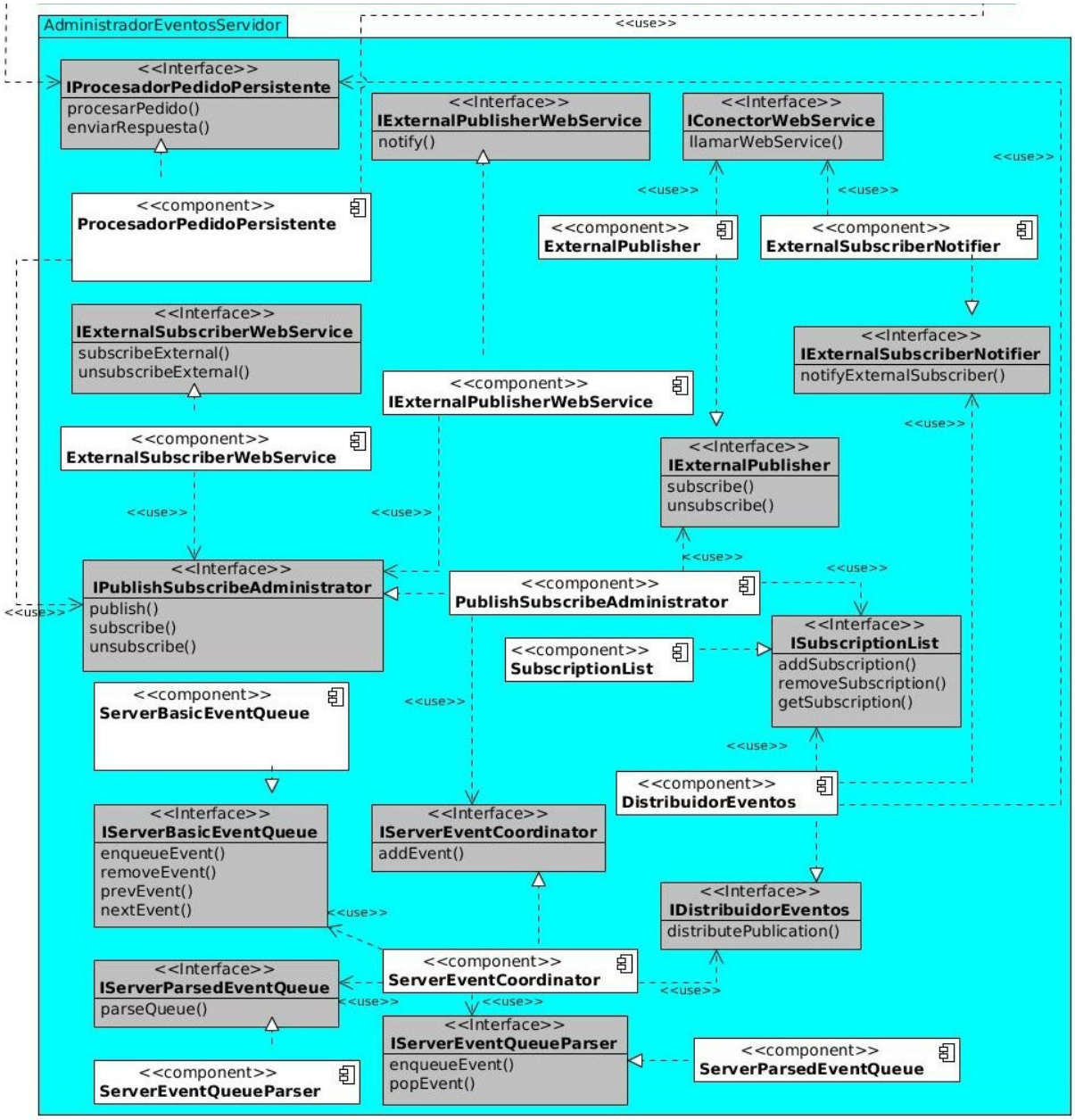

Figura 8-4: Paquete AdministradorEventosServidor de la arquitecutra de GWT

A continuación daremos la descripción de cada una de las componentes de UML que aparecen en el diagrama de componentes.

Nombre	**Display**
Descripción	Se encarga de la representación visual de los datos de la aplicación. Toda la interacción del usuario con la aplicación ocurrirá a través de estos controles. Provee dos interfaces: IDisplayData y IDisplayControl. La primera permite manipular las propiedades visuales y los datos que va a mostrar el Display. La segunda es una interfaz de control, a través de la cual el usuario puede interactuar con el Display. El Display captura los eventos del usuario y se los delega a EventCoodinator cuando el evento puede ser un componente de un evento compuesto, o directamente al presenter en el caso contrario. Existen elementos de la interfaz que son utilizados para definir el layout de la aplicación, que no tienen interfaz de control dado que el usuario no interactúa con ellos directamente(p.e.: los que ofician de contenedores)
Interfaces Provistas	IDisplayControl • Emit_<event>_on_<DisplayName>() - Es llamado como resultado de una interacción <event> (p.e: onClick) del usuario con el Display <DisplayName>. El evento es pasado al EventCoordinator en caso de que pueda formar parte de un evento compuesto, o al Presenter correspondiente en el caso contrario. • IdisplayData operaciones de tipo set para fijar contenidos en el Display. • operaciones de tipo get para obtener contenidos del Display. • setProperty(propertyName, propertyValue) – Setea una propiedad visual del Display (ancho, alto, color, etc.) • getProperty() – Devuelve el valor de una propiedad visual del Display (ancho, alto, color, etc.)
Dependencias	• IPresenter • IEventCoordinator
Comportamiento	• **Presenter:** Cuando un usuario de la aplicación interactúa con el Display que implementa la interfaz IDisplayControl, este captura dicho evento, y determina si puede o no formar parte de un evento compuesto. En caso de que no, se lo delega al Presenter correspondiente llamando a doAction() mediante el event handler registrado por tal Presenter. • **EventCoordinator:** Cuando un usuario de la aplicación interactúa con el Display que implementa la interfaz IDisplayControl, este captura dicho evento, y determina si puede o no formar parte de un evento compuesto. En caso de que si, se lo delega a EventCoordinator llamando a addEvent().
# de instancias	Una o más

Nombre	**Presenter**
Descripción	Cada Display tiene asociado un componente Presenter, el cual implementa los manejadores de eventos para todos los eventos posibles dentro del Display asociado.
Interfaces Provistas	IPresenter • bind() – implementa la asociación entre los eventos generados por el Display a causa de la interacción del usuario y los manejadores de evento declarados en el Presenter • doAction(eventType, eventData) - ejecuta una acción determinada como parte de la respuesta a un evento del Display • onActionSuccess() – es utilizado como función de callback para procesar la respuesta asincrónica exitosa de un pedido al servidor • onActionFailure() – es utilizado como función de callback para procesar la respuesta asincrónica fallida de un pedido al servidor • go(container, initializationData) – Realiza la navegación hacia el display asociado, que será mostrado en el container especificado en el primer parámetro e inicializado con los datos del segundo parámetro.
Dependencias	• IDisplayData, IEventBus. IEventCoordinator. IRpcService. IbusinessObject, AppController. ITimer
Comportamiento	• **DisplayData:** cuando el Presenter necesita obtener o setear datos del Display, lo hace a través de esta interfaz llamando a las funciones para setear y obtener datos del Display. • **EventBus:** cuando lo requiera, el Presenter puede disparar eventos globales a través de EventBus para comunicarse con AppController. Por ejemplo, cuando el procesamiento de un evento en un Presenter produce un cambio de vista. • **EventCoordinator:** cuando EventCoordinator detecte un evento compuesto el cual pueda ser procesado enteramente por un Presenter, EventCoordinator llamará a doAction correspondiente. • **RpcService:** Para enviar pedidos asincrónicos al servidor, Presenter llamará a remoteCall de RpcService. • **BusinessObject:** para acelerar los tiempos de acceso a los datos que necesita para procesar los eventos, Presenter podrá guardar y leer datos de los BusinessObject, los cuales representan los datos de la aplicación del lado del cliente. • **AppController:** los Presenter son activados e inicilizados por AppController a través de la función go() del Presenter, la cual realiza la navegación hacia el Display correspondiente. • **Timer:** como parte del procesamiento de un evento, Presenter puede utilizar el componente Timer para ejecutar tareas temporizadas llamando a setTimeout().
# de instancias	Una o más

Nombre	**RpcService**
Descripción	Este componente es el encargado de hacer los pedidos asincrónicos al servidor.
Interfaces Provistas	IRpcService • remoteCall(remoteAction, requestParameters, onActionSuccess, onActionFailure) – efectúa la comunicación con el servidor. El primer parámetro indica cual es el comando a ejecutar en el servidor, el segundo contienen los parámetros para llamar a tal comando, y además recibe también como parámetro dos funciones de callback que serán llamadas para manejar la respuesta del servidor dependiendo de si fue exitosa o no. • respuestaHTTP(httpResponse) - Es llamado por webserver para indicar que ha finalizado la transferencia de la respuesta y llama a onActionSucces u onActionFailure de Presenter
Dependencias	• IPresenter
Comportamiento	1. **Presenter:** para realizar pedidos al servidor, Presenter llamará a remoteCall de RpcService y le pasará dos funciones de callback, una para manejar las respuesta exitosas y otra para menejar las respuestas fallidas.
# de instancias	Una o más

Nombre	**AppController**
Descripción	Este componente define los manejadores de evento a nivel global de la aplicación, es decir, aquellos que realizan cambios de vista o comunican diferentes presenters entre sí.
Interfaces Provistas	IAppController • bind() – inicializa la asociación de eventos disparados a través de EventBus con sus correspondientes manejadores. • doAction() – implementa la respuesta a un evento global particular.
Dependencias	• IEventBus • IPresenter
Comportamiento	• **EventBus:** AppController utiliza EventBus para registrar los manejadores de los eventos que serán disparados a través de él llamando a addHandler. Cuando se dispara un evento global llamando a fireEvent de EventBus, este llama al manejador correspondiente doAction de AppController. • **Presenter:** cuando se navega hacia un nuevo display como resultado de un evento global, AppController instancia e inicializa el Presenter correspondiente a la nueva vista llamando a bind() y a go() de Presenter.
# de instancias	Una o más

Nombre	**BusinessObject**
Descripción	Este componente unifica el formato de los datos de la aplicación independientemente de la fuente de datos. Ya sea que el servidor envíe sus datos en formato XML, CSV, JSON o cualquier otro, siempre son convertidos en BusinessObject, y serán tratados como tales a lo largo de toda la aplicación del lado del cliente.
Interfaces Provistas	IBusinessObject • getProperty(propertyName) – devuelve el valor de la propiedad propertyName del BusinessObject • setProperty(propertyName, value) – setea el valor de propertyName a value.
Dependencias	• IPresenter
Comportamiento	• **Presenter:** cuando hay un cambio en los datos de la aplicación en la fuente de persistencia, Presenter llama a setProperty de BusinessObject para que los datos del lado del cliente y del servidor estén actualizados. Si para realizar el procesamiento de un evento, Presenter necesita datos que ya están presentes del lado del cliente en un BusinessObject, puede llamar a getProperty para obtenerlos.
# de instancias	Una o más

Nombre	**EventBus**
Descripción	Este componente permite a los Presenter emitir notificaciones (eventos) globales para delegar el procesamiento.
Interfaces Provistas	IEventBus • addHandler(eventType, eventHandler) – asocia los eventos de tipo eventType con el manejador eventHandler, de forma que cuando ocurra tal evento, se ejecute el manejador. • removeHandler(eventType, eventHandler) – elimina la asociación entre los eventos de tipo eventType y eventHandler • fireEvent(eventType) – dispara un evento lo cual produce la ejecución de todos los manejadores asociados con addHandler
Dependencias	• IAppController • IPresenter • IEventCoordinator • ITimer
Comportamiento	• **AppController:** cuando se dispara un evento global llamando a fireEvent de EventBus, este llama al manejador de evento doAction asociado en AppController. • **Presenter:** para delegar el procesamiento de tareas globales, es decir, cuyo procesamiento escapan a la responsabilidad del Presenter, este puede disparar eventos globales llamando a fireEvent de EventBus. • **EventCoodinator:** cuando EventCoordinator detecta un evento compuesto cuyo procesamiento no es responsabilidad de ningún presenter en particular, se considera que es un evento global, y por tanto será disparado llamando a fireEvent de EventBus. • **Timer:** en caso de que haya un Timer activo y este genere un evento timeout, este evento será considerado como global, y por tanto será notificado a través de fireEvent de EventBus.
# de instancias	Una o más

Nombre	**Timer**
Descripción	Permite establecer tareas temporizadas, que se ejecutarán después de cierto intervalo de tiempo.
Interfaces Provistas	ITimer • setTiemout(time) – Permite iniciar un temporizador • Emit_Timout() - Dispara un evento avisando que el tiempo seteado ha transcurrido
Dependencias	• IPresenter • IEventBus
Comportamiento	• **Presenter:** cuando un Presenter necesita de un temporizador como parte del procesamiento de un evento, llamará a setTimeout() del componente Timer. • **EventBus:** Cuando el tiempo seteado para el timer mediante setTimeout() ha transcurrido, el Timer avisa de tal evento llamando a fireEvent() de EventBus.
# de instancias	Una o más

8.2 La tecnología FLEX

8.2.1 Introducción

Flex (ver [48]) Utiliza MXML (lenguaje de etiquetas interpretado parecido a JSP, los tags se traducen a ActionScript, ActionScript se compila a un byte code, que es enviado al navegador Flash que es el plugin que hizo el pedido), un lenguaje declarativo basado en XML, para describir el aspecto y el comportamiento de la interfaz de usuario (que es tan rica visualmente hablando como en las aplicaciones desktop). Los programas desarrollados en Flex se ejecutan del lado del cliente sobre un entorno de ejecución integrado al navegador como plugin.

Flex ofrece RPC, comunicación asincrónica y comunicación en tiempo real (chat, video, etc.). También ofrece una plataforma llamada AIR que permite ejecutar las aplicaciones como aplicaciones de escritorio.

8.2.2 Descripción de la Arquitectura

Cuando construimos aplicaciones con Flex, describimos la interfaz de usuario usando componentes llamados contenedores y controles. Un contenedor es una región rectangular de la pantalla que contiene controles y otros contenedores. Ejemplos de contenedores son los Formularios, utilizados para el ingreso de datos. Los controles son elementos del Formulario tales como los botones, los campos de texto, los checkboxes, etc.

Aunque se puede considerar a Flex como parte de la vista en la arquitectura MVC, se puede utilizar para implementar dicha arquitectura enteramente en el cliente. Una aplicación Flex tiene sus propios componentes para la vista que definen la interfaz del

usuario, componentes para el modelo que representa los datos, y componentes para el controlador, responsable de la comunicación con el servidor.

8.2.3 Estructura de una aplicación FLEX

Lado cliente:

Una aplicación Flex inicia cuando el usuario escribe la URL del sitio donde esta alojada la aplicación. El servidor web envía el documento html inicial el cual, mediante la etiqueta *embed* y *object,* carga la aplicación utilizando el plugin de Flash Player en el navegador.

El lado cliente de la aplicación se basa en el patrón de diseño MVC, tomando como base el framework Cairngorm [49], el más popular entre los desarrolladores FLEX.

La Vista, es decir, la interfaz de usuario, se define utilizando MXML, un lenguaje basado en XML provisto por Flex. Los documentos MXML se componen básicamente de dos tipos de elementos: Contenedores y Controles. Lo contenedores son cajas rectangulares que contienen a otros elementos e incluso a otros contenedores. Se utilizan para definir la estructura visual del sistema. Los Controles son los elementos a través de los cuales el usuario interactúa con la aplicación.

El Modelo es simplemente una representación de los datos mediante clases AS3. Generalmente no contienen demasiada lógica sino que definen propiedades que mapean al modelo de datos del lado del servidor y además proveen métodos para acceder y escribir estas propiedades, según sea necesario.

El Controlador responde al patrón de diseño llamado Command (Comando) y consiste de un Controlador Frontal (Front Controller) y de un Command por cada una de las acciones disponibles en el sistema.

Cuando el usuario interactúa con un Control de la interfaz de usuario, este genera un evento que esta asociado a un comando. Este evento es capturado por el Controlador Frontal que es el encargado de instanciar el Command que implementa dicho comando. El Command será el responsable de actualizar el modelo acorde a las acciones que ejecuta. Los Command tienen también a su cargo la comunicación con el servidor web cuando sea necesario. Para esto utiliza el protocolo http(s), mediante el cual envía la información. La información intercambiada entre el cliente y el servidor son objetos ActionScript3 en un formato de transferencia llamado AMF.

La arquitectura que se presenta en este trabajo esta basada en Cairngorm (ver [49]), uno de los frameworks más populares para el desarrollo de aplicaciones en Flex. Este framework no hace más que guiar al desarrollador hacia la utilización de "buenas prácticas" de programación como desacoplamiento, modularización, división de responsabilidades, etc. Uno de los componentes del framework llamado ModelLocator fue removido por considerar que no aporta la suficiente riqueza a la arquitectura como para considerarlo relevante.

Lado Servidor:

Los Command Objects del lado del cliente realizan pedidos asincrónicos al servidor

cuando la funcionalidad que implementan lo requiere. Para esto, utilizan un RPCObject que encapsula el llamado al servicio provisto por el servidor. El RPCObject le envía un objeto binario AS3 que es recibido por el Gateway AMF. Cada lenguaje del lado servidor (Java, PHP, etc) tiene una implementación propia del Gateway AMF. Su misión es traducir los objetos enviados en formato binario AS3 a algo comprensible y utilizable por el lenguaje utilizado del lado servidor. Esto es recibido por el controlador frontal, que instancia el comando correspondiente al llamado realizado por el cliente, y utilizando los datos recibidos, lleva a cabo la acción sobre el modelo.

Estados y Transiciones:

Los estados son estructuras que contienen cambios sobre la presentación visual de uno o más componentes. Dichos cambios son aplicados cuando el estado se vuelve activo. La activación y desactivación de los estados depende de la interacción del usuario. Pueden definirse mediante Action Script o MXML. Cada uno tiene un nombre mediante el cual puede ser referenciado. Todas las aplicaciones Flex tienen al menos un estado por defecto llamado *base*. Los estados se definen con el objeto de manejar la estructura visual de la aplicación basándose en la actividad que realiza el usuario.

Las transiciones son grupos de efectos visuales. Su tarea es cambiar la aplicación de un estado a otro y hacer que los cambios de estado se vean más atractivos. Pueden ser definidas en MXML o en Action Script.

La Figura 8-5 describe la arquitectura de una aplicación FLEX conectada con un webserver escrito en PHP [45].

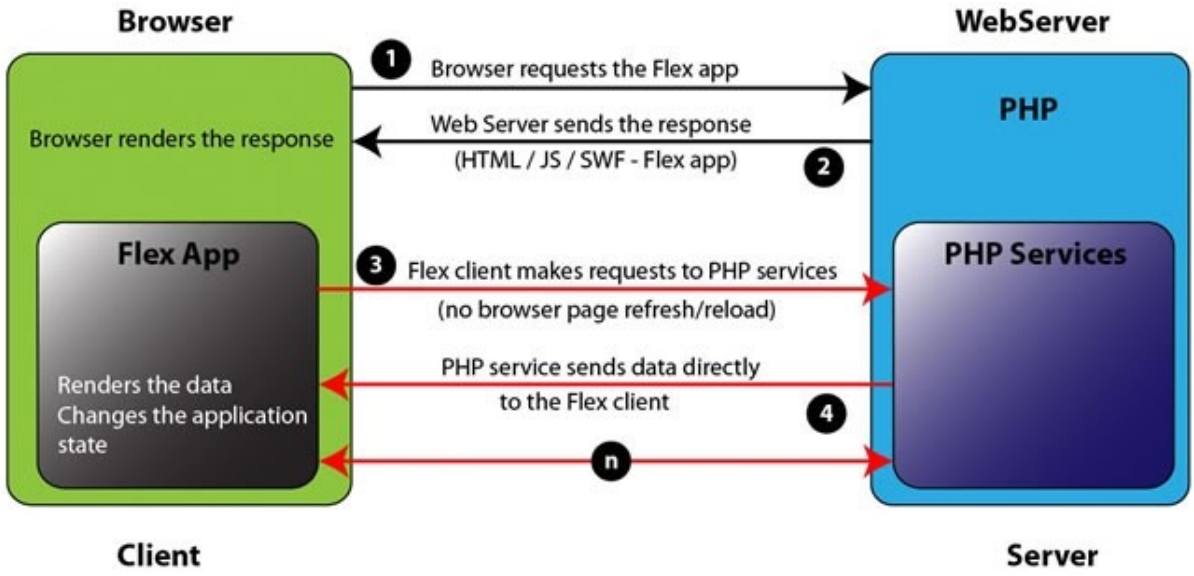

Figura 8-5: Arquitectura Flex

8.2.4 Representación UML de la arquitectura

Para este trabajo, decidimos construir una representación UML de la arquitectura de FLEX que se asemeje a lo tratado en los capítulos anteriores. El lector podrá notar que hay elementos que no fueron mencionados en la sección anterior. Estos elementos

pertenecen a un conjunto de elementos que agregamos nosotros, en nuestra meta de poder lograr una arquitectura común a todas las tecnologías contempladas. Para empezar, las plataformas que analizamos dan soluciones mayormente para la parte cliente de la aplicación. Esto quiere decir que los paquetes Servidor y AdministradorEventosServidor fueron definidos en este trabajo. Respecto del paquete cliente, todas las componentes pertenecen al enfoque de FLEX mencionado en la sección anterior, excepto las componentes EventCoordinator y sus dependencias, que también fueron definidas durante este trabajo para dar tratamiento a los eventos compuestos.

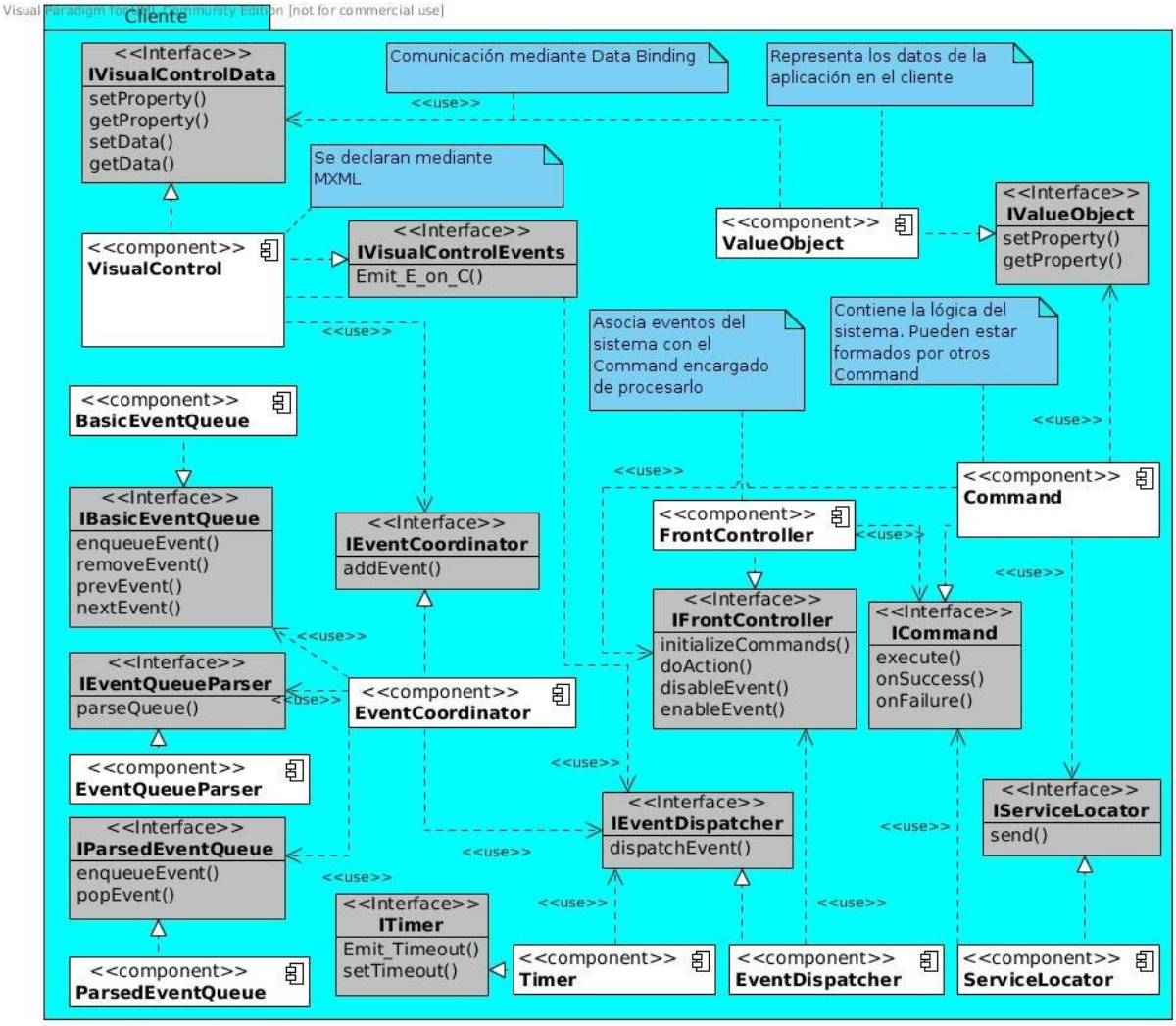

Figura 8-6: Paquete cliente de la arquitectura FLEX

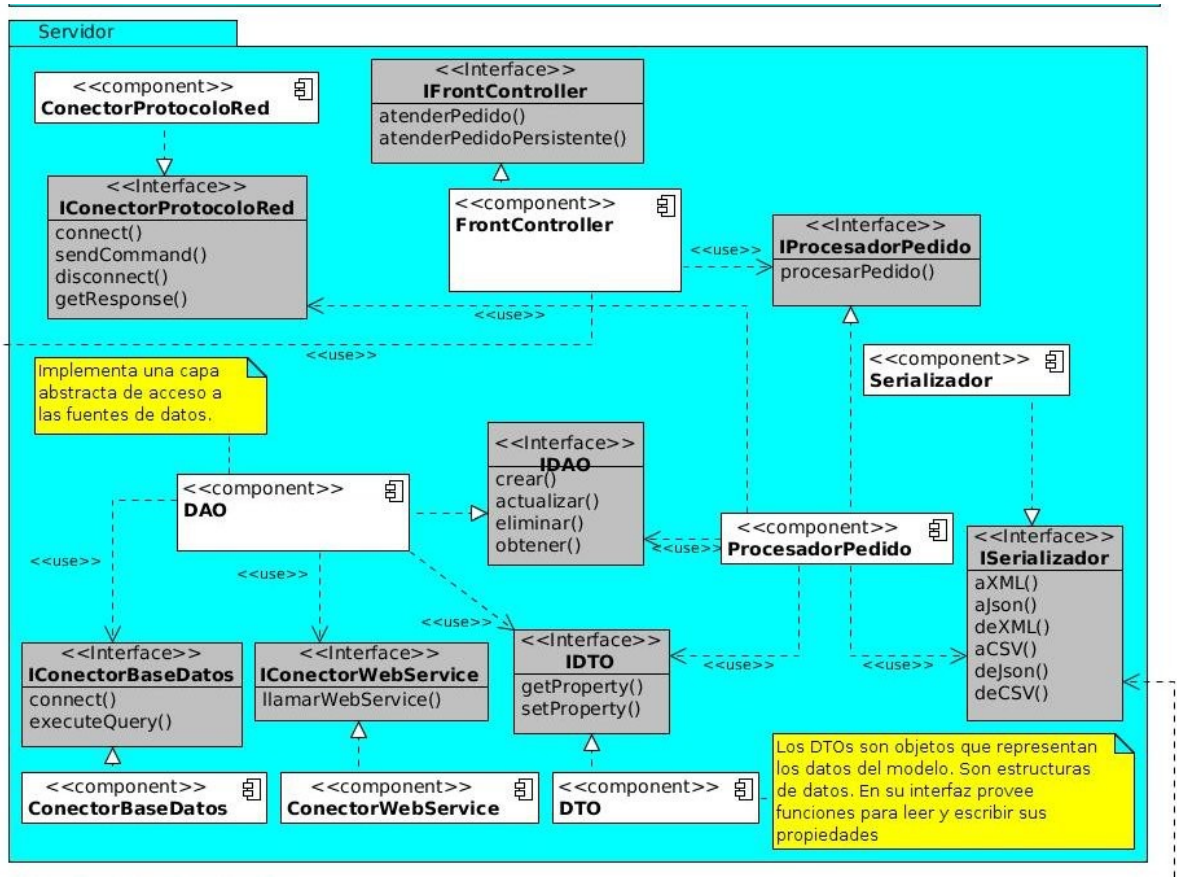

Figura 8-7: Paquere servidor de la arquitectura FLEX

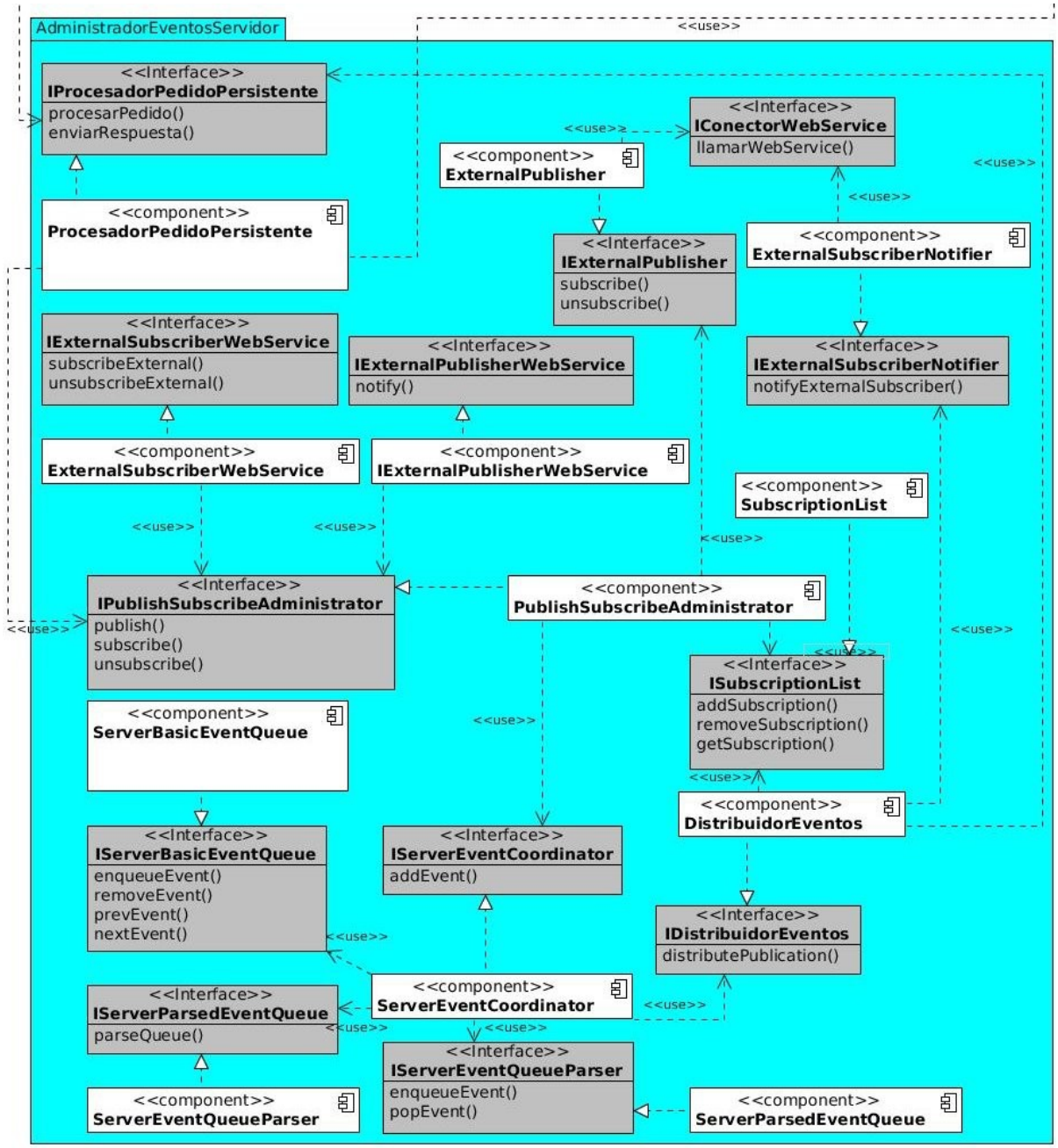

Figura 8-8: Paquete AdministradorEventosServidor de la arquitectura FLEX

A continuación damos la especificación de cada una de las componentes que aparecen en el diagrama de componentes de las Figuras 8-6, 8-7 y 8-8.

Nombre	**VisualControl**
Descripción	Se encarga de la representación visual de los datos de la aplicación. Toda la interacción del usuario con la aplicación ocurrirá a través de estos controles. Provee dos interfaces: IVisualControlData y IVisualControlEvents. La primera permite manipular las propiedades visuales y los datos que va a mostrar el control. La segunda es una interfaz de control, a través de la cual el usuario puede interactuar con el VisualControl. El VisualControl captura los eventos del usuario y se los delega a EventCoodinator. Existen elementos de la interfaz que son utilizados para definir el layout de la aplicación, que no tienen interfaz de control dado que el usuario no interactúa con ellos directamente(p.e.: los que ofician de contenedores)
Interfaces Provistas	IVisualControlEvents • Emit_<event>_on_<VisualControlName>() - Es llamado como resultado de una interacción <event> (p.e: onClick) del usuario con el VisualControl <VisualControlName>. • IVisualControlData • operaciones de tipo set para fijar contenidos en el VisualControl. • operaciones de tipo get para obtener contenidos del VisualControl. • setProperty(propertyName, propertyValue) – Setea una propiedad visual del VisualControl (ancho, alto, color, etc.) • getProperty() – Devuelve el valor de una propiedad visual del VisualControl (ancho, alto, color, etc.)
Dependencias	• IEventCoordinator, IEventDispatcher, IValueObject
Comportamiento	• **EventCoordinator:** Cuando un usuario de la aplicación interactúa con el VisualControl que implementa la interfaz IVisualControlEvents, este captura dicho evento, crea un CustomEvent con los datos relevantes del evento y si es un posible candidato a formar parte de un evento compuesto, se lo delega a EventCoordinator llamando a addEvent(). • **EventDispatcher:** Cuando un usuario de la aplicación interactúa con el VisualControl que implementa la interfaz IVisualControlEvents, este captura dicho evento, crea un CustomEvent con los datos relevantes del evento y si es NO un posible candidato a formar parte de un evento compuesto, se lo delega a EventCoordinator llamando a addEvent(). • **ValueObject:** A través del sistema de data binding provisto por Flex, se pueden asociar las propiedades de un ValueObject con las propiedades de un VisualControl, de manera que, cuando un Command modifique alguna propiedad de un ValueObject, automáticamente se refresca

	el VisualControl asociado para reflejar dicho cambio.
# de instancias	Una o más

Nombre	**CustomEvent**
Descripción	Representan los datos relevantes de un evento, es decir, contienen todos los datos que necesita o puede necesitar un Command para procesar el evento. Los CustomEvent tienen un tipo que los identifica ante el FrontController y un conjunto de propiedades las cuales almacenan información sobre el evento.
Interfaces Provistas	ICustomEvent • getType() – Permite obtener el tipo del evento, el cual es utlizado por el FrontController para identificar el evento y delegarlo al Command correspondiente. • setType(eventType) - Permite setear el tipo del evento, el cual es utlizado por el FrontController para identificar el evento y delegarlo al Command correspondiente. • getData() - Permite obtener los detalles del evento. • setData(eventData) – Permite setear los datos del evento
Dependencias	• IVisualControl • IFrontController • IEventCoordinator • ICommand
Comportamiento	• **IVisualControl:** Cuando un usuario interactúa con un VisualControl, este crea un CustomEvent, setea el tipo y lo datos llamando a setType y setData y se lo pasa a addEvent de EventCoordinator en caso de que el evento sea candidato a formar parte de un evento compuesto o a doAction de FrontController en caso de que el evento sea un evento simple. • **FrontController:** llama a getType de CustomEvent y con este dato, sabe que Command es el que debe instanciar para delegarle el procesamiento del mismo. • **EventCoordinator:** llama a getType y getData de CustomEvent para determinar si el evento forma parte de un evento compuesto o no. • **Command:** llama a getData para obtener los datos relevantes del evento y realizar el procesamiento del mismo.
# de instancias	Una o más

Nombre	**EventDispatcher**
Descripción	Este componente es provisto por el framework. Su función es la de permitir la comunicación mediante eventos entre los VisualControl y el FrontController.
Interfaces Provistas	IEventDispatcher • dispatchEvent(customEvent) – llama a doAction_A_on_E correspondiente, sólo si customEvent esta registrado en FrontController.
Dependencias	• IVisualControl • IFrontController • IEventCoordinator
Comportamiento	• **IVisualControl:** Cuando un usuario interactúa con un VisualControl, este crea un CustomEvent y lo notifica a través de dispatchEvent de EventDispatcher. • **FrontController:** EventDispatcher notifica a FrontController la ocurrencia de un evento y llama al doAction_A_on_E correspondiente, sólo si el tipo de evento ocurrido ha sido registrado en FrontController durante la inicialización. • **EventCoordinator:** Cuando EventCoordinator detecta un evento compuesto, este crea un CustomEvent y lo notifica a través de dispatchEvent de EventDispatcher.
# de instancias	Una o más

Nombre	**FrontController**
Descripción	Este componente asocia los tipos de eventos definidos en la aplicación con los Command que los atienden.
Interfaces Provistas	IFrontController • initializeCommands() – asocia los tipos de eventos con los Command que los atienden. Es llamado solamente una vez. • doAction_A_on_E() - cuando ocurre el evento E, instancia el Command que se encarga de la tarea A y le delega el procesamiento del evento. • disableEvent(eventType) - permite desactivar un evento
Dependencias	• IEventDispatcher • ICommand
Comportamiento	• **EventDispatcher:** Cuando se notifica de un evento, EventDispatcher llama a doAction_A_on_E de FrontController, sólo si el tipo de evento ocurrido ha sido registrado durante initializeCommands. • **Command:** dependiendo del tipo de evento ocurrido, FrontController instancia el Command correspondiente según fue definido durante initializeCommands y le delega el procesamiento del mismo llamando a execute de Command.
# de inst.	Una o más

Nombre	**Command**
Descripción	Este componente es el encargado de dar respuesta a los eventos ocurridos en la aplicación. Hace pedidos al servido a través de ServiceLocator en caso de necesitarlo, y actualiza los ValueObjects.
Interfaces Provistas	ICommand • execute(customEvent) – contiene toda la lógica necesaria para el procesamiento del evento al cual está asociado. • onSuccess() - esta función es llamada como callback cuando hay una respuesta existosa de un pedido al servidor. • OnFailure() - esta función es llamada como callback cuando hay una respuesta fallida de un pedido al servidor.
Dependencias	• IFrontController • IServiceLocator • IValueObject
Comportamiento	• **FrontController:** Trás la ocurrencia de un evento, FrontController instancia el Command correpondiente según fue definido durate initializeCommands y llama a execute de Command, pasándole el evento como parámetro. • **ServiceLocator:** Si el Command necesita enviar un pedido al servidor, llama a send de ServiceLocator, le pasa onSuccess y onFailure como funciones de callback para manejar la respuesta del servidor. • **ValueObject:** Cuando es necesaria una actualización de los datos del lado del cliente para que coincidan con los datos del lado servidor, es Command el encargado de realizar dicha actualización.
# de instancias	Una o más

Nombre	**ValueObject**
Descripción	Este componente unifica el formato de los datos de la aplicación independientemente de la fuente de datos. Ya sea que el servidor envíe sus datos en formato XML, CSV, JSON o cualquier otro, siempre son convertidos en ValueObjects, y serán tratados como tales a lo largo de toda la aplicación del lado del cliente.
Interfaces Provistas	IValueObject • getProperty(propertyName) – devuelve el valor de la propiedad propertyName del ValueObject • setProperty(propertyName, value) – setea el valor de propertyName a value.
Dependencias	• ICommand • IVisualControl
Comportamiento	• **Command:** Cuando es necesaria una actualización de los datos del lado del cliente para que coincidan con los datos del lado servidor, es Command el encargado de realizar dicha actualización sobre los ValueObject. • **VisualControl:** Los ValueObject se comunican con los VisualControl a través del sistema de Data Binding provisto por Flex. Al hacer data binding, los VisualControl son notificados ante cualquier cambio de estado en los ValueObject a los cuales están asociados, y se actualizan para reflejar dicho cambio.
# de instancias	Una o más

Nombre	**ServiceLocator**
Descripción	Establece las conexiones remotas con el Servidor y provee una abstracción para el método de comunicación (por URL, SOAP, etc.)
Interfaces Provistas	IServiceLocator • send(request, onSuccess, onFailure) – enviar un pedido a un servicio remoto. Dependiendo de si la respuesta es exitosa o no, llama a la función de callback onSuccess u onFailure.
Dependencias	• ICommand
Comportamiento	• **Command:** Si el Command necesita enviar un pedido al servidor, llama a send de ServiceLocator, le pasa onSuccess y onFailure como funciones de callback para manejar la respuesta del servidor.
# de instancias	Una o más

Nombre	**Timer**
Descripción	Permite establecer tareas temporizadas, que se ejecutarán después de cierto intervalo de timepo.
Interfaces Provistas	ITimer • setTiemout(time) – Permite iniciar un temporizador • Emit_Timout() - Dispara un evento avisando que el tiempo seteado ha transcurrido
Dependencias	• ICommand • IEventDispatcher
Comportamiento	• **Command:** cuando un Command necesita de un temporizador como parte del procesamiento de un evento, llamará a setTimeout() del componente Timer. • **EventDispatcher:** Cuando el tiempo seteado para el timer mediante setTimeout() ha transcurrido, el Timer avisa de tal evento llamando a dispatchEvent() de EventDispatcher.
# de instancias	Una o más

8.3 La tecnología OpenLaszlo

8.3.1 Introducción

OpenLaszlo (ver [46]) se compone de un lenguaje declarativo basado en XML llamado LZX utilizado para escribir aplicaciones cliente y un servidor de aplicaciones que compila este lenguaje, llamado servido de presentación. El resultado de la compilación es código bytecode para Flash o código HTML+JavaScript.

El servidor de presentación cumple dos diferentes roles dentro de un sistema:

- Compila aplicaciones LZX a películas flash cuando las aplicaciones son solicitadas por el navegador.
- Mantiene un cache de programas LZX compilados, y solo los recompila cuando estos han sido modificados.

8.3.2 Descripción de la Arquitectura

El lado servidor de OpenLaszlo es una aplicación Java que se ejecuta en un contenedor de Servlet J2EE. Este puede comunicarse con otras fuentes de datos usando una variedad de protocolos. Las aplicaciones son codificadas mediante LZX, compiladas por el Servidor OpenLaszlo y entregadas al cliente como byte-code ejecutable por el plugin de flash o como JavaScript para ser ejecutada directamente por el navegador.

8.3.3 Estructura de una aplicación OpenLazlo

Lado Cliente:

La arquitectura del lado cliente de OpenLaszlo se basa en una librería llamada Laszlo Foundation Class (LFC) – ver [46] -, que es compilada en todas las aplicaciones OpenLaszlo que provee servicios en tiempo de ejecución (como por ejemplo, timers) y un renderizador de presentación para mostrar gráficos en 2D y reproducir sonido. Ninguna de las clases incluidas se basan en servicios de Flash ni usan el modelo de objetos de Flash. Cuando se compila una aplicación a formato SWF, Flash Player solo es utilizado como motor de renderización.

Cuando la aplicación LZX está en ejecución, incluso si está ociosa, está manteniendo una conexión con el servidor, y todos los elementos necesarios para ejecutar la aplicación ya han sido descargados.

Existen 4 componentes primarios dentro de la LFC:

- Event System
- Data Loador/Binder
- Layout and Animation System
- OpenLaszlo Service System

Event System:

Su trabajo consiste en capturar los eventos generados por la interacción de usuario con el sistema y los generados por la recepción de datos desde el servidor.

Entre los tipos de eventos encontramos:

- Constraints: Establecen relaciones entre los componentes de la interfaz de usuario, es decir, como debe comportarse un componente en relación al cambio de estado de otro componente. Por componente no solo nos referimos a los provistos por OpenLaszlo, sino también a la interacción del Mouse (clicks, posición del puntero, etc) y del teclado.
- Eventos built-in (OpenLaszlo los llama Implicit Events): Todos los atributos de cada tag LZX tienen asociados un evento que se dispara cuando su valor cambia. Por ejemplo, cuando el atributo “height” de un tag cambia, dispara un evento “onheight” al cual puede asociársele un manejador. Como regla general, para cada atributo de nombre X existe un evento onX que es disparado cuando el valor de X es modificado. Adicionalmente, encontramos los clásicos eventos de interfaz de usuario como onclick, onblur, onfocus, etc.
- Eventos definidos por el programador: Además de contar con los eventos arriba mencionados, el programador puede definir sus propios eventos y dispararlos cuando lo crea conveniente. Los eventos son declarados utilizando el tag <event>. Luego de la declaración, debe incluirse un método que se encargue de disparar el evento llamando a sendEvent( ). No deben dispararse eventos para los cuales no existe un manejador, aunque si pueden declararse. Un manejador de evento se declara utilizando el tag <handler>, el cual debe tener el mismo nombre que el evento que controla. Un evento puede tener cero

o más de manejadores. El manejador puede ser definido utilizando JavaScript dentro del tag handler o puede llamar a un método.

Data Loader/Binder:

Su función es asociar los datos recibidos del servidor con el componente de interfaz de usuario que se encarga de mostrar dichos datos. Los datos son representados utilizando XML y se llaman datasets. Hay tres formas de incluir un dataset en una aplicación OpenLaszlo, todas ellas variaciones del tag LZX <dataset ... >

1. Embebida: El código XML del dataset se sitúa enteramente en el archivo LZX en el cual es accedido.
2. Incluido: OpenLaszlo provee un mecanismo para incluir datasets ubicados en archivos XML externos, muy parecido a como uno incluiría un archivo JavaScript o un CSS en HTML.
3. HTTP Data: El dataset es cargado mediante un request HTTP.

Los primeros dos son utilizados para datasets estáticos. El último se utiliza para cargar datasets construidos dinámicamente en el servidor.

Layout and Animation System:

Permite posicionar los elementos de la interfaz de usuario y agregar animaciones a estos cuando su estado cambia.

OpenLaszlo Service System:

Permite acceder a servicios provistos por el sistema operativo subyacente como por ejemplo timers, sonido y ventanas modales.

Lado Servidor:

El Servidor OpenLaszlo (ver [46]) se ejecuta dentro de un servidor de aplicaciones J2EE estándar o en un Contenedor de Servlet de Java. Esta compuesto por 5 subsistemas:

- Interface Compiler
- Media Transcoder
- Data Manager
- Persistent Connection Manager
- Cache

Interface Compiler:

El Interface Compiler consiste en un compilador de etiquetas LZX y un compilador de scripts de JavaScript. Adicionalmente, invoca al Media Compiler y al Data Manager para compilar los recursos multimedia y las fuentes de datos externos.

El compilador de etiquetas LZX y el de scripts convierte las etiquetas LZX y el código JavaScript en bytecode ejecutable (SWF) para la plataforma Flash Player.

Este código se almacena en el cache, desde donde luego es enviado al cliente.

El Media Transcoder convierte un amplio rango de recursos multimedia en un único formato para ser mostrados en la plataforma de ejecución del cliente. Los recursos soportados son: JPEG; GIF, PNG, MP3, TrueType y SWF (animaciones Flash).

Data Manager:

Se compone de un compilador de datos que convierte todos los datos a un formato binario comprimido utilizable por las aplicaciones OpenLaszlo y una serie de conectores que permiten a las aplicaciones obtener datos mediante XML/HTTP. Así, las aplicaciones pueden conectarse con bases de datos, XML Web Services e incluso otros servidores de aplicaciones.

Persistent Connection Manager:

Controla la autenticación y el intercambio de mensajes en tiempo real para las aplicaciones OpenLaszlo que lo requieren. (¿Pull de datos?). Sólo esta disponible si la aplicación es compilada como SWF.

Cache:

Contiene la versión compilada más reciente de una aplicación. La primera vez que la aplicación es accedida, es compilada y el SWF resultante es enviado al cliente. Una copia es guardada en el cache del servidor, para evitar tener que recompilarlo en futuros accesos.

A través de un conector HTTP, el Laszlo Presentation Server se conecta al servidor de aplicaciones, el cual puede estar implementado en Java, Php, o cualquier otro lenguaje que soporte el protocolo HTTP. El intercambio de datos se realiza en formato XML. Cuando el archivo solicitado por el cliente es un documento LZX, este es compilado a bytcode de Flash antes de ser enviado.

La Figura 8-9 describe la arquitectura de una aplicación OpenLaszlo [46].

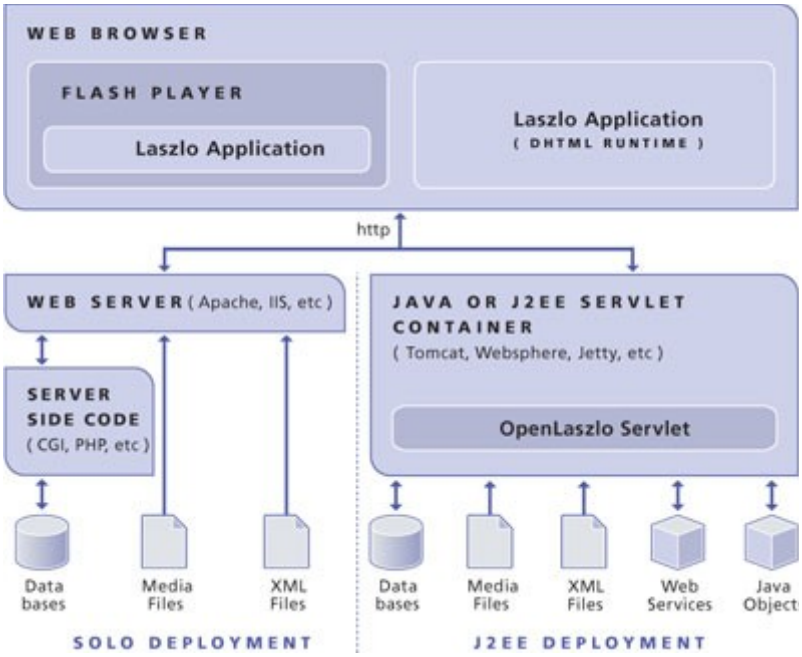

Figura 8-9: Arquitectura OpenLaszlo

8.3.4 Representación UML de la arquitectura

A continuación presentamos una representación de la arquitectura de OpenLaszlo mediante un diagrama de componentes UML, con el objetivo de acercarlo a lo tratado en los capítulos anteriores. El lector podrá notar que hay elementos que no fueron mencionados en la sección anterior. Estos elementos pertenecen a un conjunto de elementos que agregamos nosotros, en nuestra meta de poder lograr una arquitectura común a todas las tecnologías contempladas. Para empezar, las plataformas que analizamos dan soluciones mayormente para la parte cliente de la aplicación. Esto quiere decir que los paquetes Servidor y AdministradorEventosServidor fueron definidos en este trabajo. Respecto del paquete cliente, todas las componentes pertenecen al enfoque de OpenLaszlo mencionado en la sección anterior, excepto las componentes EventCoordinator y sus dependencias, que también fueron definidas durante este trabajo para dar tratamiento a los eventos compuestos.

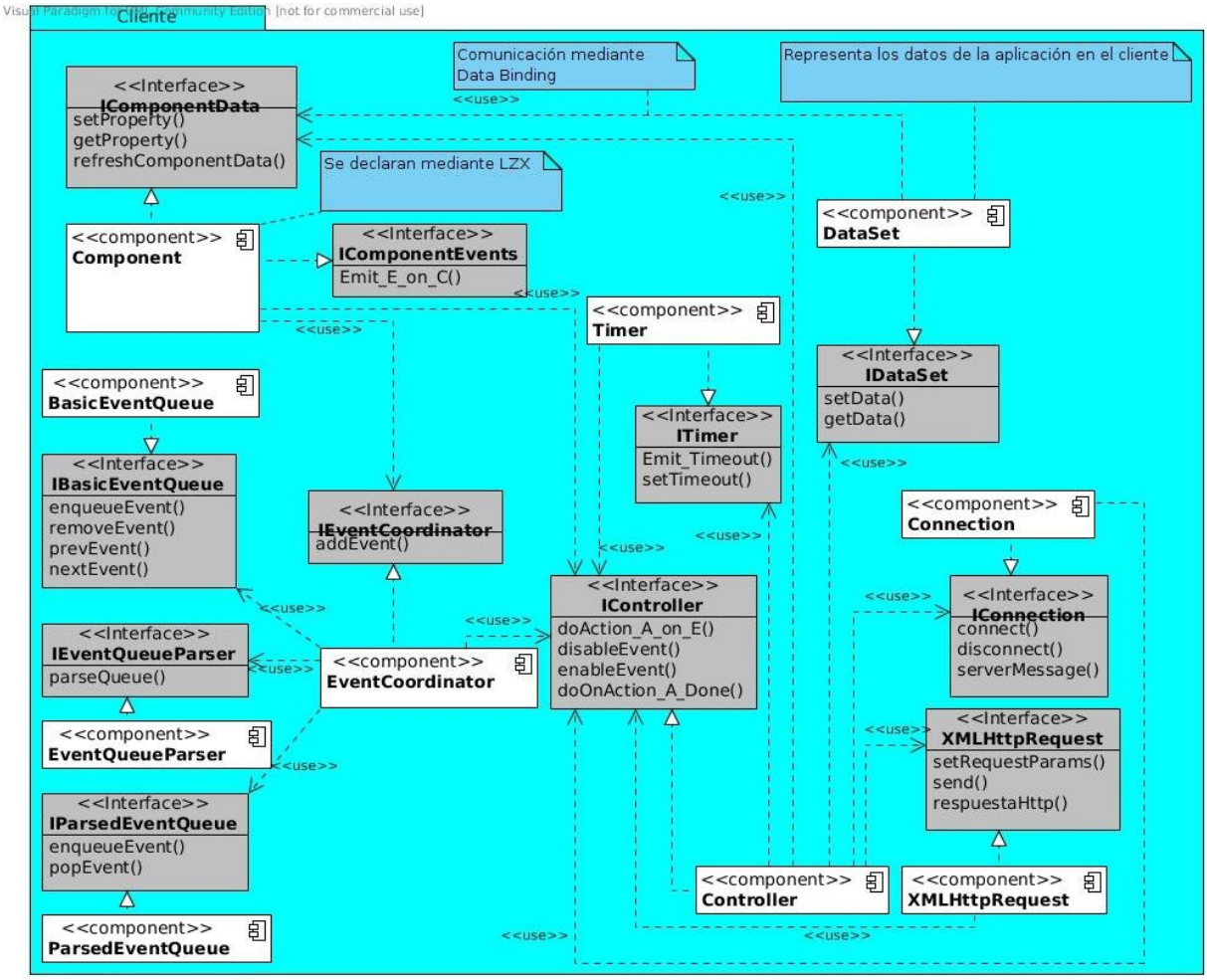

Figura 8-10: Paquete cliente de la arquitectura de OpenLaszlo

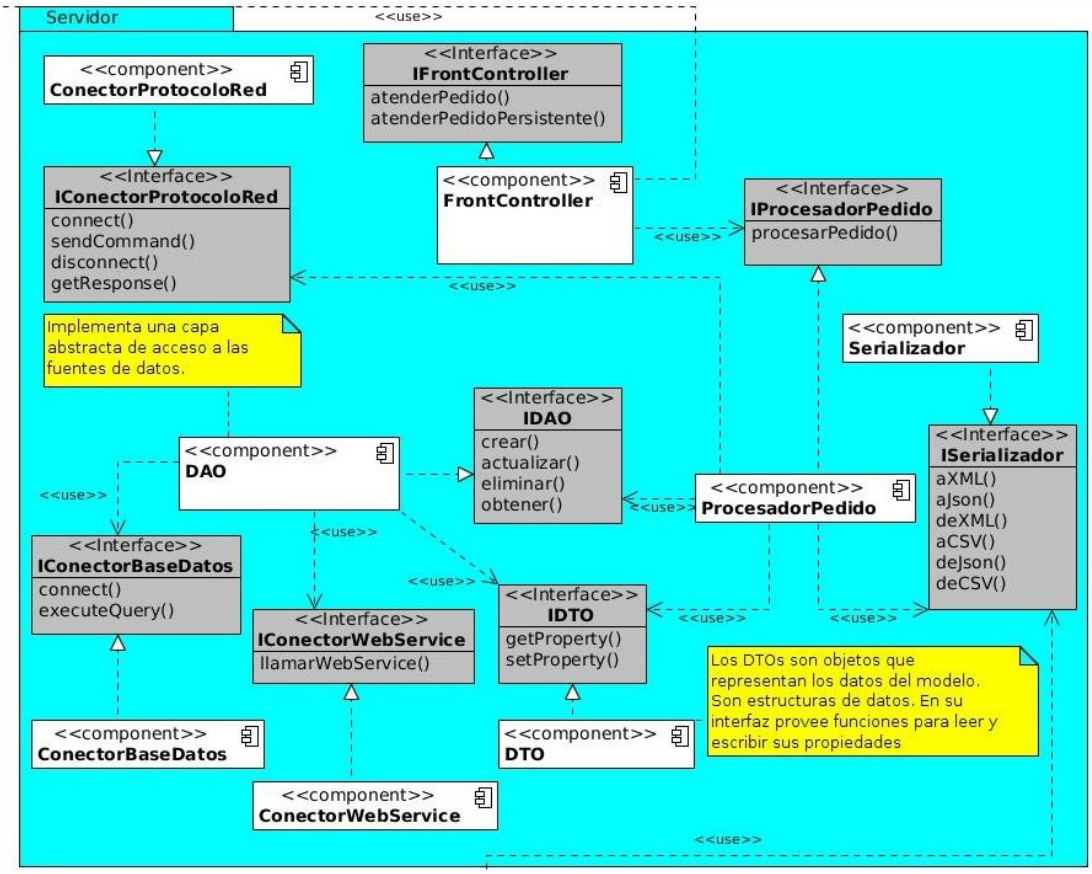

Figura 8-11: Paquete Servidor de la arquitectura de OpenLaszlo.

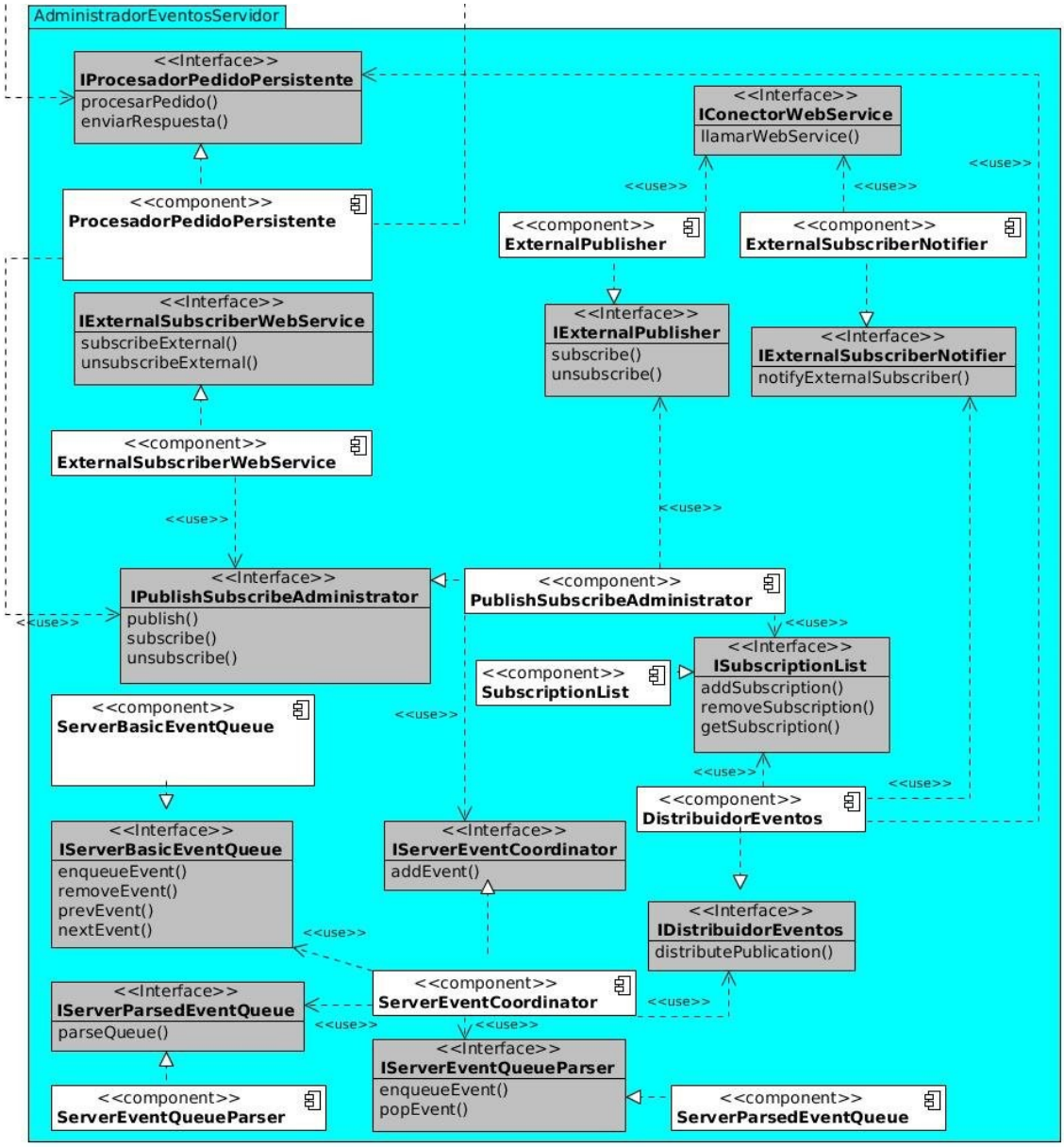

Figura 8-12: Paquete AdministradorEventosServidor de la arquitectura de OpenLaszlo.

Veamos ahora la descripción técnica de cada una de estas componentes.

Nombre	**Component**
Descripción	Se encarga de la representación visual de los datos de la aplicación. Toda la interacción del usuario con la aplicación ocurrirá a través de ellos. Provee dos interfaces: IComponentData y IComponentEvents. La primera permite manipular las propiedades visuales. La segunda es una interfaz de control, a través de la cual el usuario puede interactuar con el Component. Existen elementos de la interfaz que son utilizados para definir el layout de la aplicación, que no tienen interfaz de control dado que el usuario no interactúa con ellos directamente (p.e.: los que ofician de contenedores)
Interfaces Provistas	IComponentEvents • Emit_<event>_on_<ComponentName>() - Es llamado como resultado de una interacción <event> (p.e: onClick) del usuario con el Component <ComserverMessageponentName>. El evento es pasado al EventCoordinator o a Controller dependiendo de si el evento puede formar parte de un evento compuesto o no. IComponentData • setProperty(propertyName, propertyValue) – Setea una propiedad visual del Component (ancho, alto, color, etc.) • getProperty() – Devuelve el valor de una propiedad visual del Component (ancho, alto, color, etc.) • refreshComponentData(data) – actualiza los datos de la vista a partir de los datos del DataSet asociado. Esto es llamado internamente por el sistema de Data Binding
Dependencias	• IEventCoordinator, IController, IDataSet
Comportamiento	• **EventCoordinator:** Cuando un usuario de la aplicación interactúa con el Component que implementa la interfaz IComponentEvents, este captura dicho evento y, si es un posible candidato a formar parte de un evento compuesto, se lo delega a EventCoordinator llamando a addEvent(). • **Controller:** Cuando un usuario de la aplicación interactúa con el Component que implementa la interfaz IComponentEvents, este captura dicho evento y, si NO es un posible candidato a formar parte de un evento compuesto, se lo delega a Controller llamando a doAction_A_on_E. • **DataSet:** A través del sistema de data binding provisto por OpenLaszlo, se pueden asociar las propiedades de un DataSet con las propiedades de un Component, de manera que, cuando ocurre algún cambio en el estado del DataSet, automáticamente se refresca el Component asociado para reflejar dicho cambio llamando a refreshComponentData.
# de inst.	Una o Más

Nombre	**Controller**
Descripción	Este componente es el encargado de manejar los eventos que resultan de la interacción del usuario con los Component interactivos y los que son disparados desde la misma aplicación, como el evento timout de Timer.
Interfaces Provistas	IController • doAction_A_on_E() implementa la respuesta al evento E. • disableEvent(eventType) - permite desactivar un evento • enableEvent(eventType) - permite activar un evento • doOnAction_A_done(response) – función de callback que será llamada tras haber concluido el pedido al servidor solicitado por el controller
Dependencias	• IComponentData • IEventCoordinator • IXMLHttpRequest • IDataSet • ITimer • IConnection
Comportamiento	• **Component:** como respuesta a los eventos generados a causa de la interacción del usuario con un component, se llama a doAction_A_on_E, siempre y cuando dichos eventos no sean candidatos a formar parte de un evento compuesto. • **EventCoordinator:** cuando EventCoordinator detecta un evento compuesto, es Controller el encargado de responder a tal evento llamando a doAction_A_on_E. • **XMLHttpRequest:** para obtener los datos remotos para llenar el dataset local, Controller llamará a setRequestParams y send de XMLHttpRequest y este llamará a doOnAction_A_done de Controller como función de callback luego de que el pedido al servidor ha concluido. • **DataSet:** Controller utilizará DataSets para almacenar los datos recibidos del servidor de manera local, llamando a setData(). • **Timer:** cuando Controller necesite hacer uso de un temporizador, llamará a setTimeout de Timer. Si ocurriera el evento timeout, Timer llamará a doAction_A_on_E correspondiente. • **Connection:** Cuando Controller necesita subscribirse a un evento del servidor para recibir notificaciones de la ocurrencia de tales eventos, abre una conexión persistente llamando a connect de Connection. Cuando la conexión ya no es necesaria, se cierra llamando a disconnect de Connection.
# de instancias	Una o más

Nombre	**DataSet**
Descripción	Este componente unifica el formato de los datos de la aplicación independientemente de la fuente de datos. Ya sea que el servidor envíe sus datos en formato XML, CSV, JSON o cualquier otro, siempre son almacenados en DataSets, y serán tratados como tales a lo largo de toda la aplicación del lado del cliente.
Interfaces Provistas	IDataSet • setData(xml) – permite agregar datos al dataset. • getData() – permite obtener datos del dataset
Dependencias	• IController • IComponentData
Comportamiento	• **Controller:** Controller utilizará DataSets para almacenar del lado del cliente, datos de la aplicación recibidos del servidor. • **Component** Los DataSet se comunican con los Components a través del sistema de Data Binding provisto por OpenLaszlo. Al hacer data binding, los Components son notificados ante cualquier cambio de estado en los DataSets a los cuales están asociados, y se actualizan para reflejar dicho cambio llamando a refreshComponentData.
# de instancias	Una o más

Nombre	**XMLHttpRequest**
Descripción	Permite enviar pedidos asincrónicos al servidor y asociar una funcion de callback para manejar la respuesta.
Interfaces Provistas	IXMLHttpRequest • setRequestParams(requestHeaders) – Setea los encabezados del pedido http • send(request, callback) – envía el pedido al servidor y llama a callback con el resultado del pedido como parámetro • respuestaHttp() - es llamado como funcion de callback por el browser una vez que concluye la transferencia de la respuesta desde el servidor.
Dependencias	IController
Comportamiento	• **Controller:** cuando Controller necesita realizar un pedido al servidor, llama a setRequestParams y a send de XMLHttpRequest, pasándole como parámetros los datos para el pedido y una función de callback para procesar la respuesta.
# de instancias	Una o más

Nombre	**Connection**
Descripción	Permite a la aplicación establecer conexiones persistentes con el

	servidor de forma de poder subscribirse a eventos y recibir notificaciones del servidor cuando estos ocurran.
Interfaces Provistas	IConnection • connect() – Establece una conexión persistente con el servidor • disconnect() - Cierra la conexión persistente previamente establecida con el servidor mediante connect() • serverMessage() - es llamado como función de callback cuando la conexión persistente recibe un mensaje del servidor.
Dependencias	IController
Comportamiento	• **Controller:** cuando la aplicación se subscribe a un evento del servidor, Controller abre una conexión persistente con el servidor para recibir notificaciones de la ocurrencia de tales eventos, llamando a connect de Connection. Cuando la conexión ya no es necesaria, Controller llama a disconnect de Connection. Cuando llega un mensaje del servidor a traves de la conexión persistente, Connection llama a doAction_A_on_E de Controller.
# de instancias	Una o más

Nombre	**Timer**
Descripción	Permite establecer tareas temporizadas, que se ejecutarán después de cierto intervalo de timepo.
Interfaces Provistas	ITimer • setTiemout(time) – Permite iniciar un temporizador • Emit_Timout() - Dispara un evento avisando que el tiempo seteado ha transcurrido
Dependencias	IController
Comportamiento	• **Controller:** cuando un Controller necesita de un temporizador como parte del procesamiento de un evento, llamará a setTimeout() del componente Timer. Cuando el evento timout ocurre, se llama a doAction correspondiente de Controller
# de instancias	Una o más

8.4 La tecnología AJAX

8.4.1 Introducción

Las aplicaciones AJAX (ver [50]) se ejecutan en el cliente, es decir, en el navegador del usuario mientras se mantiene una comunicación asíncrona con el servidor en

segundo plano. Así, es posible realizar cambios sobre las páginas sin necesidad de recargarlas, lo que aumenta significativamente la interactividad, velocidad y usabilidad de la aplicación.

AJAX significa Asynchronous Javascript and XML. Javascript es el lenguaje interpretado en el que se llama a Ajax, mientras que el acceso a los datos se realiza mediante XMLHttpRequest, objeto disponible en los navegadores actuales. No es necesario que el contenido asíncrono este formateado en XML.

AJAX es una combinación de 4 tecnologías ya existentes:

- **XHTML** + **CSS** para la estructura visual y la apariencia con que será mostrada la información.
- **DOM** para modificar e interactuar dinámicamente la estructura visual y apariencia de la información.
- **XMLHttpRequest** para intercambiar datos de forma asíncrona con el servidor Web.
- **XML** es el formato utilizado generalmente para la transferencia de datos solicitados al servidor, aunque cualquier formato puede funcionar (HTML, JSON, etc.)
- **Javascript** es un lenguaje de scripting del lado del cliente que funciona como nexo entre el resto de las tecnologías.

Ajax no es una tecnología en sí, sino que es un término que engloba a un grupo de éstas que trabajan conjuntamente.

Normalmente, para lograr una buena organización y confiable implementación de todo esto, se utilizan librerías de JavaScript como Jquery (ver [51]), Script.aculo (ver [52]), Dojo (ver [53]), Prototype (ver [54]), las cuales apuntan a:

- Proveer una interfaz amigable para el manejo de pedidos asincrónicos al servidor, es decir, una abstracción de la funcionalidad provista por el objeto XMLHttpRequest.
- Resolver las incompatibilidades entre las implementaciones de JavaScript de los diferentes navegadores.
- Proveer una interfaz amigable para lógica de uso frecuente, como por ejemplo, manipulación dinámica de DOM y CSS.
- Proveer un conjunto de widgets, elementos visuales configurables que enriquecen la interfaz con el usuario e incrementan la interacción y representan visualmente el modelo de datos.
- Proveer un conjunto de efectos visuales para obtener una aplicación más interactiva y atractiva.
- Proveer una interfaz para el manejo de eventos, usualmente permitiendo que los objetos se subscriban a eventos y permitiendo definir la manera en que responden a dichos eventos.

Con el objeto de desacoplar las diferentes unidades funcionales, se propone una arquitectura orientada a eventos. Así, lo componentes visuales, los objetos JSON que representan el modelo de datos y las acciones remotas se comunicarán a través de un administrador de eventos, mediante el cual pueden disparar eventos propios, y subscribirse a eventos disparados por otros componentes. El administrador de eventos es un objeto javascript que basa su funcionalidad sobre la API de eventos de la librería javascript subyacente.

Para encapsular la interfaz con el servidor, es decir, los servicios remotos disponibles, se define también un componente Controlador. Este componente es un objeto javascript que hace uso del objeto XMLHttpRequest para realizar pedidos asincrónicos al servidor.

8.4.2 Descripción de la Arquitectura

Las aplicaciones basadas en AJAX presentan una arquitectura cliente/servidor.

El lado cliente de la aplicación es construido sobre la base de 4 tecnologías: XHTML, CSS, DOM y JavaScript.

XHTML es un lenguaje de hipertexto estándar basado en XML que permite definir los elementos que componen a una página web, así como formularios, áncoras, enlaces, contenedores, imágenes, entre otros. En resumen, mediante XHTML se define la estructura del sitio, sin especificar posiciones, colores, fuentes, tamaños ni ningún concepto relacionado con la presentación, dado que para todo esto utilizaremos CSS.

CSS u hojas de estilo en cascada son un lenguaje formal usado para definir la presentación de un documento estructurado escrito en HTML, XML o XHTML. La idea principal es la de separar la estructura de la presentación con el fin de lograr un desacoplamiento efectivo que favorezca el mantenimiento y la reutilización. Provee directivas para establecer posiciones, fuentes, color del texto, bordes, márgenes, alineación, entre otras. Si bien las directivas CSS pueden incluirse dentro del mismo documento HTML, e incluso dentro de cada tag HTML a través del atributo “style”, lo más recomendable es tenerlo en un archivo separado y enlazarlo al documento utilizando el tag “link”.

DOM o Document Object Model es una interfaz de programación que proporciona un conjunto estándar de objetos para representar documentos HTML y XML, un modelo estándar sobre como pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos. A través de DOM, los programas pueden acceder y modificar el contenido, estructura y estilo de los documentos HTML y XML que es para lo que diseño originalmente.

JavaScript es un lenguaje de programación interpretado, es decir, no requiere compilación. Es un lenguaje orientado a objetos, ya que dispone de herencia, si bien esta se realiza siguiendo un paradigma de programación basada en prototipos. Para poder interactuar con la página web, se provee al lenguaje Javascript de una implementación DOM. También cuenta con una implementación de la interfaz XMLHttpRequest, uno de los pilares de AJAX dado que permite realizar pedidos HTTP de manera asíncrona, evitando así, la necesidad de recargar la página

completa. Nuevamente, el código JavaScript puede ser incluido en el documento HTML, pero es recomendable tenerlo en un archivo separado a los fines de poder mantenerlo y reutilizarlo con mayor grado de facilidad.

La estructura general de las librería se divide en dos partes. Una parte se denomina **núcleo de la librería (library core)** que contiene aquella funcionalidad que se considera básica para desarrollar una aplicación RIA usando Javascript. Esto incluye, manejo de eventos, comunicación asincrónica a través de XMLHttpRequest y manejo del DOM. La otra parte, contiene todos aquellos objetos y funcionalidad destinada a enriquecer la interfaz del usuario. Esto incluye, widgets, efectos visuales, etc.

8.4.3 Estructura de una aplicación AJAX

Para describir la estructura de la aplicación seguimos el enfoque propuesto por [47].

Lado cliente:

Inicialmente, el usuario accede la página principal de la aplicación ingresando la URL correspondiente en su navegador web, es decir, el navegador le solicita al servidor web la página inicial de la aplicación. Cuando la página es accedida por primera vez, carga todo el javascript necesario para inicializar la aplicación. Esta inicialización incluye:

- Inicialización de la librería Javascript.
- Instanciación del controlador.
- Carga desde el servidor del estado inicial de la aplicación.
- Creación de los componentes visuales de la página inicial.
- Carga de la hoja u hojas de estilo de la aplicación.

La arquitectura del lado del cliente está basado en el patrón arquitectónico llamado Invocación Implícita. Los distintos componentes de la arquitectura interaccionan entre sí a través de un sistema de eventos. Cuando el usuario manipula la interfaz visual de la aplicación, haciendo click, drag and drop o a través del teclado, los componentes visuales disparan eventos indicando el tipo de acción que ha ocurrido. Los componentes que están suscritos a este evento particular, pueden registrar manejadores para responder al evento producido. Para esto, existe un repositorio central de eventos y manejadores de evento que permite agregar y eliminar eventos, agregar y eliminar manejadores asociados a un evento y disparar eventos.

A modo ilustrativo, el siguiente es un flujo posible de la aplicación en el lado cliente:

- El usuario interactúa con algún componente visual de entrada de la aplicación utilizando el mouse o el teclado.
- Esto dispara un evento asociado con el tipo de interacción ocurrido.
- El controlador tiene registrado un manejador de eventos para ese evento como parte de su inicialización o debido a un cambio en el estado de la aplicación.
- A través de ese manejador de eventos, el controlador actualiza el modelo del lado del cliente.

- El controlador envía el modelo actualizado al servidor el cual persiste el cambio en la base de datos.
- El servidor responde exitosamente.
- El controlador dispara un evento que notifica que la actualización se ha completado exitosamente.
- La vista principal y posiblemente el componente que generó el primer evento tienen registrado un manejador para el evento generado por el controlador.
- Se ejecuta dicho manejador, actualizando la vista, particularmente algún widget visual de salida, permitiendo al usuario saber el resultado de su acción.

A continuación, se presenta una breve reseña de cada componente:

- **Widgets:** estos componentes corresponden a la vista en el modelo MVC del lado del cliente. Son los encargados de representar visualmente los datos del modelo. Una misma entidad del modelo puede tener cero, una o más representaciones visuales posibles, por ejemplo, en forma de tabla, de árbol, de ficha, etc. Es mediante estos componentes que el usuario interactuará con la aplicación. Un widget permite mostrar e interactuar y manipular una o más entidades del modelo. Normalmente, extienden o enriquecen a través de JavaScript los controles ya existentes en XHTML como tablas, botones, formularios, divs, listas, etc que son manipulados utilizando el DOM y el CSS para crear efectos visuales atractivos con el objeto de incrementar la interactividad. La funcionalidad de los widgets se basa en objetos JavaScript provistos por la librería, generalmente en un archivo separado descripto en la arquitectura como un <<artifact>> denominado "Widgets librería JavaScript". Podemos clasificarlos de acuerdo a su comportamiento, de la siguiente manera:
  - *Widget Contenedor:* este tipo de widget se utiliza para definir el layout de la aplicación, definir en que parte de la pantalla se muestra cada elemento de la vista. Normalmente, contiene dentro de sus límites a otros widgets, también del tipo contenedor y/o de entrada o salida. Tabs, recuadros, paneles, son ejemplos de este tipo de widgets.
  - *Widget de Input:* permiten al usuario ingresar datos y manipular el modelo subyacente. Botones, campos de formularios, links, son ejemplos de este tipo de widgets.
  - *Widget de Output:* permiten al usuario visualizar los datos del modelo subyacente. Es decir, representan visualmente uno o más elementos del modelo de forma que el usuario pueda entenderlos. Tablas, listas, árboles, son ejemplos de este tipo de widgets.
- **Controlador:** este componente corresponde al controlador en el modelo MVC del lado del cliente. Su tarea es la de responder a los eventos generados por la vista y manipular el modelo en respuesta a esos eventos. En caso de ser necesario, se comunica con el servidor, enviándole el modelo actualizado para

que este sea persistido en la base de datos. Finalmente, dispara un evento informando sobre la respuesta del servidor. Este evento es atendido por los componentes visuales correspondientes para informar al usuario sobre el resultado de la operación. La comunicación con el servidor se basa en el componente representado en la arquitectura como "Abstracción Ajax". Este componente, provisto generalmente por el núcleo de la librería (representado como un <<artifact>> de nombre "Núcleo librería JavaScript") presenta una interfaz sencilla para la utilización del objeto XMLHttpRequest.

- **DTOs:** estos componentes corresponden al modelo en la arquitectura MVC. Es una representación del estado de la aplicación del lado del cliente, implementado a través de objetos de JavaScript que mapean directamente a los DTOs del lado servidor. Estos objetos son actualizados por el controlador cuando el usuario interactúa con la vista, y posteriormente son enviados al servidor para persistir dicha actualización. El modelo también dispara eventos cuando ocurre algún cambio en el estado de la aplicación. Los componentes visuales asociados a la parte del modelo actualizada están subscritos a estos eventos para poder actualizarse y mostrar el estado actual.

- **Administrador de Eventos:** este componente es el canal de comunicación entre la vista, el modelo y el controlador y el corazón de la arquitectura del lado del cliente. Permite a los distintos componentes registrar eventos y manejadores para estos eventos. De esta manera se reduce la dependencia y el acoplamiento, dado que no hay comunicación directa entre ellos sino que se invocan implícitamente, (excepto entre el controlador y el modelo). El evento es representado con un objeto JavaScript que contiene los datos de lo ocurrido. Cada evento puede tener más de un manejador, lo cual permite que los diferentes componentes puedan realizar más de una acción en respuesta a un mismo evento. Los manejadores se implementan como funciones de JavaScript, que normalmente toman al evento como parámetro. Dentro de la función manejadora, se lleva a cabo alguna tarea basada en los datos que contiene el evento. Un manejador puede manejar solamente un evento. Cabe destacar que los eventos y sus manejadores no son estáticos dentro de la aplicación, sino dinámicos. El administrador de eventos permite también remover manejadores previamente registrados o remover eventos para que ya no sean disparados.

- **Librería JavaScript:** su principal objetivo esta definido más arriba en este documento.

- **Hojas de estilo CSS:** representado en la arquitectura como un <<artifact>> de nombre "Hoja de Estilo CSS", consta de uno o mas archivos .css que contienen la declaración de todas las propiedades relacionadas con la presentación visual de los widgets, ya sea posiciones, colores, tamaños, tipos de letra, bordes, etc.

- **Invocación Implícita:** Hemos creado un estereotipo de conector denominado <<invocacionImplicita>> para representar la interacción indirecta de los componentes a través de eventos.

Lado Servidor

Del lado servidor se recibe un pedido HTTP. Al servidor en sí, no le interesa si el pedido ha sido realizado de manera asincrónica o sincrónica. Sin embargo, la programación de la lógica del lado servidor varía con respecto a la programación de aplicaciones web tradicionales. En la web tradicional, es normal que el servidor envíe toda una página HTML completa en cada respuesta al cliente. En contraste, las aplicaciones Ajax permiten enviar porciones de código HTML que luego pueden ser agregados a la página HTML dinámicamente, mediante JavaScript. Al igual que en al web tradicional, para lograr esto, generalmente se utilizan Template Engines como JSP (Java) o Smarty (PHP). Estos permiten separar la estructura visual de la aplicación, del contenido. Generalmente, proveen un conjunto de etiquetas que cuando son ejecutadas, generan el código HTML para dar formato visual a un conjunto de datos.

Además, las aplicaciones Ajax permiten también mantener todos los aspectos de presentación en el cliente. Esto es, el servidor sólo envía datos en algún formato particular y el cliente, mediante JavaScript, se encarga de parsear esos datos, armar la estructura visual para mostrarlos y agregarlos dinámicamente en el documento existente. Para este tipo de implementaciones se utiliza JSON para codificar los datos que envía el servidor, ya que es una notación compacta y directamente evaluable por JavaScript, lo cual la hace más rápida de transferir y de interpretar. Este es el modo de comunicación representado en el gráfico de la arquitectura.

El lado servidor de la aplicación puede estar implementado en un lenguaje de scripting como Java, PHP, Python, Ruby, etc. o puede ser un Web Service del tipo SOAP o XML-RPC, las cuales implementan RPC a través del protocolo HTTP, enviando y recibiendo mensajes codificados en XML. Se elimina toda la funcionalidad que no requiere en realidad la intervención del servidor, pasando esto a ser implementado del lado cliente. En el caso de aplicaciones basadas en Ajax, lo normal, es tener la lógica del lado servidor, implementada en algún lenguaje de scripting y los web services son aplicaciones externas, quizás de otros fabricantes a las cuales se conecta nuestra aplicación para obtener datos, aunque esto puede variar dependiendo del tipo de aplicación.

A continuación se describen los componentes que forman parte de la arquitectura del servidor:

- **Controlador:** el controlador es un objeto implementado en el lenguaje de programación del servidor. Se encarga de recibir los pedidos del cliente, decodificar la información recibida, instanciar el servicio correspondiente y proveerle dicha información, y finalmente, codificar la respuesta antes de enviarla al cliente. Esta compuesto por los siguiente componentes.
    - *Front Controller:* recibe los pedidos del cliente, decodifica los datos del pedido y lo delega al servicio correspondiente. Cuando el servicio termina de procesar, le entrega los datos al front controller, el cual los codifica y a continuación los envía al cliente.
    - *Codificador/Decodificador JSON:* cuando llega un pedido del cliente, el

controlador le pedirá a este componente que decodifique los datos recibidos en formato JSON. De la misma forma, cuando se envía la respuesta desde el servidor, el controlador le pedirá que codifique la respuesta.

- *Implementación de Servicio:* cada acción permitida desde el servidor está encapsulada dentro de un objeto definido en el lenguaje del servidor. Cuando llega un pedido, el front controller instancia el servicio correspondiente al tipo de pedido y le delega la responsabilidad.

- **Modelo:** representa el estado actual de la aplicación. Contiene a los siguientes componentes:
  - *DAOs:* Data Access Object, son los objetos encargados de manipular los datos que conforman el modelo. El DAO permite crear, modificar, obtener o eliminar un determinado elemento del modelo. Generalmente, estas acciones mapean a las sentencias INSERT, UPDATE, SELECT y DELETE de SQL respectivamente.
  - *DTOs:* Data Transfer Object. El modelo y la implementación de un servicio se comunican a través de estos componentes. Representan una unidad de información, generalmente mapean a un registro de una tabla de la base de datos. El servicio instancia un DAO para manipular los datos del modelo. El DAO devuelve un DTO al servicio que luego es codificado y devuelto al cliente con el resultado correspondiente.
  - *Conectores:* Dependiendo de cual sea la fuente de los datos, los DAO utilizan diferentes conectores provistos por el lenguaje del servidor para obtener la información del modelo.

Cabe hacer un comentario relacionado a los RPC y Web Services. Por razones de seguridad, los navegadores suelen prohibir los llamados desde el lado cliente de la aplicación a sitios externos, es decir, fuera del dominio dentro del cual se accede la aplicación. Por ejemplo, si la aplicación se accede mediante www.miaplicación.com y se intenta realizar mediante XMLHttpRequest un pedido al sitio www.otrositio.com, el navegador no lo permitirá. Este tipo de pedidos externos suelen ser necesarios cuando intentamos utilizar un Web Service, alojado en un dominio externo. Existen formas de resolver este problema, pero considero que deberá ser tenido en cuenta cuando escribamos la arquitectura abstracta para aplicaciones RIA.

Una forma de resolver este inconveniente y lograr que el Web Service sea transparente para el cliente es utilizar la aplicación del lado servidor como puente para obtener datos del web service. Es decir, el cliente realiza un pedido normal al servidor, y este, internamente, obtiene los datos del web service, los formatea, los codifica y se los retorna al cliente en un formato que este puede interpretar.

La Figura 8-13 describe la arquitectura de una aplicación Ajax.

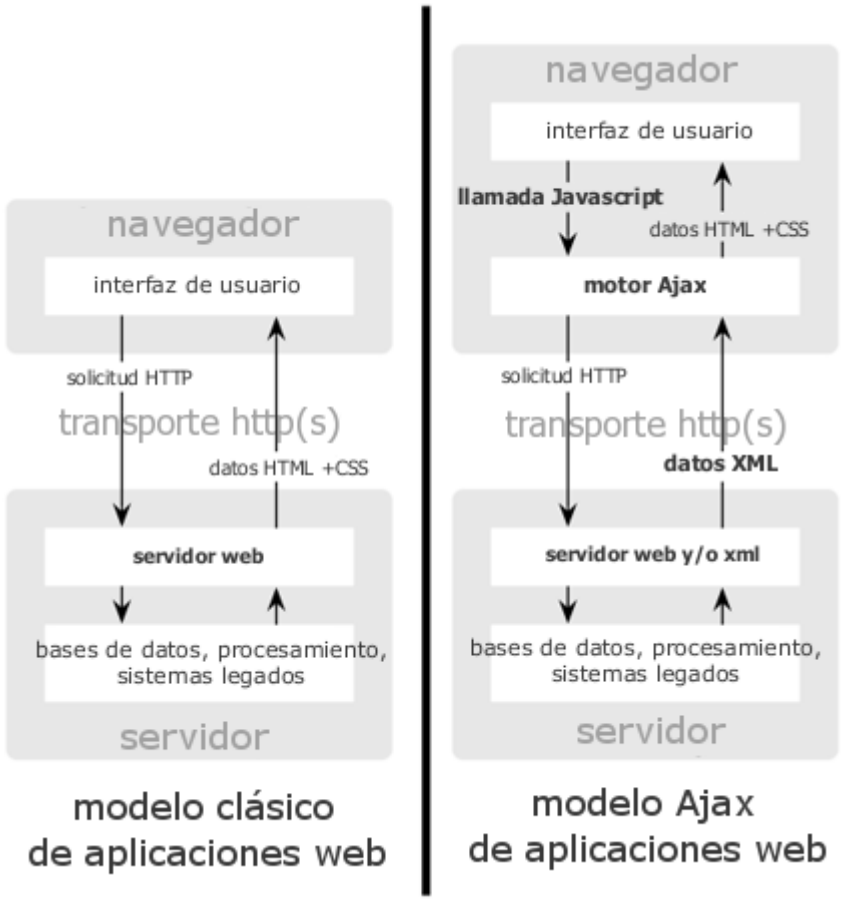

Figura 8-13: Arquitectura Ajax

8.4.4 Representación UML de la arquitectura

Nombre	**BasicWidget**
Descripción	Se encarga de la representación visual de los datos de la aplicación. Provee dos interfaces: IWidgetControl y IWidgetData. La primera permite manipular las propiedades visuales y los datos que va a mostrar el widget. La segunda es una interfaz de control, a través de la cual el usuario puede interactuar con el widget. El widget captura los eventos del usuario y se los delega a EventCoodinator. Algunos widgets no tienen interfaz de control ya que el usuario no interactúa directamente con ellos (p.e.: los widgets que ofician de contenedores)
Interfaces Provistas	IWidgetControl • Emit_<event>_on_<basicWidgetName>() - Es llamado como resultado de una interacción <event> (p.e: onClick) del usuario con el widget <basicWidgetName>. El evento es pasado al EventCoordinator IWidgetData • operaciones de tipo set para fijar contenidos en el widget. • operaciones de tipo get para obtener contenidos del widget. • setProperty(propertyName, propertyValue) – Setea una propiedad visual del widget (ancho, alto, color, etc.) • getProperty() – Devuelve el valor de una propiedad visual del widget (ancho, alto, color, etc.)
Dependencias	• JSL Librería Widgets – Los BasicWidgets se basan en la librería de widgets provistos por la librería de JavaScript utilizada (jquery, prototype, dojo, etc.) • IEventCoordinator
Comportamiento	• **EventCoordinator:** Cuando un usuario de la aplicación interactúa con el widget que implementa la interfaz IWidgetControl, haciendo click por ejemplo, este captura dicho evento y se lo delega a EventCoordinator llamando a addEvent(). Dentro del evento que el widget le pasa a EventCoordinator están todos los datos necesarios para procesar tal evento. • **ResultsViewer:** Utilizando la interfaz IWidgetData, el componente ResultsViewer modificará el estado visual del widget para que este refleje el resultado del procesamiento de un evento. Para esto, llamará a setData() y setProperty().
# de instancias	Una o más

Nombre	**EventCoordinator**
Descripción	Este componente coordina el análisis de eventos básicos con el fin de encontrar eventos compuestos. Para esto, almacena los eventos que van ocurriendo en la cola de eventos básicos y los analiza utilizando el parser. Si el parser detecta la ocurrencia de un evento compuesto, el EventCoordinator pone ese evento en la cola de eventos parseados y luego lo utilizará para saber a que método de la interfaz de AdministradorEventos debe llamar.
Interfaces Provistas	IEventCoordinator • addEvent(basicEvent) – realiza la coordinación entre la cola de eventos básicos, parser y cola de eventos parseados para detectar eventos compuestos y luego solicitar que estos sean tratados.
Dependencias	• IAdministradorEventos, IBasicEventQueue, IEventQueueParser. IParsedEventQueue
Comportamiento	• **BasicWidget, JsProcesadorEvento, Timer:** Estos tres componentes son los que hacen uso de EventCoordinator, agregando enventos a través de addEvent() cuando estos ocurren. • **BasicEventQueue:** Utilizada por EventCoordinator para almacenar eventos básicos llamando a enqueueEvent(). También llama a prevEvent() y nextEvent() para recorrer los eventos encolados. En la medida que los eventos son reconocidos por el parser como componentes de un evento compuesto, EventCoodinator llama a removeEvent() para sacarlo de la cola. Esto puede suceder como consecuencia de una restricción temporal, es decir, luego de que transcurre un cierto tiempo desde que el evento fue encolado y no se detecta que forme parte de un evento compuesto, se lo descarta. También puede suceder que un evento sea descartado luego de que ocurre otro evento que resulta ser el último de la secuencia que conforma un evento compuesto. • **EventQueueParser:** Utilizado por EventCoordinator para pasear los eventos básicos encolados y detectar eventos significativos de la aplicación llamando a parseQueue. • **ParsedEventQueue:** Utilizado por EventCoordinator para almacenar eventos parseados llamando a enqueueEvent(), es decir, eventos significativos para la aplicación. Luego de utilizar el evento parseado para saber a que método de AdministradorEvento debe llamar, EventCoordinator remueve el evento de la cola llamando a popEvent(). • **AdministradorEventos:** Los eventos parseados son utilizados por EventCoordinator para saber a qué método de la

	interfaz de AdministradorEventos debe llamar.
# de instancias	Una

Nombre	**BasicEventQueue**
Descripción	La cola de eventos básicos es utilizada para almacenar eventos que ocurren en los BasicWidgets, Timers o JsProcesadorEventos
Interfaces Provistas	IBasicEventQueue • enqueueEvent(basicEvent) – Permite agregar un evento a la cola de eventos básicos • removeEvent(basicEvent) – Permite eliminar un evento de la cola de eventos básicos. • prevEvent(), nextEvent() - Permiten recorrer la cola de eventos básicos.
Dependencias	• No tiene
Comportamiento	• **EventCoordinator:** La BasicEventQueue será utilizada por EventCoordinator para almacenar eventos básicos llamando a enqueueEvent(). Para recorrer los eventos encolados, llamará a prevEvent() y nextEvent(), y una vez que el evento es descartado por restricciones temporales, o porque se detectó que forma parte de un evento compuesto que fue procesado, lo elimina de la cola llamando a removeEvent().
# de instancias	Una

Nombre	**EventQueueParser**
Descripción	El parser es el encargado de analizar los items de la cola de eventos básicos en busca de eventos compuestos o eventos significativos para el sistema.
Interfaces Provistas	IEventQueueParser • parseQueue(basicEventList) – Recibe uno o más eventos básicos para analizar y pone el resultado en la cola de eventos parseados.
Dependencias	• No tiene
Comportamiento	• **EventCoordinator:** El parser será utilizado por EventCoordinator para realizar el análisis de los eventos básicos encolados, en busca de eventos compuestos.
# de instancias	Una

Nombre	**ParsedEventQueue**
Descripción	La cola de eventos parseados será utilizada para almacenar el resultado del parseo de eventos básicos. En esta cola solo se almacenan los eventos significativos del sistema.
Interfaces Provistas	IParsedEventQueue • enqueueEvent(parsedEvent) – Permite encolar un evento parseado. • popEvent() - Permite extraer y eliminar el primer elemento de la cola de eventos parseados.
Dependencias	• No tiene
Comportamiento	• **EventCoordinator:** El EventCoordinator llamará a enqueueEvent() para agregar eventos parseados a la cola y a popEvent() para extraer eventos de la misma, luego de utilizarlo para saber a qué método del AdministradorEventos debe llamar.
# de instancias	Una

Nombre	**AdministradorEventos**
Descripción	El administrador de eventos es el encargado de recibir los eventos parseados generados tanto por el sistema como por la interacción del usuario y delegarlos al procesador de evento correspondiente.
Interfaces Provistas	IAdministradorEventos • fireEvent(event, eventData) – Se encarga de ejecutar el JsProcesadorEvento de acuerdo al evento recibido como parámetro. El parámetro eventData está el detalle del evento. • enableEvent(event) – Permite habilitar un evento, es decir, cuando ocurra el evento, se ejecutará el JsProcesadorEvento correspondiente. • disableEvent(event) – Permite deshabilitar un evento, es decir, cuando ocurra el evento, NO se ejecutará el JsProcesadorEvento asociado.
Dependencias	• IJsProcesadorEvento, IResultsViewer
Comportamiento	• **EventCoordinator:** Cuando se termina de parsear un evento, EventCoordinator llama a fireEvent() pasándole como parámetro el evento ocurrido. • **JsProcesadorEvento:** Cuando EventCoordinator llama a fireEvent(), este delega el procesamiento del evento ocurrido

	llamando a procesarEvento() del JsProcesadorEvento asociado con el evento ocurrido. • **ResultsViewer:** Cuando el procesamiento de un evento termina, JsProcesadorEvento llama a fireEvent() que a su vez llama a ShowResultPart_<partialResult> de ResultsViewer para modificar los widgets de forma que reflejen el resultado del procesamiento de tal evento.
# de instancias	Una

Nombre	**JsProcesadorEvento**
Descripción	Es el encargado de realizar todas las acciones correspondientes a fin de responder al evento al que está asociado.
Interfaces Provistas	IJsProcesadorEvento • procesarEvento() - Contiene la lógica necesaria para responder al evento. • manejarRespuesta() - En caso de que exista una comunicación con el servidor como parte del procesamiento del evento, este método será el encargado de recibir el resultado de esa comunicación y tomar las acciones correspondientes dependiendo de si la respuesta fue exitosa o fallida.
Dependencias	• IEventCoordinator • IjsDataStore • ITimer • IJsProcesadorPedidoRemoto
Comportamiento	• **AdministradorEventos:** Cuando ocurre un evento significativo del sistema, AdministradorEventos llama a procesarEvento() del procesador que se encarga de responder al evento ocurrido. • **JsProcesadorPedidoRemoto:** En caso de ser necesaria una comunicación con el servidor como parte del procesamiento de un evento, JsProcesadorEvento llamará a hacerPedido() de de JsProcesadorPedidoRemoto. • **JsDataStore:** La respuesta del servidor es procesada por manejarRespuesta() y a continuación, en caso de ser necesario, esa respuesta es almacenada en el navegador utilizando JsDataStore. Este funciona como cache de una parte de los datos de la aplicación, en el navegador. Se almacena en una variable JavaScript que puede ser consultada a través de queryDataStore(). • **EventCoordinator:** Un procesador de evento puede disparar otro evento que será agregado a la cola de eventos básicos llamado a addEvent() • **Timer:** El procesador de evento puede utilizar un timer

	como parte de la lógica de procesamiento llamando a setTimeout().
# de instancias	Una o más

Nombre	**JsDataStore**
Descripción	Guarda una copia local de los datos enviados por el servidor, de modo de acelerar la velocidad de respuesta de la aplicación.
Interfaces Provistas	IJsDataStore • loadDataStore() - Permite almacenar los datos enviados por el servidor en el browser. • queryDataStore() - Permite realizar búsqueda de datos dentro del data store.
Dependencias	• No tiene
Comportamiento	• **JsProcesadorEvento:** cuando se llama a manejarRespuesta() de JsProcesadorEvento como resultado de una respuesta del servidor, este llama a loadDataStore() de JsDataStore. Cuando el procesamiento de un evento necesita datos de la aplicación, JsProcesadorEvento llama a queryDataStore() para ver si existe una copia local de esos datos. En caso de que no, hace una llamado al servidor.
# de instancias	Una o más

Nombre	**Timer**
Descripción	Permite establecer tareas temporizadas, que se ejecutarán después de cierto intervalo de tiempo.
Interfaces Provistas	ITimer • setTiemout(time) – Permite iniciar un temporizador • Emit_Timout() - Dispara un evento avisando que el tiempo seteado ha transcurrido
Dependencias	• IEventCoordinator
Comportamiento	• **JsProcesadorEvento:** cuando un procesador de evento necesita de un temporizador como parte del procesamiento de un evento, llamará a setTimeout() del componente Timer. • **EventCoodinator:** Cuando el tiempo seteado para el timer mediante setTimeout() ha transcurrido, el Timer avisa de tal evento llamando a addEvent() de EventCoordinator.
# de instancias	Una o más

Nombre	**JsProcesadorPedidoRemoto**
Descripción	Se encarga de establecer una comunicación asincrónica o sincrónica con el servidor utilizando XmlHttpRequest. Todas las comunicaciones, tanto el pedido como la recepción de la respuesta del servidor son realizadas a través de este componente.
Interfaces Provistas	IJsProcesadorPedidoRemoto • hacerPedido() - Establece la comunicación con el servidor y envía un pedido http asincrónico. • tratarRespuesta() - Es llamado cuando se termina de recibir la respuesta desde el servidor y delega la responsabilidad de manejar la respuesta llamando a manejarRespuesta de JsProcesadorEvento.
Dependencias	• No Tiene
Comportamiento	• **JsProcesadorEvento:** Cuando JsProcesadorEvento necesita hacer un pedido al servidor para persistir u obtener datos de la aplicación, llama a hacerPedido(). La respuesta del servidor es recibida a través de tratarRespuesta() el cual llama a manejarRespuesta() de JsProcesadorEvento. Existen escenarios en los que es necesario que el pedido sea sincrónico. Tener en cuenta que XmlHttpRequest permite pedidos sincrónicos y asincrónicos.
# de instancias	Una

Nombre	**JsProcesadorPedidoRemotoPersistente**
Descripción	Es igual a JsProcesadorPedidoRemoto, solo que mantiene la conexión con el servidor, y recibe periódicamente datos de este, aunque no los solicite, mediante la técnica de data push.
Interfaces Provistas	IJsProcesadorPedidoRemoto • hacerPedido() - Establece la comunicación con el servidor y envía un pedido http asincrónico. La conexión permanece abierta a la espera de mensajes del servidor. • tratarRespuesta() - Es llamado cada vez que el servidor envía un mensaje a través de la conexión. La respuesta es delegada llamando a manejarRespuesta de JsProcesadorEvento.
Dependencias	• No Tiene
Comportamiento	• **JsProcesadorEvento:** JsProcesadorEvento hace uso de JsProcesadorPedidoRemotoPersitente, de modo de establecer una comunicación con el servidor, dejando la conexión abierta para que el servidor pueda notificar la ocurrencia de eventos a los que el cliente está subscripto.

# de instancias	Una

Nombre	**ResultsViewer**
Descripción	Manipula las propiedades visuales y los datos de los widgets para reflejar el resultado del procesamiento de un evento.
Interfaces Provistas	IResultsViewer • showResultPart_P() - utiliza la interfaz de datos de los widgets para mostrar el resultado del procesamiento de un pedido.
Dependencias	• IWidgetData
Comportamiento	• **AdministradorEventos:** cuando un JsProcesadorEvento termina termina de procesar el evento llama a fireEvent() de AdministradorEventos el cual llama a ShowResultPart_P() de ResultsViewer. • **BasicWidget:** ResultsViewer llamará a setDatos() y a serProperty() de BasicWidget para que este refleje el resultado del procesamiento.
# de instancias	Una o más

SERVIDOR:

Nombre	**FrontController**
Descripción	Oficia de receptor de los pedidos del cliente. Es el encargado de delegar el pedido al ProcesadorPedido correspondiente.
Interfaces Provistas	IFrontController • atenderPedido() - Recibe el pedido del cliente y lo delega el ProcesadorPedido responsable de procesarlo. • atenderPedidoPersistente() - En caso de ser un pedido con conexión persistente, este se delega a un ProcesadorPedidoPersitente.
Dependencias	• IProcesadorPedido • IProcesadorPedidoPersistente
Comportamiento	• **ProcesadorPedido:** FrontController recibe un pedido del cliente, llama a atenderPedido() y lo delega al ProcesadorPedido correspondiente llamando a procesarPedido(). • **ProcesadorPedidoPersistente:** FrontController recibe un pedido del cliente, llama a atenderPedidoPersistente() y lo delega al ProcesadorPedidoPersistente correspondiente llamando a procesarPedido().
# de instancias	Una o más

Nombre	**ProcesadorPedido**
Descripción	Cada pedido del cliente al servidor tiene un ProcesadorPedido que se encarga de procesarlo y responder.
Interfaces Provistas	IProcesadorPedido • procesarPedido() - Implementa la lógica de procesamiento y respuesta de un pedido particular del cliente.
Dependencias	• IDao • IDto • ISerializador
Comportamiento	• **FrontController:** cuando recibe un pedido del cliente, atenderPedido() instancia el ProcesadorPedido correspondiente y llama a procesarPedido(). • **DAO:** Para acceder a los datos de la aplicación, ProcesadorPedido utilizará los Dao correspondientes. Si necesita persistir datos, llamará a las operaciones de creación, actualización o eliminación del DAO. Si necesita obtener datos de la aplicación, llamar a las operaciones del DAO definidas para este fin. • **DTO:** Los DAOs almacenan los datos traídos desde la fuente de datos en DTOs, los cuales le pasa al ProcesadorPedido. De la misma forma, si procesador pedido quiere pasarle datos al DAO para que este los persista, debe pasarle un DTO con esos datos. • **Serializador:** El cliente envía en el pedido, datos serializados en algún formato estándar como XML o JSON. Estos son deserializados y convertidos a una estructura de datos en el lenguaje nativo del servidor llamando a deXML o deJSON. De la misma forma, antes de enviar la respuesta al cliente, el ProcesadorPedido realiza el proceso inverso, serializando los datos llamando a aXML o aJSON. • **ConectorProtocoloRed:** Cuando JsProcesadorPedido necesita utilizar algún servicio como FTP, SMTP, IMAP, POP3, etc, utiliza ConectorProtocoloRed para abrir una conexión y enviar comandos del protocolo requerido.
# de instancias	Una o más

Nombre	**DAO**
Descripción	Controla el acceso a las fuentes de datos de la aplicación
Interfaces Provistas	IDAO • operaciones para crear, modificar, eliminar y obtener datos.
Dependencias	• IDTO • IConectorBaseDatos • IConectorWebService
Comportamiento	• **ProcesadorPedido:** Ante el pedido de un cliente, el ProcesadorPedido correspondiente hará uso de uno o más DAOs para manipular las fuentes de datos de la aplicación, llamando a alguno de los métodos de su interfaz. • **DTO:** Una vez que el DAO obtiene los datos de la fuente de datos, los almacena en DTOs antes de devolvérselos al ProcesadorPedido llamando a setProperty(). • **ConectorBaseDatos:** En caso de que la fuente de datos sea una base de datos relacional, el DAO manipulara los datos a través de queries sql llamando a executeQuery(). • **ConectorWebService:** Cuando la fuente de datos es un Web Service, el DAO manipulará los datos a través de SOAP, XML-RPC o el protocolo que corresponda, a través de ConectorWebService llamando a llamarWebService().
# de instancias	Una o más

Nombre	**DTO**
Descripción	Los DTOs son utilizados por los DAOs para almacenar los datos obtenidos de la fuente de datos y por los ProcesadorPedido para indicarle al DAO cuales son los datos que debe persistir. Los datos entre ProcesadorPedido y DAO se intercambian a través de DTOs.
Interfaces Provistas	IDTO • setProperty(propertyName, propertyValue) – permite setear una propiedad del DTO • getProperty(propertyName) – devuelve el valor actual de una propiedad de DTO.
Dependencias	• No tiene
Comportamiento	• **ProcesadorPedido:** los DAO le darán DTO a los ProcesadorPedido y estos llamarán a getProperty de ese DTO llamando a getProperty(). Cuando ProcesadorPedido quiera persistir datos en la fuente de datos, creará un DTO con esos datos llamando a setProperty y luego llamará a alguna de las operaciones de la interfaz de DAO con ese DTO como parámetro. • **DAO:** Luego de obtener los datos de la fuente de datos, el DAO le dará esos datos al ProcesadorPedido en formato DTO.

# de instancias	Una o más

Nombre	**ConectorBaseDatos**
Descripción	Se encarga del acceso a los datos de la aplicación almacenados en una base de datos relacional. Permite ejecutar consultas para manipular dichos datos.
Interfaces Provistas	IConectorBaseDatos • connect() - permite establecer una conexión con el servidor de base de datos. • executeQuery(sqlQuery) – permite enviar consultas en formato SQL al motor de base de datos relacional.
Dependencias	• No tiene
Comportamiento	• **DAO:** los DAO utilizarán ConectorBaseDatos para enviar consultas al motor de base de datos relacional llamando a executeQuery(), con el fin de obtener o modificar los datos de la aplicación.
# de instancias	Una

Nombre	**ConectorWebService**
Descripción	Se encarga del acceso a los datos de la aplicación provistos por una aplicación externa a través de un Web Service.
Interfaces Provistas	IConectorWebService 4. llamarWebService(action, data) - permite ejecutar comandos provistos por la interfaz del Web Service.
Dependencias	• No tiene
Comportamiento	• **DAO:** los DAO utilizarán ConectorWebService para ejecutar comandos permitidos por el Web Service llamando a llamarWebService(), para obtener y enviar datos necesarios para la aplicación.
# de instancias	Una

Nombre	**ConectorProtocoloRed**
Descripción	Permite a la aplicación enviar comandos a un servidor utilizando protocolos como ftp, smtp, imap, http, etc.
Interfaces Provistas	IConectorProtocoloRed • connect(ip, port) – establece una conexión al servidor especificado por “ip” a través del puerto “port” • sendCommand(command) – envía un comando al servidor • disconnect() - cierra la conexión con el servidor • getResponse() - procesa los datos recibidos como respuesta de un comando

Dependencias	• No tiene
Comportamiento	• **ProcesadorPedido:** los JsProcesadorPedido utilizarán ConectorProtocoloRed para enviar comandos válidos al servicio indicado (ftp, smtp, imap, etc.)
# de instancias	Una

Nombre	**Serializador**
Descripción	El cliente envia en los pedidos, datos en formatos estándar como XML y JSON. El servidor necesita convertir estos datos a un formato nativo del lenguaje del servidor. El componente Serializador es el responsable de esta conversión. Permite convertir de formato estandar a nativo del servidor y viceversa.
Interfaces Provistas	ISerializador • de<standardFormat>(data) – Convierte los datos de un formato estandar (XML, JSON, etc) a una estructura de datos en el lenguaje nativo del servidor. • a<standardFormat>(data) - Convierte los datos de un formato nativo del servidor a un formato estandar (XML, JSON, etc).
Dependencias	• No tiene
Comportamiento	• **ProcesadorPedido:** Antes de empezar el procesamiento del pedido, ProcesadorPedido deserializa los datos enviados por el servidor llamando a de<standardFormat> y antes de enviar la respuesta al cliente, hace la serialización de los datos llamando a a<standardFormat>.
# de instancias	Una

AdministradorEventosServidor:

Nombre	**ProcesadorPedidoPersistente**
Descripción	Es el encargado de procesar los pedidos de subscripción del cliente a eventos en el servidor, y de notificar al cliente cuando un evento al que está subscripto ocurre.
Interfaces Provistas	IProcesadorPedidoPersistente • procesarPedido() - procesa los pedidos de subscripción a eventos del servidor. • enviarRespuesta() - notifica, a través de las conexiones persistentes abiertas por los clientes, las ocurrencias de eventos a los que el cliente está subscripto.
Dependencias	• IpublishSubscribeAdministrator • ISerializador

Comportamiento	• **FrontController:** Cuando llega un pedido persistente del cliente, se ejecuta atenderPedidoPersistente(). Este instancia el ProcesadorPedidoPersistente correspondiente y llama a procesarPedido().
	• **DistribuidorEventos:** Cuando se identifica un evento significativo, se notifica a todos los clientes subscritos a tal evento, llamando a enviarRespuesta().
	• **Serializador:** El cliente envia datos en formato estándar en el pedido al servidor, los cuales deben ser decodificados. Además, las notificaciones de evento al cliente deben ser enviadas en formato estándar. Para esto, ProcesadorPedidoPersitente hace uso de Serializador.
	• **PublishSubscribeAdministrator:** Los pedidos persistentes pueden deberse a tres motivos:
	  • pedido de subscripción a un evento del servidor. En este caso, ProcesadorPedidoPersistente llama a subscribe() de PublishSubscribeAdministrator.
	  • Pedido de publicación de un evento del cliente en el servidor. En este caso, ProcesadorPedidoPersistente llama a publish() de PublishSubscribeAdministrator.
	  • Pedido de des-subscripción de un cliente a un

	evento del servidor. En este caso llama a unsubscribe().
# de instancias	Una

Nombre	**PublishSubscribeAdministrator**
Descripción	Permite a los clientes y aplicaciones externas subscribirse, desubscribirse y publicar eventos en el servidor.
Interfaces Provistas	IPublishSubscribeAdministrator • subscribe(observer, event) – permite a un cliente o aplicación externa subscribirse a un evento del servidor. • unsubscribe(observer, event) – permite a un cliente o aplicación externa desubscribirse de un evento del servidor. • publish(event) – permite a un cliente o aplicación externa notificar al servidor de la ocurrencia de un evento.
Dependencias	• IServerEventCoordinator • ISubscriptionList • IExternalPublisher
Comportamiento	• **ProcesadorPedidoPersistente:** Cada pedido persistente buscará ejecutar una de las tres acciones permitidas por PublishSubscribeAdministrator. • **ExternalSubscriberWebService:** Cuando una aplicación externa solicita una subscripción o desubscripción a un evento del servidor a través del Web Service provisto para este fin, este llamará a subscribe() o unsubscribe() respectivamente. • **ExternalPublisherWebService:** Cuando una aplicación externa notifica al servidor de la ocurrencia de un evento a través de este Web Service, este llamará a publish(). • **ExternalPublisher:** Cuando un cliente solicita subscribirse a un evento compuesto del servidor, que tiene como componente un evento de una aplicación externa, PublishSubscribeAdministrator llama a subscribe() de ExternalPublisher. PublishSubcribeAdministrator mantiene una tabla que asocia los eventos compuestos del servidor con eventos de aplicaciones externas que forman parte de ellos. Cuando ya no es necesario que el servidor sea notificado de tal evento, llama a unsubcribe(). • **SubscriptionList:** utilizará esta lista para almacenar la asociación entre subscriptores y eventos disponibles. • **ServerEventCoordinator:** Cuando se llame a publish de PublishSubscribeAdministrator, llamará a addEvent() de ServerEventCoordinator para que el evento ocurrido sea puesto en la cola de eventos básicos.

# de instancias	Una

Nombre	**ServerEventCoordinator**
Descripción	Se encarga de recibir todos los eventos publicados en el servidor y mediante la utilización de la cola de eventos básicos, parser de eventos y cola de eventos parseados, detectar eventos significativos de la aplicación.
Interfaces Provistas	IServerEventCoordinator • addEvent(basicServerEvent) – permite indicar que se ha publicado un evento básico en el servidor, que será posteriormente analizado en busca de eventos significativos.
Dependencias	• IServerBasicEventQueue • IServerEventQueueParser • IServerParsedEventQueue • IDistribuidorEventos
Comportamiento	• **PublishSubscribeAdministrator:** cada vez que se llame a publish() de este componente, se llamará a addEvent() para encolar el evento para su posterior análisis. • **ServerBasicEventQueue:**Utilizada por ServerEventCoordinator para almacenar eventos básicos llamando a enqueueEvent. • **ServerEventQueueParser:** Utilizado por ServerEventCoordinator para pasear los eventos básicos encolados y detectar eventos significativos de la aplicación llamando a parseQueue. • **ServerParsedEventQueue:** Utilizado por ServerEventCoordinator para almacenar eventos parseados llamando a enqueueEvent, es decir, eventos significativos para la aplicación. • **DistribuidorEventos:** A medida que se detectan eventos significativos como resultado del parseo, se llama a distributePublication() de DistribuidorEventos, para notificar a los clientes o aplicaciones externas subscriptas a tales eventos.
# de instancias	Una

Nombre	**ServerBasicEventQueue**
Descripción	La cola de eventos básicos es utilizada para almacenar eventos publicados llamando a publish() de PublishSubscribeAdministrator.
Interfaces Provistas	IServerBasicEventQueue • enqueueEvent(basicEvent) – Permite agregar un evento a la cola de eventos básicos • removeEvent(basicEvent) – Permite eliminar un evento de la cola de eventos básicos. • prevEvent(), nextEvent() - Permiten recorrer la cola de eventos básicos.
Dependencias	• No tiene
Comportamiento	• **ServerEventCoordinator:** La ServerBasicEventQueue será utilizada por ServerEventCoordinator para almacenar eventos básicos, con el fin de analizarlos posteriormente en busca de eventos compuestos significativos para el sistema.
# de instancias	Una

Nombre	**ServerEventQueueParser**
Descripción	El parser es el encargado de analizar los items de la cola de eventos básicos del servidor en busca de eventos compuestos o eventos significativos para el sistema.
Interfaces Provistas	IServerEventQueueParser • parseQueue(basicServerEventList) – Recibe uno o más eventos básicos del servidor para analizar y pone el resultado en la cola de eventos parseados.
Dependencias	• No tiene
Comportamiento	• **ServerEventCoordinator:** El parser será utilizado por ServerEventCoordinator para realizar el análisis de los eventos básicos del servidor encolados, en busca de eventos compuestos.
# de instancias	Una

Nombre	**ServerParsedEventQueue**
Descripción	La cola de eventos parseados será utilizada para almacenar el resultado del parseo de eventos básicos del servidor. En esta cola solo se almacenan los eventos significativos del sistema.
Interfaces Provistas	IParsedEventQueue • enqueueEvent(parsedServerEvent) – Permite encolar eventos. • popEvent() - Permite extraer y eliminar el primer elemento

	de la cola de eventos parseados.
Dependencias	• No tiene
Comportamiento	• **ServerEventCoordinator:** El ServerEventCoordinator llama a enqueueEvent() para agregar eventos parseados a la cola y a popEvent() para extraer eventos de la misma, si es necesario.
# de instancias	Una

Nombre	**DistribuidorEventos**
Descripción	Es el responsable de notificar a los clientes o aplicaciones subscriptas a un evento, que ese evento ha ocurrido.
Interfaces Provistas	IDistribuidorEventos • distributePublication() - dada la ocurrencia de un evento significativo del sistema, notifica a todos los interesados.
Dependencias	• IProcesadorPedidoPersistente • ISubscriptionList • IExternalSubscriberNotifier
Comportamiento	• **ServerEventCoordinator:** A medida que detecta la ocurrencia de eventos significativos del sistema, llamará a distributePublication() para notificar a los interesados. • **SubscriptionList:** Será utilizada por DistribuidorEventos para saber quienes son los interesados en los eventos ocurridos. • **ExternalSubscriberNotifier:** En caso de existir la necesidad de notificar de la ocurrencia de un evento a una aplicación externa, DistribuidorEventos llamará a notifyExternalSubscriber() de ExternalSubscriberNotifier. • **ProcesadorPedidoPersistente:** Para notificar a los clientes de la ocurrencia de un evento en el servidor, DistribuidorEventos llamará a enviarRespuesta() de ProcesadorPedidoPersistente.
# de instancias	Una

Nombre	**SubscriptionList**
Descripción	Es la encargada de almacenar las relaciones entre eventos disponibles del servidor e interesados (clientes o aplicaciones externas).
Interfaces Provistas	ISubscriptionList • addSubscription(observer, event) – Asocia un evento del servidor con un interesado que será notificado cuando tal evento ocurra. • removeSubscription(observer, event) – Elimina la asociación

	del interesado con el evento. • getSubscription(event) – Devuelve todos los subscriptores o interesados en un evento determinado.
Dependencias	• No tiene
Comportamiento	• **PublishSubscribeAdministrator:** Cuando se llama a subscribe() o unsubscribe(), estos almacenarán o eliminarán la relación evento/subscriptor de la SubscriptionList llamando a addSubscription() y removeSubscription(). • **DistribuidorEventos:** Para obtener a los interesados a los que debe notificar, esta componente llama a getSubscription con el evento ocurrido como parámetro.
# de instancias	Una

Nombre	**ExternalSubscriberWebService**
Descripción	Provee un web service que permite a aplicaciones externas subscribirse y desubscribirse a eventos del servidor.
Interfaces Provistas	IExternalSubscriberWebService • subscribeExternal() - permite a una aplicación externa subscribirse a un evento del servidor • unsubscribeExternal() - permite a una aplicación externa eliminar su subscripción a un evento del servidor.
Dependencias	• IPublishSubscribeAdministrator
Comportamiento	• **PublishSubscribeAdministrator:** Cuando una aplicación externa intenta subscribirse o desubscribirse a un evento del servidor utilizando el web service provisto para tal fin, se llama a subscribe() y unsubscribe() de PublishSubscribeAdministrator respectivamente.
# de instancias	Una

Nombre	**ExternalPublisherWebService**
Descripción	Provee un web service que permite a aplicaciones externas publicar eventos en el servidor.
Interfaces Provistas	IExternalPublisherWebService • notify() - permite a una aplicación externa notificar al servidor de la ocurrencia de un evento.
Dependencias	• IPublishSubscribeAdministrator
Comportamiento	• **PublishSubscribeAdministrator:** Cuando una aplicación externa intenta publicar un evento en el servidor, a través del web service provisto para tal fin, se llama a publish() de PublishSubscribeAdministrator.
# de instancias	Una

Nombre	**ExternalPublisher**
Descripción	Se conecta a un web service provisto por una aplicación externa para subscribirse a un evento disparado por tal aplicación.
Interfaces Provistas	IExternalPublisher • subscribe() - permite a la aplicación subscribirse a un evento disparado por una aplicación externa. • Unsubscribe() - permite a la aplicación desubscribirse de un evento de una aplicación externa.
Dependencias	• IConectorWebService
Comportamiento	• **PublishSubscribeAdministrator:** Cuando llega un pedido de subscripción a un evento del servidor y tal evento tiene como componente un evento de una aplicación externa, PublishSubscribeAdministrator llama a subscribe() de ExternalPublisher. Cuando ese evento ya no es de interés, se llama a unsubscribe(). • **ConectorWebService:** Para conectarse al web service provisto por la aplicación externa para subscribirse a eventos de esa aplicación, llama a llamarWebService().
# de instancias	Una

Nombre	**ExternalSubscriberNotifier**
Descripción	Se conecta a un web service provisto por una aplicación externa para notificarle la ocurrencia de un evento del servidor.
Interfaces Provistas	IExternalSubscriberNotifier • notifyExternalSubscriber() - permite al servidor, notificar a una aplicación externa de la ocurrencia de un evento.
Dependencias	• IConectorWebService
Comportamiento	• **DistribuidorEventos:** Cuando se detecta la ocurrencia de un evento significativo del servidor, y uno o más de los subscriptores es una aplicación externa, DistribuidorEventos llama a notifyExternalSubscriber() • **ConectorWebService:** Para conectarse al web service provisto por la aplicación externa para recibir notificaciones de eventos del servidor, llama a llamarWebService().
# de instancias	Una

Capítulo 9 - Conclusiones y Trabajo Futuro

En este trabajo, la descripción de casos de uso se realiza usando modelos de análisis teniendo en cuenta patrones de comunicación para RIA. Una suposición fuerte que hacemos en este trabajo es que los casos de uso deben estructurarse de modo que cada caso de uso se describa mediante un solo patrón de comunicación.
Los modelos de análisis fueron probados con los siguientes casos de estudio: aplicación de chat, caso de uso de envío de mensaje ingresado; manejo de tabla de productos editable, caso de uso de edición de celda y almacenamiento de la edición; drag and drop, caso de uso drag de producto de lista de productos y drop del mismo en carrito de compras y almacenamiento de esta operación; trabajo con árbol de información, caso de uso que al seleccionar un nodo lo expande con contenido y sus nodos hijo; aplicación que trabaja con precios de acciones en la bolsa y que depende de una aplicación externa que le notifique de los cambios de precio en las acciones, caso de uso notificar precio de acción y presentar y almacenar el cambio de precio; aplicación que me permite ofertar precios para acciones y notificar a aplicaciones externas de mi oferta, caso de uso hacer oferta y notificarla.
Con estos casos de estudio se ejercitaron los distintos patrones de comunicación, el manejo de eventos compuestos y de eventos simples, y el uso de algunas UI de contenido más complejas como tablas o árboles, y se cubrieron los casos de UI de contenido editables y no editables.
El perfil UML de diseño de arquitectura planteado está orientado a aplicaciones RIA que incluyen las siguientes características: uso intensivo de datos; manejo de eventos simples/compuestos tanto del lado del cliente como del lado del servidor; tratamiento de notificaciones provenientes de aplicaciones externas, notificación de eventos a aplicaciones externas, notificación de eventos suscriptos por otros clientes de la aplicación web; interacción con protocolos de aplicaciones de red (p.ej. correo electrónico, RPC, etc.); manejo de datos volátiles locales en el cliente y manejo de datos persistentes del lado del servidor. Nuestro perfil para arquitecturas RIA se puede usar para todo tipo de aplicaciones internet y para aplicaciones intranet pequeñas y medianas.
El diseño de arquitectura independiente de tecnología fue probado con los mismos casos de estudio que se testeo la notación de modelado de análisis.
El diseño de arquitectura independiente de tecnología tiene las siguientes limitaciones:

- No describe la UI en detalle: no se contempla layout, estilos y presentación estática completa. Solo tiene en cuenta componentes visuales mínimas para poder trabajar, pero no jerarquías de contenedores y organización de estos en páginas. Esta tarea la debe realizar un diseñador de UI usando alguna notación adecuada de presentación.

- No describe el diseño de la base de datos ni las consultas en las fuentes de datos, esto lo debe hacer por separado un diseñador de BD.

Es asombroso e inesperado para nosotros que las transformaciones de análisis a arquitectura RIA independiente de plataforma necesitaran un modelo de marcas bastante sencillo con solo unos pocos tipos de informaciones.
Las transformaciones de modelos de análisis a diseño de arquitectura tanto dinámico como estático fueron probadas con los casos de estudio presentados en el capítulo 7. Los mismos cubren los distintos patrones de comunicación para aplicaciones RIA presentados en este trabajo.
El paradigma orientado a objetos se mostró flexible y suficientemente expresivo para implementar las transformaciones de modelos de este trabajo, aunque el código quedó algo extenso (entre otras razones, esto se debe a que se tiene que incluir código que controle el orden de aplicación de las reglas de transformación). Por el alto riesgo involucrado de resultar inadecuados y el poco tiempo disponible, no se probaron lenguajes de transformación más específicos para realizar nuestras transformaciones. Quisimos usar un enfoque que de partida sabíamos que iba a resultar, aunque el código no resultara ser el ideal en cuanto a longitud y abstracción (p. ej. varios lenguajes de transformación específicos se abstraen de detalles de control de aplicación de reglas de transformación, lo que favorece el tener código menos extenso y más simple).

Algunas limitaciones de la investigación:

- No contempla patrones que generan eventos en el cliente distinto de evento sobre UI (p. ej. evento de timer, evento de notificación de evento producido por otro cliente) que son tratados en el servidor.
- No contempla patrones que tratan eventos solo en el cliente y no en el servidor.

Como trabajo futuro contemplamos las siguientes tareas:

- Generalizar algunos patrones de comunicación para dar más poder expresivo como el client-server e incorporar algunos patrones de comunicación nuevos (p.ej. tratamiento de eventos solo en el cliente).
- Definir mapeos de modelos de arquitectura independiente de tecnología a modelos del mismo tipo pero dependientes de tecnología (por ejemplo para los modelos de componentes específicos de tecnología del capítulo 8).
- Mapeo de modelos de arquitectura independiente de tecnología a algún framework para RIAs; esto es generación de código.

Capítulo 10 - Bibliografía de Referencia

[1] Urbieta, M., Rossi, G., Ginzburg, J., Schwabe, D.: Designing the Interface of Rich Internet Applications. In: Proccedings of Latin American Web Congress 2007: pp. 144-153, IEEE (2007).

[2] Dolog, P., Stage, J, 2007. Designing Interaction Spaces for Rich Internet Applications with UML. In ICWE'07, 7 th Intl. Conf. on Web Engineering, LNCS vol. 4607, pp. 358-363, Springer-Verlag.

[3] Melia, S., Gomez, J., Perez, S. and Diaz, O.: A Model- Driven Development for GWT-Based Rich Internet Applications with OOH4RIA. In: Proceedings of the 8th Intl. Conf. on Web Engineering: pp.13-23, IEEE (2008).

[4] Paternò, F., Santoro, C., Spano. L. D.: MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. ACM Trans. Comput.-Hum. Interact., 16(4): 1-30 (November 2009).

[5] Linaje Trigueros, M.: RUX-method: modelado de interfaces de usuario Web multi- dispositivo, multimedia, interactivas y accesibles.. Sánchez Figueroa, F. dir. (2009).

[6] Filho, O., Ribeiro, J.: UWE-R: An Extension to a Web Engineering Methodology for Rich Internet Applications. WSEAS Trans. Info. Sci. and App. 6(4): 601-610 (Apr. 2009).

[7] Valverde Giromé, F.: OOWS 2.0: Un Método De Ingeniería Web Dirigido Por Modelos Para La Producción De Aplicaciones WEB 2.0. Pastor López, O. dir. (2010).

[8] Brambilla, M., Fraternali, P., Molteni, E.: A Tool for Model-driven Design of Rich Internet Applications based on AJAX. In "Handbook of Research on Web 2.0, 3.0, and X.0: Technologies, Business, and Social Applications", San

Murugesan (ed.), Chapter 31, 2010, pp. 96-118, IGI Global, 2010, ISBN: 9781605663845 (2010).

[9] Fraternali, P., Comai S., Bozzon, A., Toffetti. G.: Engineering rich internet applications with a model-driven approach. ACM Trans. Web 4(2), Article 7, 47 pages (April 2010).

[10] Koch, N., Kozuruba, S, 2012. Requirements Models as First Class Entities in Model-Driven Web Engineering. In ICWE'12, 12th Intl. Conf. on Web Engineering, LNCS vol. 7703: pp. 158-169, Springer Verlag.

[11] Casalánguida, H. and Durán, J. E., 2013. A Method for Integrating Process Description and User Interface Use During Design of RIA Applications. In ICWE'13, 13 th Intl. Conf. on Web Engineering. Springer Verlag.

[12] UML superstructure specification, version 2.3. Object Management Group, 2010. From http://www.omg.org/spec/UML/2.3/

[13] Russ Miles, Kim Hamilton. Learning UML 2.0. "O'Reilly Media, Inc.", Apr 25, 2006 - 290 pages.

[14] Conallen, J.: Building Web Applications with UML. Addison-Wesley (2002).

[15] Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., & Matera, M. Designing Data-Intensive Web Applications, San Francisco, CA, USA: Morgan Kauffmann (2002).

[16] Valderas, P.: A requirements engineering approach for the development of web applications. PHD-thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia (2004).

[17] Escalona, M. J., Koch, N.: Metamodeling the Requirements of Web Systems. In: Proceedings of the 2nd Intl. Conf. on Web Information Systems and Technologies: pp. 267–280, Springer Verlag (2006)

[18] Koch, N., Knapp, A., Zhang, G., Baumeister, H.: UML-Based Web Engineering. An Approach Based on Standards. In Web Engineering: Modelling and Implementing Web Applications, Human Computer Interaction Series, Springer, pp. 157—191 (2008).

[19] Luna, E., R., Escalona, M., J., Rossi, G., 2010. A Requirements Metamodel for Rich Internet Applications. In ICSOFT 2010.

[20] Casalánguida, H. and Durán, J. E. 2011. Requirements Engineering of Web Application Product Lines. In WEBIST 2011, 418-425.

[21] Morales-Chaparro, R.; Linaje, M.; Preciado, J. C.; Sánchez-Figueroa F. MVC Web design patterns and Rich Internet Applications. Proceedings of the Jornadas de Ingenieria del Software y Bases de Datos 2007, Zaragoza.

[22] Ralph F. Grove, Eray Ozkan. The MVC-web Design Pattern. In proceeding of: WEBIST 2011, pp. 127--130. Proceedings of the 7th International Conference on Web Information Systems and Technologies, Noordwijkerhout, The Netherlands, 6-9 May, 2011.

[23] Esposito, D., 2010. Programming Microsoft ASP.NET MVC, Microsoft Press.

[24] Thomas, D., Hansson, D.H., 2007. Agile Web Development with Rails. The Pragmatic Bookshelf.

[25] Apache Struts, http://struts.apache.org/, is a framework for creating enterprise-ready Java web applications

[26] The-M-Project. (2013). Retrieved Jan. 29, 2013, from http:// the-m-project.org/

[27] Pure MVC (stable release, August 2008). Retrived from http://puremvc.org.

[28] Model-Glue. Retrived from http://www.model-glue.com/. Currently Available Version: 3.1.299.

[29] Czarnecki K., Helsen S. Feature-based suvey of model transformation approaches.

[30] IBM Systems Journal - Model-driven software development, Volume 45 Issue 3, July 2006 Pages 621 – 645.

[31] Sendall S., Kozaczynski W. Model transformation: the heart and soul of model-driven softwae development. IEEE Software, Vol. 20 Issue 5, September 2003, page 42—45.

[32] Anneke Kleppe, Jos Warmer, Wim Bast. *MDA Explained: The Model Driven Architecture--Practice and Promise*. Addison-Wesley Pub Co; 1st edition (April 25, 2003)

[33] Stephen J. Mellor, Kendall Scott, Axel Uhl, Dirk Weise. MDA Distilled: Principles of Model-Driven Architecture. Addison Wesley, 2004.

[34] Anthony T. Holdener, III. Ajax: the definitive guide. O'Reilly Media, Inc., 2008.

[35] Federico Kereki. Essential GWT: Building for the Web with Google Web Toolkit 2, 1st edition, Addison-Wesley Professional, August 2010.

[36] Chafic Kazoun, Joey Lott. Programming Flex 3: The Comprehensive Guide to Creating Rich Internet Applications with Adobe Flex, 1 edition, Adobe Dev Library - Imprint of: O'Reilly Media, September 2008

[37] Lambert M Surhone, Mariam T Tennoe, Susan F Henssonow. OPenLaszlo. Betascript Publishing, Mar 29, 2011 - 100 pages.

[38] Grady Booch, James Rumbaugh, Ivar Jacobson. The Unified Modeling Language User Guide – 1st Edition. Publisher: Addison Wesley, Oct 20, 1998.

[39] Tom Pender. UML Bible. Publisher: Wiley; 1st. edition (September 26, 2003)

[40] Grady Booch, James Rumbaugh, Ivar Jacobson. The Unified Modeling Language User Guide – 2nd Edition. Publisher: Addison Wesley, Jul 2004.

[41] Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)

[42] By Deepak Alur, John Crupi, Dan Malks. Core J2EE™ Patterns: Best Practices and Design Strategies, Second Edition Prentice Hall PTR, 2003.

[43] Surbhi, Rama Chawla. Object Serialization Formats and Techniques a Review Global Journal of Computer Science and Technology, Vol 13, No 6-C (2013)

[44] http://www.gwtproject.org/articles/mvp-architecture.html. Chris Ramsdale, Google Developer Relations Updated March 2010

[45] https://www.genuitec.com/generating-enterprise-class-gwt-applications-for-spring/

[46] http://www.adobe.com/devnet/flex/articles/flex_php_architecture.htm

[47] http://www.openlaszlo.org/architecture, OpenLaszlo official documentation.

[48] Dana Moore, Raymond Budd, Edward Benson. Rich Internet Applications: Ajax and Beyond. An Imprint of Wiley 2007.

[49] http://www.adobe.com/products/flex.html

[50] http://www.adobe.com/devnet/flex/articles/introducing_cairngorm.html

[51] http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/

[52] http://jquery.com/

[53] http://script.aculo.us/

[54] http://dojotoolkit.org/

[55] http://prototypejs.org/

Apéndice A

A continuación, proporcionamos la implementación ejecutable de las transformaciones de comunicaciones de análisis a componentes y a comunicaciones de arquitectura independiente de plataforma. La implementación fue realizada utilizando Rails, un marco de trabajo MVC (modelo-vista-controlador) escrito en el lenguaje de programación Ruby.

Ruby es un lenguaje de script orientado a objetos creado en 1994. En Ruby, todo es un objeto. Se le puede asignar propiedades y acciones a toda información y código. La programación orientada a objetos llama a las propiedades *variables de instancia* y las acciones son conocidas como *métodos*. Ruby es considerado un lenguaje flexible, ya que permite a sus usuarios alterarlo libremente. Las partes esenciales de Ruby pueden ser quitadas o redefinidas a placer. Se puede agregar funcionalidad a partes ya existentes. Ruby intenta no restringir al desarrollador. Los bloques de Ruby son también vistos como una fuente de gran flexibilidad. El desarrollador puede anexar una cláusula a cualquier método, describiendo cómo debe actuar. La cláusula es llamada *bloque* y se ha convertido en una de las más famosas funcionalidades para los recién llegados a Ruby que vienen de otros lenguajes imperativos como PHP o Visual Basic. A diferencia de otros lenguajes de programación orientada a objetos, Ruby se caracteriza por su intencional herencia simple. Sin embargo, Ruby incorpora el concepto de módulos (llamados categorías en Objective-C), que son colecciones de métodos.

El lector debe tener en cuenta que la implementación no está completa.

A.i Descripción de la herramienta implementada

La implementación de la herramienta para aplicar las transformaciones puede dividirse en tres partes.

Primero, definimos una estructura de datos para representar visualmente a los diagramas de comunicaciones y de componentes. Los diagaramas de comunicaciones, tanto para análisis como para arquitectura independiente de plataforma, son representados mediante ternas con los siguientes elementos:

- Nombre y estereotipo del objeto emisor del mensaje.
- Número y nombre del mensaje. Opcionalmente con sus condiciones e iteraciones si las tuviere.
- Nombre y estereotipo del objeto receptor del mensaje.

Mediante esta estructura podemos representar la secuencia de mensajes que componen al diagrama.

Los diagramas de componentes, en cambio, fueron representados visualmente listando cada una de sus componentes, agrupados por el paquete (cliente, servidor o administrador eventos servidor) al que pertenencen, con el siguiente formato:

- Estereotipo y nombre de la componente.

- Nombre de la interfaz de la componente.
- Operaciones de la interfaz de la componente.

Y luego de listar las componentes, se listan las relaciones entre ellas, con el siguiente formato:

- Nombre de la componente.
- Lista de interfaces que utiliza, según la relación <<use>>.

A.ii Implementaciones pendientes

- Todos los parámetros de la transformación que corresponden al modelo de marcas, a excepción de StorageTypes.
- Detección del patrón de comunicación que respeta el modelo de análisis tomado como origen para la transformación.

A.iii Diferencias entre el pseudo-código y el código ejecutable

A continuación señalamos algunos puntos claves del pasaje de pseudo-código a código ejecutable.

A.iii.i Transformación de comunicación de análisis a componentes de arquitectura de referencia

- Las funciones de generación de interfaces, que en el pseudo-código están separadas, en el código ejecutable fueron unificadas con las funciones de transformación.
- Las funciones de generación de componentes, que en el pseudo-código están separadas, en el código ejecutable fueron unificadas con las funciones de transformación.

A.iii.ii Transformación de comunicación de análisis a comunicación de arquitectura de referencia

- Las clases AnalysisModel y ArchModel incluyen métodos que permiten darles un formato legible.
- Los paquetes cliente, servidor y administradorEventosServidor se implementaron utilizando el tipo de datos Hash provisto por Ruby.Código ejecutable en lenguaje Ruby utilizando el framework Ruby on Rails

A.iv Código ejecutable

```
class CommunicationController < ApplicationController

  def client_server
    analysis_model = AnalysisModel.new("client-server/project.xml")
    transformation = CommPatternToCommDiagram.new
    transformation.clientServerCommPatternToCommDiagram(analysis_model, true)

    render text: analysis_model.as_text + '<br><br>' +
        transformation.arch_model.as_text
  end

  def client_server_clients
    analysis_model = AnalysisModel.new("client-server-clients/project.xml")
    transformation = ClientServerClientsCommPatternToCommDiagram.new
    transformation.clientServerCommPatternToCommDiagram(analysis_model)

    render text: analysis_model.as_text + '<br><br>' +
        transformation.arch_model.as_text
  end

  def client_server_externalapps
    analysis_model = AnalysisModel.new("client-server-externalapps/project.xml")
    transformation = ClientServerExternalAppsCommPatternToCommDiagram.new
    transformation.clientServerExternalAppsCommPatternToComponentDiagram(analysis_model)

    render text: analysis_model.as_text + '<br><br>' +
        transformation.arch_model.as_text
  end

  def externalapp_server_clients
    analysis_model = AnalysisModel.new("externalapp-server-clients/project.xml")
    transformation = ExternalAppsServerClientsCommPatternToCommDiagram.new
    transformation.externalAppServerClientsCommPatternToComponentDiagram(analysis_model)

    render text: analysis_model.as_text + '<br><br>' +
```

```
      transformation.arch_model.as_text
  end
end

class AnalysisModel
  attr_reader :objects, :messages

  def initialize(file_path)
    @doc = Nokogiri::XML(File.open(file_path))

    self.read_objects
    self.read_messages
  end

  def read_objects
    @objects = {}
    actor_from_xml = @doc.xpath("//InteractionActor")

    actor = AnalysisModelLifeline.new(actor_from_xml[0]["Id"], actor_from_xml[0]["Name"])

    @objects[actor_from_xml[0]["Id"]] = actor

    objects_from_xml = @doc.xpath("//InteractionLifeLine")

    objects_from_xml.each do |object|
      @objects[object["Id"]] = AnalysisModelLifeline.new([object["Id"]], object["Name"])
    end

    return @objects
  end

  def find_terna(message_name)
    found = nil
    self.messages.each do |message_number, terna|
      if terna.message.name == 'message_name'
        found = terna
      end
```

```
    end

    return found
  end

  def read_messages
    @messages = {}

    messages_from_xml = @doc.xpath("//ModelRelationshipContainer//Message")

    iteration = 0
    messages_from_xml.each do |message|
      message_source = message.xpath(".//FromEnd/MessageEnd")
      message_destination = message.xpath(".//ToEnd/MessageEnd")
      message_object = Message.new(message["Id"], message["Name"], message['SequenceNumber'])

      src_element_id = message_source.attr("EndModelElement").to_s
      dst_element_id = message_destination.attr("EndModelElement").to_s

      src_lifeline = @objects[src_element_id]
      dst_lifeline = @objects[dst_element_id]

      constraint_elements = @doc.xpath("//ConstraintElement")
      constraint_elements.each do |element|
                                                    constrained_message =
element.xpath(".//ConstrainedElements/Message[@Idref=\"#{message_object.id}\"]")
        if constrained_message.count > 0
          message_object.condition = element["Name"]
        end
      end

      @messages[message['SequenceNumber']] = Terna.new(src_lifeline, message_object, dst_lifeline)
      iteration += 1
    end

    return @messages
  end
```

```
  def as_text
    to_return = []
    @messages.keys.sort.each do |id|
      terna = @messages[id]
      to_return.push(terna.to_s)
    end
    return to_return.join("<br>")
  end
end

class AnalysisModelLifeline
  attr_reader :id, :name, :multiobject

  def initialize(id, name)
    @id = id
    @name = name
  end

  def name_with_stereotype
    @name
  end

  def get_name
    if self.name.include?('boundary')
      return self.name.sub('<<boundary>> ', '')
    elsif self.name.include?('control')
      return self.name.sub('<<control>> ', '')
    elsif self.name.include?('entity')
      return self.name.sub('<<entity>> ', '')
    end
  end

  def is_boundary_ui?
    name.match(/boundary/).present? && name.match(/UI/).present?
  end

  def is_boundary_timer?
```

```
    name.match(/boundary/).present? && name.match(/Timer/).present?
  end

  def is_boundary_external_app?
    name.match(/boundary/).present? && name.match(/ExternalApp/).present?
  end

  def is_control?
    name.match(/control/).present?
  end

  def is_entity?
    name.match(/entity/).present?
  end
end

class ArchModelCommunications
  attr_reader :objects, :messages

  def initialize
    @objects = []
    @messages = {}
  end

  def as_text
    to_return = []
    @messages.keys.sort{ |x, y| Message.sort_messages(x, y) }.each do |id|
      terna = @messages[id]
      to_return.push(terna.to_s)
    end
    return to_return.join("<br>")
  end

  def find_component_by_name(component_name)
    found = nil
    objects.each do |lifeline|
      if lifeline.name == component_name
```

```
        found = lifeline
      end
    end

    return found
  end

  def find_component_by_stereotype(stereotype_name)
    found = []
    objects.each do |lifeline|
      if lifeline.stereotype == stereotype_name
        found << component
      end
    end

    return found
  end
end

class ArchModelComponents
  attr_reader :client_package, :server_package, :admin_eventos_servidor_package

  def initialize
    @client_package = []
    @server_package = []
    @admin_eventos_servidor_package = []
  end

  def exists?(component_name)
    find_component_by_name(component_name, @client_package).present? ||
        find_component_by_name(component_name, @server_package).present? ||
        find_component_by_name(component_name, @admin_eventos_servidor_package).present?
  end

  def as_text
    output = []
    output << '<h2>Client Package</h2>'
```

```
    self.client_package.each do |component|
      output << component.to_s
      output << '<hr>'
    end

    output << '<h2>Server Package</h2>'

    self.server_package.each do |component|
      output << component.to_s
      output << '<hr>'
    end

    output << '<h2>AdministradorEventosServidor Package</h2>'

    self.admin_eventos_servidor_package.each do |component|
      output << component.to_s
      output << '<hr>'
    end

    return output.join('<br>')
  end

  def find_component_by_name(component_name, package)
    found = nil
    package.each do |component|
      if component.name == component_name
        found = component
      end
    end

    return found
  end

  def find_component_by_stereotype(stereotype_name, package)
    found = []
    package.each do |component|
```

```
      if component.stereotype == stereotype_name
        found << component
      end
    end

    return found
  end
end

class ArchModelLifeline
  attr_reader :name, :stereotype, :multiobject

  def initialize(name, stereotype)
    @name = name
    @stereotype = stereotype
    @multiobject = false
  end

  def set_multiobject
    @multiobject = true
  end

  def name_with_stereotype
    return '<<' + @stereotype + '>> ' + @name
  end
end

class ClientServerClientsCommPatternToCommDiagram < CommPatternToCommDiagram

  attr_reader :arch_model

  def initialize
    @arch_model = ArchModelCommunications.new
    @helpers = CommPatternToComponentDiagramHelpers.new
    @objects_map = ObjectMap.new
    @message_number = '1'
    @current_package = 'client'
```

```
    @subscriber_response_generated = false
    @starting_event = nil
  end

  def clientServerClientsCommPatternToComponentDiagram(analysis_model)
    analysis_model.messages.keys.sort{ |x,y| Message.sort_messages(x, y) }.each do |message_number|
      terna = analysis_model.messages[message_number]
      if terna.message.is_emit_message?
        emitMessageToModArqEmitMessage(terna)
      elsif terna.message.is_doon_message?
        doOnMessageToModArqMessages(terna)
      elsif terna.message.is_get_message? || terna.message.is_save_message?
        getSaveMessageToModArqMessages(terna)
      elsif terna.message.is_control_to_boundary_ui?
        controlToBoundaryUIMessageToModArqMessages(terna)
      end
    end
  end

  def doOnMessageToModArqMessages(terna)
    client_controller = ArchModelLifeline.new('ClientController', 'ClientController')
    async_remote_request = ArchModelLifeline.new('AsyncRemoteRequest', 'AsyncRemoteRequest')
    webserver = ArchModelLifeline.new('Webserver', 'Webserver')
    server_controller = ArchModelLifeline.new('ServerController', 'ServerController')
            procesador_pedido_persistente = ArchModelLifeline.new('ProcesadorPedidoPersistente',
'ProcesadorPedidoPersistente')
        publish_subscribe_administrator = ArchModelLifeline.new('PublishSubscribeAdministrator',
'PublishSubscribeAdministrator')
    distribuidor_eventos = ArchModelLifeline.new('DistribuidorEventos', 'DistribuidorEventos')

    @objects_map.add(terna.dst_lifeline, client_controller)
    @objects_map.add(terna.dst_lifeline, async_remote_request)
    @objects_map.add(terna.dst_lifeline, webserver)
    @objects_map.add(terna.dst_lifeline, server_controller)
    @objects_map.add(terna.dst_lifeline, procesador_pedido_persistente)
    @objects_map.add(terna.dst_lifeline, publish_subscribe_administrator)
    @objects_map.add(terna.dst_lifeline, distribuidor_eventos)
```

```
    if @helpers.has_target_in_name?(terna.message)
      ui_visual_component_name = @helpers.get_target_name(terna.message)
      ui_visual_component = @objects_map.get(terna.src_lifeline, ui_visual_component_name)
      message = Message.new('', terna.message.name)
      self.generateArchModTerna(ui_visual_component, message, client_controller)
    else
      ui_visual_component_name = nil
    end

    message = Message.new('', terna.message.name)
    self.generateArchModTerna(client_controller, message, async_remote_request)

    message = Message.new('', 'httpRequest')
    self.generateArchModTerna(async_remote_request, message, webserver)

    message = Message.new('', 'processRequest')
    self.generateArchModTerna(webserver, message, server_controller)

    message = Message.new('', 'procesarPedido')
    self.generateArchModTerna(server_controller, message, procesador_pedido_persistente)

    message = Message.new('', 'publish')
                                self.generateArchModTerna(procesador_pedido_persistente,  message,
publish_subscribe_administrator)

    message = Message.new('', 'distributePublication')
    self.generateArchModTerna(publish_subscribe_administrator, message, distribuidor_eventos)

    # guardar el evento ocurrido para luego poder generar los doOn de los receptores del evento
    @starting_event = @helpers.get_event_name(terna.message)
  end

  def getSaveMessageToModArqMessages(terna)
    if terna.message.is_save_message?
      distribuidor_eventos = @objects_map.get(terna.src_lifeline, 'DistribuidorEventos')
      notifications_history = ArchModelLifeline.new('NotificationsHistory', 'DAO')
      message = Message.new('', terna.message.name)
      message.iteration = terna.message.iteration
```

```
      message.condition = terna.message.condition
      self.generateArchModTerna(distribuidor_eventos, message, notifications_history)

                        data_source_connector = ArchModelLifeline.new('DataSourceConnector',
'DataSourceConnector')
      message = Message.new('', 'insertQuery')
      self.generateArchModTerna(notifications_history, message, data_source_connector)
    elsif terna.message.is_get_message?
      distribuidor_eventos = @objects_map.get(terna.src_lifeline, 'DistribuidorEventos')
      subscription_list = ArchModelLifeline.new('SubscriptionList', 'SubscriptionList')
      message = Message.new('', terna.message.name)
      self.generateArchModTerna(distribuidor_eventos, message, subscription_list)

                        data_source_connector = ArchModelLifeline.new('DataSourceConnector',
'DataSourceConnector')
      message = Message.new('', 'selectQuery')
      self.generateArchModTerna(subscription_list, message, data_source_connector)

      message = Message.new('', 'result(subscriptors)')
      self.generateArchModTerna(subscription_list, message, distribuidor_eventos)
    end
  end

  def controlToBoundaryUIMessageToModArqMessages(terna)
    distribuidor_eventos = @objects_map.get(terna.src_lifeline, 'DistribuidorEventos')
          procesador_pedido_persistente = ArchModelLifeline.new('ProcesadorPedidoPersistente2',
'ProcesadorPedidoPersistente')
    procesador_pedido_persistente.set_multiobject
    browser = ArchModelLifeline.new('Browser2', 'Browser')
    browser.set_multiobject
    async_remote_request = ArchModelLifeline.new('AsyncRemoteRequest', 'AsyncRemoteRequest')
    async_remote_request.set_multiobject
    client_controller = ArchModelLifeline.new('ClientController2', 'ClientController')
    client_controller.set_multiobject

    if !@subscriber_response_generated
      message = Message.new('', 'enviarRespuesta')
      message.iteration = terna.message.iteration
```

```
      message.condition = terna.message.condition
      self.generateArchModTerna(distribuidor_eventos, message, procesador_pedido_persistente)
      message = Message.new('', 'respuestaHTTP')
      self.generateArchModTerna(procesador_pedido_persistente, message, browser)
      message = Message.new('', 'httpResponse')
      self.generateArchModTerna(browser, message, async_remote_request)
      message = Message.new('', 'doOn' + @starting_event + 'Received')
      self.generateArchModTerna(async_remote_request, message, client_controller)
      @subscriber_response_generated = true
    end
    ui_visual_component_name = @helpers.get_target_name(terna.message)
                    ui_visual_component      =      ArchModelLifeline.new(ui_visual_component_name,
'UIVisualComponent')
    ui_visual_component.set_multiobject
    message = Message.new('', terna.message.name)
    self.generateArchModTerna(client_controller, message, ui_visual_component)
  end

  def returnFromServer(terna)
    if (@current_package == 'server')
      server_controller =  @objects_map.get(terna.src_lifeline, 'ServerController')
      browser = ArchModelLifeline.new('Browser', 'Browser')
      async_remote_request = @objects_map.get(terna.src_lifeline, 'AsyncRemoteRequest')
      client_controller = @objects_map.get(terna.src_lifeline, 'ClientController')

      message = Message.new('', 'httpResponse')
      self.generateArchModTerna(server_controller, message, browser)

      message = Message.new('', 'httpResponse')
      self.generateArchModTerna(browser, message, async_remote_request)

      message = Message.new('', message.name_without_args + '_Done')
      self.generateArchModTerna(async_remote_request, message, client_controller)
    end
    @current_package = 'client'
  end

  def generateArchModTerna(src, message, dst)
```

```
    mod_arch_terna = Terna.new(src, message, dst)
    message.number = self.getNextArchModMessageNumber(mod_arch_terna)
    @arch_model.messages[message.number] = mod_arch_terna

    return mod_arch_terna
  end

  def getPreviousMessage
    previous_message_number = @arch_model.messages.keys[@arch_model.messages.keys.count - 1]
    return @arch_model.messages[previous_message_number]
  end
end

class ClientServerExternalAppsCommPatternToCommDiagram < ClientServerClientsCommPatternToCommDiagram

  attr_reader :arch_model

  def initialize
    @arch_model = ArchModelCommunications.new
    @helpers = CommPatternToComponentDiagramHelpers.new
    @objects_map = ObjectMap.new
    @message_number = '1'
    @current_package = 'client'
    @subscriber_response_generated = false
    @starting_event = nil
  end

  def clientServerExternalAppsCommPatternToComponentDiagram(analysis_model)
    analysis_model.messages.keys.sort{ |x,y| Message.sort_messages(x, y) }.each do |message_number|
      terna = analysis_model.messages[message_number]
      if terna.message.is_emit_message?
        emitMessageToModArqEmitMessage(terna)
      elsif terna.message.is_doon_message?
        doOnMessageToModArqMessages(terna)
      elsif terna.message.is_get_message? || terna.message.is_save_message?
        getSaveMessageToModArqMessages(terna)
      elsif terna.message.is_notify_message?
```

```
        controlToBoundaryExternalAppsToModArqMessages(terna)
      end
    end
  end

  def controlToBoundaryExternalAppsToModArqMessages(terna)
    distribuidor_eventos = @objects_map.get(terna.src_lifeline, 'DistribuidorEventos')
    external_app_interface = ArchModelLifeline.new('ExternalAppInterface', 'ExternalAppInterface')
    external_app_interface.set_multiobject

    message = Message.new('', terna.message.name)
    self.generateArchModTerna(distribuidor_eventos, message, external_app_interface)
  end
end

class CommPatternToCommDiagram

  attr_reader :arch_model

  def initialize
    @arch_model = ArchModelCommunications.new
    @helpers = CommPatternToComponentDiagramHelpers.new
    @objects_map = ObjectMap.new
    @message_number = '1'
    @current_package = 'client'
  end

  def clientServerCommPatternToCommDiagram(analysis_model, composed_events = false)
    @composed_events = composed_events

    analysis_model.messages.keys.sort{ |x,y| Message.sort_messages(x, y) }.each do |message_number|
      terna = analysis_model.messages[message_number]
      if terna.message.is_emit_message?
        emitMessageToModArqEmitMessage(terna)
      elsif terna.message.is_doon_message?
        doOnMessageToModArqMessages(terna)
      elsif terna.message.is_get_message? || terna.message.is_save_message?
```

```
      getSaveMessageToModArqMessages(terna)
    elsif terna.message.is_control_to_boundary_ui?
      controlToBoundaryUIMessageToModArqMessages(terna)
    end
  end
end

def clientServerClientsCommPatternToComponentDiagram(analysis_model)
  analysis_model.messages.keys.sort{ |x,y| Message.sort_messages(x, y) }.each do |message_number|
    terna = analysis_model.messages[message_number]
    if terna.message.is_emit_message?
      emitMessageToModArqEmitMessage(terna)
    elsif terna.message.is_doon_message?
      doOnMessageToModArqMessages(terna)
    elsif terna.message.is_get_message? || terna.message.is_save_message?
      getSaveMessageToModArqMessages(terna)
    elsif terna.message.is_control_to_boundary_ui?
      controlToBoundaryUIMessageToModArqMessages(terna)
    end
  end
end

def emitMessageToModArqEmitMessage(terna)
  actor = ArchModelLifeline.new(terna.src_lifeline.name, 'Actor')
  if @helpers.has_target_in_name?(terna.message)
    lifeline_name = @helpers.get_target_name(terna.message)
  else
    lifeline_name = terna.dst_lifeline.name
  end
  ui_visual_component = ArchModelLifeline.new(lifeline_name, 'UIVisualComponent')
  message = Message.new('', terna.message.name)
  self.generateArchModTerna(actor, message, ui_visual_component)

  if @composed_events
    event_name = @helpers.get_event_name(message).sub('Emit_', '')
    event_coodinator = ArchModelLifeline.new('EventCoordinator', 'EventCoordinator')
    @objects_map.add(terna.dst_lifeline, event_coodinator)
```

```
      message = Message.new('', "addEvent(#{event_name})")
      self.generateArchModTerna(ui_visual_component, message, event_coodinator)
    end

    @objects_map.add(terna.dst_lifeline, ui_visual_component)
  end

  def doOnMessageToModArqMessages(terna)
    event_coodinator = @objects_map.get(terna.src_lifeline, 'EventCoordinator')
    client_controller = ArchModelLifeline.new('ClientController', 'ClientController')
    async_remote_request = ArchModelLifeline.new('AsyncRemoteRequest', 'AsyncRemoteRequest')
    webserver = ArchModelLifeline.new('Webserver', 'Webserver')
    server_controller = ArchModelLifeline.new('ServerController', 'ServerController')

    @objects_map.add(terna.dst_lifeline, client_controller)
    @objects_map.add(terna.dst_lifeline, async_remote_request)
    @objects_map.add(terna.dst_lifeline, webserver)
    @objects_map.add(terna.dst_lifeline, server_controller)

    if event_coodinator.present?
      message = Message.new('', terna.message.name)
      self.generateArchModTerna(event_coodinator, message, client_controller)
    else
      if @helpers.has_target_in_name?(terna.message)
        ui_visual_component_name = @helpers.get_target_name(terna.message)
      else
        ui_visual_component_name = nil
      end
      ui_visual_component = @objects_map.get(terna.src_lifeline, ui_visual_component_name)
      message = Message.new('', terna.message.name)
      self.generateArchModTerna(ui_visual_component, message, client_controller)
    end

    message = Message.new('', 'doRequest')
    self.generateArchModTerna(client_controller, message, async_remote_request)

    message = Message.new('', 'httpRequest')
```

```
    self.generateArchModTerna(async_remote_request, message, webserver)

    message = Message.new('', 'processRequest')
    self.generateArchModTerna(webserver, message, server_controller)
  end

  def getSaveMessageToModArqMessages(terna)
    server_controller =  @objects_map.get(terna.src_lifeline, 'ServerController')
    dao = ArchModelLifeline.new(@helpers.get_persistent_entity_name(terna.dst_lifeline), 'DAO')
    data_source_connector = ArchModelLifeline.new('DataSourceConnector', 'DataSourceConnector')

    message = Message.new('', terna.message.name)
    self.generateArchModTerna(server_controller, message, dao)

    data_source_connector_interface = ComponentInterface.new('DataSourceConnector')
    if message.name.include?('add')
      data_source_connector_operation = 'insertQuery'
    elsif message.name.include?('save')
      data_source_connector_operation = 'updateQuery'
    elsif message.name.include?('get')
      data_source_connector_operation = 'selectQuery'
    elsif message.name.include?('remove')
      data_source_connector_operation = 'deleteQuery'
    end
    message = Message.new('', data_source_connector_operation)
    self.generateArchModTerna(dao, message, data_source_connector)

    @current_package = 'server'
  end

  def controlToBoundaryUIMessageToModArqMessages(terna)
    self.returnFromServer(terna)

    client_controller =  @objects_map.get(terna.src_lifeline, 'ClientController')

    if @helpers.has_target_in_name?(terna.message)
      lifeline_name = @helpers.get_target_name(terna.message)
```

```
    else
      lifeline_name = terna.dst_lifeline.get_name
    end
    ui_visual_component = @objects_map.get(terna.src_lifeline, lifeline_name)
    if ui_visual_component.nil?
      ui_visual_component = ArchModelLifeline.new(lifeline_name, 'UIVisualComponent')
    end

    message = Message.new('', terna.message.name)
    self.generateArchModTerna(client_controller, message, ui_visual_component)
  end

  def returnFromServer(terna)
    if (@current_package == 'server')
      server_controller =  @objects_map.get(terna.src_lifeline, 'ServerController')
      browser = ArchModelLifeline.new('Browser', 'Browser')
      async_remote_request = @objects_map.get(terna.src_lifeline, 'AsyncRemoteRequest')
      client_controller = @objects_map.get(terna.src_lifeline, 'ClientController')

      message = Message.new('', 'httpResponse')
      self.generateArchModTerna(server_controller, message, browser)

      message = Message.new('', 'httpResponse')
      self.generateArchModTerna(browser, message, async_remote_request)

      message = Message.new('', message.name_without_args + '_Done')
      self.generateArchModTerna(async_remote_request, message, client_controller)
    end
    @current_package = 'client'
  end

  def getNextArchModMessageNumber(current_message)
    previous_message = self.getPreviousMessage
    if previous_message.present?
      nested_message_patterns = [
        ['UIVisualComponent', 'ClientController', 'AsyncRemoteRequest'],
        ['UIVisualComponent', 'EventCoordinator', 'ClientController'],
```

```
      ['EventCoordinator', 'ClientController', 'AsyncRemoteRequest'],
      ['ClientController', 'AsyncRemoteRequest', 'Webserver'],
      ['Webserver', 'ServerController', 'DAO'],
      ['Webserver', 'ServerController', 'ProcesadorPedidoPersistente'],
      ['PublishSubscribeAdministrator', 'DistribuidorEventos', 'SubscriptionList'],
      ['DistribuidorEventos', 'SubscriptionList', 'DataSourceConnector'],
      ['DistribuidorEventos', 'DAO', 'DataSourceConnector'],
      ['ServerController', 'DAO', 'DataSourceConnector'],
      ['AsyncRemoteRequest', 'ClientController', 'UIVisualComponent'],
      ['Browser', 'AsyncRemoteRequest', 'ClientController'],
      ['DataSourceConnector', 'DAO', 'ServerController'],
      ['ServerController', 'ProcesadorPedidoPersistente', 'PublishSubscribeAdministrator'],
      ['ProcesadorPedidoPersistente', 'PublishSubscribeAdministrator', 'DistribuidorEventos']
    ]

    unnested_message_patterns = [
      ['SubscriptionList', 'DistribuidorEventos', 'DAO'],
      ['DAO', 'DistribuidorEventos', 'ProcesadorPedidoPersistente']
    ]

    reset_numeration_message_patterns = [
      ['AsyncRemoteRequest', 'Webserver', 'ServerController'],
      ['ProcesadorPedidoPersistente', 'Browser', 'AsyncRemoteRequest'],
      ['DAO', 'ServerController', 'Browser'],
      ['DAO', 'DistribuidorEventos', 'ExternalAppInterface']
    ]

                    nested_message_candidate = [previous_message.src_lifeline.stereotype,
current_message.src_lifeline.stereotype, current_message.dst_lifeline.stereotype]
    parsed_message_number = @message_number.split('.')
    if nested_message_patterns.include?(nested_message_candidate)
      parsed_message_number << ['1']
    elsif unnested_message_patterns.include?(nested_message_candidate)
      parsed_message_number.pop
                    parsed_message_number[parsed_message_number.count - 1] =
parsed_message_number.last.to_i + 1
    elsif reset_numeration_message_patterns.include?(nested_message_candidate)
      parsed_message_number = [parsed_message_number.first.to_i + 1]
```

```
      else
                                    parsed_message_number[parsed_message_number.count - 1] =
parsed_message_number.last.to_i + 1
      end
      @message_number = parsed_message_number.join('.')
    else
      return '1'
    end
  end

  def generateArchModTerna(src, message, dst)
    mod_arch_terna = Terna.new(src, message, dst)
    message.number = self.getNextArchModMessageNumber(mod_arch_terna)
    @arch_model.messages[message.number] = mod_arch_terna

    return mod_arch_terna
  end

  def getPreviousMessage
    previous_message_number = @arch_model.messages.keys[@arch_model.messages.keys.count - 1]
    return @arch_model.messages[previous_message_number]
  end
end

class CommPatternToComponentDiagram

  attr_reader :arch_model

  def initialize
    @arch_model = ArchModelComponents.new
    @helpers = CommPatternToComponentDiagramHelpers.new
  end

  def clientServerCommPatternToComponentDiagram(analysis_model, storage_types = nil)
    analysis_model.messages.each do |id, terna|
      if terna.dst_lifeline.is_boundary_ui?
        boundaryObject2UiVisualComponents(terna.dst_lifeline, terna.message)
      elsif terna.dst_lifeline.is_control?
```

```
      controlObject2ComponentsForClientServerPattern(terna.dst_lifeline, terna.message)
    elsif terna.dst_lifeline.is_entity?
            if storage_types.is_persistent?(terna.dst_lifeline.get_name) || storage_types.is_both?
(terna.dst_lifeline.get_name)
      entityObject2PersistentStorageComponent(terna.dst_lifeline, terna.message)
            elsif storage_types.is_volatile?(terna.dst_lifeline.get_name) || storage_types.is_both?
(terna.dst_lifeline.get_name)
      entityObject2VolatileStorageComponent(terna.dst_lifeline, terna.message)
      end
    end
  end
end

def clientServerClientsCommPatternToComponentDiagram(analysis_model)
  analysis_model.messages.each do |id, terna|
    if terna.dst_lifeline.is_boundary_ui?
      boundaryObject2UiVisualComponents(terna.dst_lifeline, terna.message)
    elsif terna.dst_lifeline.is_control?
      controlObject2ComponentsForClientServerClientsPattern(terna.dst_lifeline, terna.message)
    elsif terna.dst_lifeline.is_entity?
      if terna.message.is_get_message?
        entityObject2SubscriptionListComponent(terna.dst_lifeline, terna.message)
      else
        entityObject2NotificationsHistoryComponent(terna.dst_lifeline, terna.message)
      end
    end
  end
end

def clientServerExternalAppsCommPatternToComponentDiagram(analysis_model)
  analysis_model.messages.each do |id, terna|
    if terna.dst_lifeline.is_boundary_ui?
      boundaryObject2UiVisualComponents(terna.dst_lifeline, terna.message)
    elsif terna.dst_lifeline.is_control?
      controlObject2ComponentsForClientServerClientsPattern(terna.dst_lifeline, terna.message)
    elsif terna.dst_lifeline.is_entity?
      if terna.message.is_get_message?
        entityObject2SubscriptionListComponent(terna.dst_lifeline, terna.message)
```

```
      else
        entityObject2NotificationsHistoryComponent(terna.dst_lifeline, terna.message)
      end
    elsif terna.dst_lifeline.is_boundary_external_app?
      boundaryObject2ExternalAppInterface(terna.dst_lifeline, terna.message)
    end
  end
end

def externalAppServerClientsCommPatternToComponentDiagram(analysis_model)
  analysis_model.messages.each do |id, terna|
    if terna.dst_lifeline.is_boundary_ui?
      boundaryObject2UiVisualComponents(terna.dst_lifeline, terna.message)
    elsif terna.dst_lifeline.is_control?
      controlObject2ComponentsForClientServerClientsPattern(terna.dst_lifeline, terna.message)
    elsif terna.dst_lifeline.is_entity?
      if terna.message.is_get_message?
        entityObject2SubscriptionListComponent(terna.dst_lifeline, terna.message)
      else
        entityObject2NotificationsHistoryComponent(terna.dst_lifeline, terna.message)
      end
    elsif terna.dst_lifeline.is_boundary_external_app?
      boundaryObject2ExternalAppInterface(terna.dst_lifeline, terna.message)
    end
  end
end

def boundaryObject2UiVisualComponents(object, message)

    if @helpers.has_target_in_name?(message)
      component_name = @helpers.get_target_name(message)
    else
      component_name = object.get_name
    end
    if !@arch_model.exists?(component_name)
      component = Component.new(component_name, 'UIVisualComponent')
      @arch_model.client_package.push(component)
```

```
    else
                        component = arch_model.find_component_by_name(component_name,
arch_model.client_package)
    end
    if message.is_event?
      component_interface = ComponentInterface.new(component_name + 'InteractionInterface')
      component_interface.add_operation(@helpers.get_event_name(message))
    else
      component_interface = ComponentInterface.new(component_name + 'DataInterface')
      component_interface.add_operation(message.name)
    end
    component.add_interface(component_interface)
  end

  def boundaryObject2ExternalAppInterface(object, message)
    component = Component.new('ExternalAppInterface', 'ExternalAppInterface')
    @arch_model.admin_eventos_servidor_package.push(component)
    component_interface = ComponentInterface.new('ExternalAppInterface')
    component_interface.add_operation('notify')
    component.add_interface(component_interface)
  end

  def boundaryObject2TimerComponent(object, message)
    component = Component.new('Timer', 'Timer')
    @arch_model.client_package.push(component)
    component_interface = ComponentInterface.new('Timer')
    component_interface.add_operation('timeout')
    component.add_interface(component_interface)
  end

  def boundaryObject2ProtocoloRedComponent(object, message)
    component = Component.new('ProtocoloRedComponent', 'ProtocoloRedComponent')
    @arch_model.server_package.push(component)
    component_interface = ComponentInterface.new('ProtocoloRedComponent')
    component_interface.add_operation('connect')
    component_interface.add_operation('sendCommandor')
    component_interface.add_operation('disconnect')
    component_interface.add_operation('getResponse')
```

```
    component.add_interface(component_interface)
  end

  def controlObject2ComponentsForClientServerPattern(object, message)
    client_controller_name = @helpers.get_client_controller_name(object)
    if !@arch_model.exists?(client_controller_name)
      client_controller = Component.new(client_controller_name, 'ClientController')
      client_controller_interface = ComponentInterface.new(client_controller_name)
      client_controller_interface.add_operation(message.name)
      client_controller_interface.add_operation(message.name_without_args + '_Done')
      client_controller.add_interface(client_controller_interface)

      @arch_model.client_package.push(client_controller)

      async_remote_request = Component.new('AsyncRemoteRequest', 'AsyncRemoteRequest')
      async_remote_request_interface = ComponentInterface.new('AsyncRemoteRequest')
      async_remote_request_interface.add_operation('doRequest')
      async_remote_request_interface.add_operation('httpResponse')
      async_remote_request.add_interface(async_remote_request_interface)

      @arch_model.client_package.push(async_remote_request)

      server_controller = Component.new('ServerController', 'ServerController')
      server_controller_interface = ComponentInterface.new('ServerController')
      server_controller_interface.add_operation('atenderPedido')
      server_controller.add_interface(server_controller_interface)

      @arch_model.server_package.push(server_controller)
    end
  end

  def controlObject2ComponentsForClientServerClientsPattern(object, message)
    client_controller_name = @helpers.get_client_controller_name(object)
    if !@arch_model.exists?(client_controller_name)
      client_controller = Component.new(client_controller_name, 'ClientController')
      client_controller_interface = ComponentInterface.new(client_controller_name)
      client_controller_interface.add_operation(message.name)
```

```
client_controller_interface.add_operation(message.name_without_args + '_Done')
client_controller.add_interface(client_controller_interface)

@arch_model.client_package.push(client_controller)

async_remote_request = Component.new('AsyncRemoteRequest', 'AsyncRemoteRequest')
async_remote_request_interface = ComponentInterface.new('AsyncRemoteRequest')
async_remote_request_interface.add_operation('doRequest')
async_remote_request_interface.add_operation('httpResponse')
async_remote_request.add_interface(async_remote_request_interface)

@arch_model.client_package.push(async_remote_request)

server_controller = Component.new('ServerController', 'ServerController')
server_controller_interface = ComponentInterface.new('ServerController')
server_controller_interface.add_operation('atenderPedidoPersistente')
server_controller.add_interface(server_controller_interface)

@arch_model.server_package.push(server_controller)

procesador_pedido_persistente = Component.new('ProcesadorPedidoPersistente',
'ProcesadorPedidoPersistente')
procesador_pedido_persistente_interface =
ComponentInterface.new('ProcesadorPedidoPersistente')
procesador_pedido_persistente_interface.add_operation('procesarPedido')
procesador_pedido_persistente_interface.add_operation('enviarRespuesta')
procesador_pedido_persistente.add_interface(procesador_pedido_persistente_interface)

@arch_model.admin_eventos_servidor_package.push(procesador_pedido_persistente)

publish_subscribe_administrator = Component.new('PublishSubscribeAdministrator',
'PublishSubscribeAdministrator')
publish_subscribe_administrator_interface =
ComponentInterface.new('PublishSubscribeAdministrator')
publish_subscribe_administrator_interface.add_operation('publish')
publish_subscribe_administrator.add_interface(publish_subscribe_administrator_interface)

@arch_model.admin_eventos_servidor_package.push(publish_subscribe_administrator)
```

```
    distribuidor_eventos = Component.new('DistribuidorEventos', 'DistribuidorEventos')
    distribuidor_eventos_interface = ComponentInterface.new('DistribuidorEventos')
    distribuidor_eventos_interface.add_operation('distributePublication')
    distribuidor_eventos.add_interface(distribuidor_eventos_interface)

    @arch_model.admin_eventos_servidor_package.push(distribuidor_eventos)
  end
end

def entityObject2PersistentStorageComponent(object, message)
  dao_name = @helpers.get_persistent_entity_name(object)
  if !@arch_model.exists?(dao_name)
    dao = Component.new(dao_name, 'DAO')
    dao_interface = ComponentInterface.new(dao_name)
    dao_interface.add_operation(message.name)
    dao.add_interface(dao_interface)

    @arch_model.server_package << dao

    data_source_connector = Component.new('DataSourceConnector', 'DataSourceConnector')
    data_source_connector_interface = ComponentInterface.new('DataSourceConnector')
    if message.name.include?('add')
      data_source_connector_operation = 'insertQuery'
    elsif message.name.include?('save')
      data_source_connector_operation = 'updateQuery'
    elsif message.name.include?('get')
      data_source_connector_operation = 'selectQuery'
    elsif message.name.include?('remove')
      data_source_connector_operation = 'deleteQuery'
    end
    data_source_connector_interface.add_operation(data_source_connector_operation)
    data_source_connector.add_interface(data_source_connector_interface)

    @arch_model.server_package << data_source_connector
  end
end
```

```
  def entityObject2VolatileStorageComponent(object, message)
    local_store_name = @helpers.get_volatile_entity_name(object)
    if !@arch_model.exists?(local_store_name)
      local_store = Component.new(local_store_name, 'LocalStore')
      local_store_interface = ComponentInterface.new(local_store_name)
      local_store_interface.add_operation(message.name)
      local_store.add_interface(local_store_interface)

      @arch_model.client_package << local_store
    end
  end

  def entityObject2SubscriptionListComponent(object, message)
    subscription_list = Component.new('SubscriptionList', 'SubscriptionList')
    subscription_list_interface = ComponentInterface.new('SubscriptionList')
    subscription_list_interface.add_operation('getSubscriptors')
    subscription_list.add_interface(subscription_list_interface)

    @arch_model.admin_eventos_servidor_package.push(subscription_list)
  end

  def entityObject2NotificationsHistoryComponent(object, message)
    notifications_history = Component.new('NotificationsHistory', 'DAO')
    notifications_history_interface = ComponentInterface.new('NotificationsHistory')
    notifications_history_interface.add_operation('saveHistoryItem')
    notifications_history.add_interface(notifications_history_interface)
    @arch_model.server_package.push(notifications_history)

    data_source_connector = Component.new('DataSourceConnector', 'DataSourceConnector')
    data_source_connector_interface = ComponentInterface.new('DataSourceConnector')
    data_source_connector_interface.add_operation('saveHistoryItem')
    data_source_connector.add_interface(data_source_connector_interface)
    @arch_model.server_package.push(data_source_connector)
  end

  def generate_client_use_relationships
```

```
    use_relationships = {}
    @arch_model.client_package.each do |component|
      case component.stereotype
        when 'UIVisualComponent'
          use_relationships[component.name] = [
                                        @arch_model.find_component_by_stereotype('ClientController',
@arch_model.client_package)[0].interfaces[0]
          ]
        when 'ClientController'
                                                                    ui_visual_components        =
@arch_model.find_component_by_stereotype('UIVisualComponent', @arch_model.client_package)
          use_relationships[component.name] = []
          ui_visual_components.each do |ui_visual_component|
            use_relationships[component.name] << ui_visual_component.interfaces[0]
          end
                                                          use_relationships[component.name]       <<
@arch_model.find_component_by_stereotype('AsyncRemoteRequest',  @arch_model.client_package)
[0].interfaces[0]
        when 'AsyncRemoteRequest'
          use_relationships[component.name] = [
                                        @arch_model.find_component_by_stereotype('ClientController',
@arch_model.client_package)[0].interfaces[0]
          ]
        else
          use_relationships[component.name] = []
      end
    end

    return use_relationships
  end

  def generate_server_use_relationships

    use_relationships = {}
    @arch_model.server_package.each do |component|
      case component.stereotype
        when 'ServerController'
                                                          procesador_pedido_persistente        =
@arch_model.find_component_by_stereotype('ProcesadorPedidoPersistente',
@arch_model.admin_eventos_servidor_package)[0]
```

```
        if procesador_pedido_persistente.present?
          component_interface = procesador_pedido_persistente.interfaces[0]
        else
                    component_interface = @arch_model.find_component_by_stereotype('DAO',
@arch_model.server_package)[0].interfaces[0]
        end
        use_relationships[component.name] = [
          component_interface
        ]
      when 'DAO'
        use_relationships[component.name] = [
                                @arch_model.find_component_by_stereotype('DataSourceConnector',
@arch_model.server_package)[0].interfaces[0]
        ]
      else
        use_relationships[component.name] = []
    end
  end

  return use_relationships
 end

 def generate_administrador_eventos_servidor_use_relationships

  use_relationships = {}
  @arch_model.admin_eventos_servidor_package.each do |component|
    case component.stereotype
      when 'ProcesadorPedidoPersistente'
        use_relationships[component.name] = [
                    @arch_model.find_component_by_stereotype('PublishSubscribeAdministrator',
@arch_model.admin_eventos_servidor_package)[0].interfaces[0]
        ]
      when 'PublishSubscribeAdministrator'
        use_relationships[component.name] = [
                                @arch_model.find_component_by_stereotype('DistribuidorEventos',
@arch_model.admin_eventos_servidor_package)[0].interfaces[0],
                                @arch_model.find_component_by_stereotype('SubscriptionList',
@arch_model.admin_eventos_servidor_package)[0].interfaces[0]
        ]
```

```
        when 'DistribuidorEventos'
          use_relationships[component.name] = [
                              @arch_model.find_component_by_stereotype('ProcesadorPedidoPersistente',
@arch_model.admin_eventos_servidor_package)[0].interfaces[0],
                                        @arch_model.find_component_by_stereotype('SubscriptionList',
@arch_model.admin_eventos_servidor_package)[0].interfaces[0],
                                         @arch_model.find_component_by_name('NotificationsHistory',
@arch_model.server_package).interfaces[0]
          ]
        when 'ExternalAppInterface'
          use_relationships[component.name] = [
                            @arch_model.find_component_by_stereotype('PublishSubscribeAdministrator',
@arch_model.admin_eventos_servidor_package)[0].interfaces[0],
          ]
        else
          use_relationships[component.name] = []
      end
    end

    return use_relationships
  end

  def use_relationships_as_text
    client_use_relationships = self.generate_client_use_relationships
    output = []
    output << '<h2>Client Use Relationships</h2>'
    client_use_relationships.each do |key, use_rel|
      output << "<h4>#{key}</h4>"
      output << "<ul>"
      use_rel.each do |use|
        output << "<li>#{use.name}</li>"
      end
      output << "</ul>"
    end

    output << '<hr>'

    server_use_relationships = self.generate_server_use_relationships
    output << '<h2>Server Use Relationships</h2>'
```

```
    server_use_relationships.each do |key, use_rel|
      if use_rel.length > 0
        output << "<h4>#{key}</h4>"
        output << "<ul>"
        use_rel.each do |use|
          output << "<li>#{use.name}</li>"
        end
        output << "</ul>"
      end
    end

                                                admin_eventos_servidor_use_relationships =
self.generate_administrador_eventos_servidor_use_relationships
    output << '<h2>AdministradorEventosServidor Use Relationships</h2>'
    admin_eventos_servidor_use_relationships.each do |key, use_rel|
      if use_rel.length > 0
        output << "<h4>#{key}</h4>"
        output << "<ul>"
        use_rel.each do |use|
          output << "<li>#{use.name}</li>"
        end
        output << "</ul>"
      end
    end

    return output.join('')
  end
end



class CommPatternToComponentDiagramHelpers

  def has_target_in_name?(message)
    message.name.match(/_on_/).present? ||
      message.name.match(/_To_/).present? ||
      message.name.match(/_From_/).present?
  end
```

```
  def get_target_name(message)
    message.name.split('_').last.gsub(/\(.+\)/, '')
  end

  def get_event_name(message)
    if message.is_emit_message?
      return message.name.split('_').first + '_' + message.name.split('_').second
    else
      return message.name_without_args.sub('doOn', '')
    end
  end

  def get_client_controller_name(lifeline)
    return lifeline.name.sub('EventProcessor', '').sub('<<control>> ', '') + 'Controller'
  end

  def get_persistent_entity_name(lifeline)
    return lifeline.name.sub('Data', '').sub('<<entity>> ', '') + 'DAO'
  end
end

class Component
  attr_reader :name, :stereotype, :interfaces

  def initialize(name, stereotype)
    @name = name
    @stereotype = stereotype
    @interfaces = []
  end

  def add_interface(component_interface)
    @interfaces.push(component_interface)
  end

  def to_s
    require 'htmlentities'
    coder = HTMLEntities.new
```

```
    interfaces = []
    @interfaces.each do |interface|
      interfaces.push(interface.to_s)
    end

    return coder.encode("<<" + @stereotype + ">>" + @name) + '<br>' + interfaces.join('<br>')
  end
end

class ComponentInterface

  attr_reader :name, :operations

  def initialize(name)
    @name = name
    @operations = []
  end

  def add_operation(operation)
    @operations.push(operation)
  end

  def name
    'I' + @name
  end

  def to_s
    output = "<h4>#{self.name}</h4>"
    output += '<ul>'
    @operations.each do |operation|
      output += "<li>#{operation}</li>"
    end
    output += '</ul>'
  end
end

class ExternalAppsServerClientsCommPatternToCommDiagram < ClientServerClientsCommPatternToCommDiagram
```

```
  attr_reader :arch_model

  def initialize
    @arch_model = ArchModelCommunications.new
    @helpers = CommPatternToComponentDiagramHelpers.new
    @objects_map = ObjectMap.new
    @message_number = '1'
    @current_package = 'client'
    @subscriber_response_generated = false
    @starting_event = nil
  end

  def externalAppServerClientsCommPatternToComponentDiagram(analysis_model)
    analysis_model.messages.keys.sort{ |x,y| Message.sort_messages(x, y) }.each do |message_number|
      terna = analysis_model.messages[message_number]
      if terna.message.is_notify_message?
        notifyMessageToModArqEmitMessage(terna)
      elsif terna.message.is_doon_message?
        doOnMessageToModArqMessages(terna)
      elsif terna.message.is_get_message? || terna.message.is_save_message?
        getSaveMessageToModArqMessages(terna)
      elsif terna.message.is_control_to_boundary_ui?
        controlToBoundaryUIMessageToModArqMessages(terna)
      end
    end
  end

  def notifyMessageToModArqEmitMessage(terna)
    actor = ArchModelLifeline.new(terna.src_lifeline.name, 'Actor')

    external_app_interface = ArchModelLifeline.new('ExternalAppInterface', 'ExternalAppInterface')
    message = Message.new('', terna.message.name)
    self.generateArchModTerna(actor, message, external_app_interface)

    @objects_map.add(terna.dst_lifeline, external_app_interface)
  end
```

```
  def doOnMessageToModArqMessages(terna)
    external_app_interface = @objects_map.get(terna.src_lifeline, 'ExternalAppInterface')
        publish_subscribe_administrator = ArchModelLifeline.new('PublishSubscribeAdministrator',
'PublishSubscribeAdministrator')
    distribuidor_eventos = ArchModelLifeline.new('DistribuidorEventos', 'DistribuidorEventos')

    @objects_map.add(terna.dst_lifeline, publish_subscribe_administrator)
    @objects_map.add(terna.dst_lifeline, distribuidor_eventos)

    message = Message.new('', 'publish')
    self.generateArchModTerna(external_app_interface, message, publish_subscribe_administrator)

    message = Message.new('', 'distributePublication')
    self.generateArchModTerna(publish_subscribe_administrator, message, distribuidor_eventos)

    # guardar el evento ocurrido para luego poder generar los doOn de los receptores del evento
    @starting_event = @helpers.get_event_name(terna.message)
  end
end

class Message
  attr_accessor :id, :name, :number, :iteration, :condition

  def initialize(id, name, number = nil)
    @id = id
    @number = number

    self.initialize_name(name)
  end

  def initialize_name(name)
    if /^(\[.+\])(.+)/.match(name).present?
      @iteration, @name = /^(\[.+\]) (.+)/.match(name).captures
      if  self.is_control_to_boundary_ui?
        @condition = '[clientIsOnline(client) OK]'
      else
        @condition = '[clientIsOnline(client) ERROR]'
```

```
    end
  else
    @name = name
  end
end

def name_without_args
  @name.gsub(/\(.+\)/, '')
end

def is_event?
  @name.match(/Emit_/).present?
end

def is_emit_message?
  self.is_event?
end

def is_doon_message?
  @name.match(/doOn/).present?
end

def is_get_message?
  @name.match(/^get/).present?
end

def is_save_message?
  @name.match(/^save/).present?
end

def is_notify_message?
  @name.match(/^notify/).present?
end

def is_control_to_boundary_ui?
  @name.match(/appendTo_/) ||
      @name.match(/appendItem_/) ||
```

```
      @name.match(/append_/) ||
      @name.match(/append/) ||
      @name.match(/pushTo_/) ||
      @name.match(/enqueueTo_/) ||
      @name.match(/removeFrom_/) ||
      @name.match(/removeItem_/) ||
      @name.match(/popFrom_/) ||
      @name.match(/dequeueFrom_/) ||
      @name.match(/get_/) ||
      @name.match(/replaceIn_/) ||
      @name.match(/addContentTo_/) ||
      @name.match(/addSubtreeTo_/) ||
      @name.match(/removeSubtreeFrom_/) ||
      @name.match(/findSubtreeIn_/) ||
      @name.match(/removeNodeFrom_/) ||
      @name.match(/findNodeIn_/) ||
      @name.match(/updateNodeIn_/) ||
      @name.match(/addRegisterTo_/) ||
      @name.match(/removeRegisterFrom_/) ||
      @name.match(/cancelEditIn_/) ||
      @name.match(/getRegisterFrom_/) ||
      @name.match(/updateRegisterIn_/) ||
      @name.match(/setTextTo_/) ||
      @name.match(/setStatus/) ||
      @name.match(/getTextFrom_/) ||
      @name.match(/clearText_/)
  end

  def self.sort_messages(x, y)
    parsed_x = x.split('.')
    parsed_y = y.split('.')

    shortest_one = [parsed_x.count, parsed_y.count].min
    i = 0
    comparison = 0
    while comparison == 0 && i < shortest_one
      comparison = parsed_x[i].to_i <=> parsed_y[i].to_i
```

```
      i = i + 1
    end
    if comparison == 0
      comparison = parsed_x.count <=> parsed_y.count
    end
    if comparison == 0
      comparison = parsed_x <=> parsed_y
    end

    return comparison
  end
end

class ObjectMap

  def initialize
    @objects_map = {}
  end

  def add(analysis_model_lifeline, arch_model_lifeline)
    if !@objects_map[analysis_model_lifeline.get_name].is_a?(Array)
      @objects_map[analysis_model_lifeline.get_name] = []
    end
    @objects_map[analysis_model_lifeline.get_name] << arch_model_lifeline
  end

  def get(analysis_model_lifeline, arch_model_lifeline_name)
    found = nil
    if arch_model_lifeline_name.present?
      @objects_map[analysis_model_lifeline.get_name].each do |arch_model_lifeline|
        if arch_model_lifeline.name == arch_model_lifeline_name
          found = arch_model_lifeline
        end
      end
    end

    found
```

```
  end
end

class StorageTypes
  def initialize(storage_types)
    @storage_types = storage_types
  end

  def is_volatile?(analysis_model_lifeline_name)
    @storage_types[analysis_model_lifeline_name] == 'volatile'
  end

  def is_persistent?(analysis_model_lifeline_name)
    puts analysis_model_lifeline_name.inspect
    @storage_types[analysis_model_lifeline_name] == 'persistent'
  end

  def is_both?(analysis_model_lifeline_name)
    @storage_types[analysis_model_lifeline_name] == 'both'
  end
end

class Terna
  attr_reader :src_lifeline, :message, :dst_lifeline

  def initialize(src_lifeline, message, dst_lifeline)
    @src_lifeline = src_lifeline
    @message = message
    @dst_lifeline = dst_lifeline
  end

  def to_s
    require 'htmlentities'
    coder = HTMLEntities.new
    src_lifeline_multiobject = ''
    dst_lifeline_multiobject = ''
    if self.src_lifeline.multiobject
```

```
    src_lifeline_multiobject = ' (MO)'
  end
  if self.dst_lifeline.multiobject
    dst_lifeline_multiobject = ' (MO)'
  end

                coder.encode("[#{self.src_lifeline.name_with_stereotype}#{src_lifeline_multiobject},
#{self.message.number} - #{self.message.iteration} #{self.message.name} #{self.message.condition},
#{self.dst_lifeline.name_with_stereotype}#{dst_lifeline_multiobject}]")
  end
end
```